Magnum Chaos

Relazione progetto

Programmazione ad oggetti
A.A. 2017/2018
Consegna: 29/04/2018

Davide Conti
Massimo Donini
Andrea Lavista
Ivan Mazzanti

Indice

Capitolo 1
Analisi

1.1 Requisiti

Il team pone come obiettivo la realizzazione di un videogioco, di nome Magnum Chaos, in stile "bullet-hell". In questo genere videoludico il giocatore controlla una navicella che deve affrontare ondate di nemici, cercando di sopravvivere il più a lungo possibile.

Requisiti funzionali:

- La navicella potrà muoversi e sparare nell'ambiente di gioco, con la possibilità di distruggere i nemici.
- I nemici cercheranno di infliggere danni alla navicella dell'utente.
- Vi saranno due tipologie di nemici: attivi (hanno la facoltà di sparare) e passivi (ostacolano la navicella senza sparare).
- I nemici verranno generati frequentemente ed avranno comportamenti variabili per rendere il gioco più avvincente.
- Sarà possibile per l'utente potenziare la propria navicella raccogliendo degli eventuali potenziamenti disseminati dopo la morte dei nemici.
- Vi saranno quattro tipologie di potenziamenti: vita, scudo (invincibilità per N secondi), danno, frequenza di fuoco.
- Durante il corso della partita sarà visibile lo stato attuale della navicella (vita, potenziamenti, etc.) e il tempo trascorso.

Requisiti funzionali opzionali:

- Il giocatore otterrà un punteggio in seguito alla distruzione di ogni nemico.
- Ci saranno varie tipologie di nemici e ostacoli.
- Più modalità di gioco per l'utente.
- Si potrà accedere ad una sezione dove l'utente potrà visualizzare alcune informazioni sulla totalità delle partite giocate (punteggi migliori, sfide).
- Il gioco effettuerà il salvataggio su file delle impostazioni, dei punteggi migliori e delle sfide.

Requisiti non funzionali opzionali:

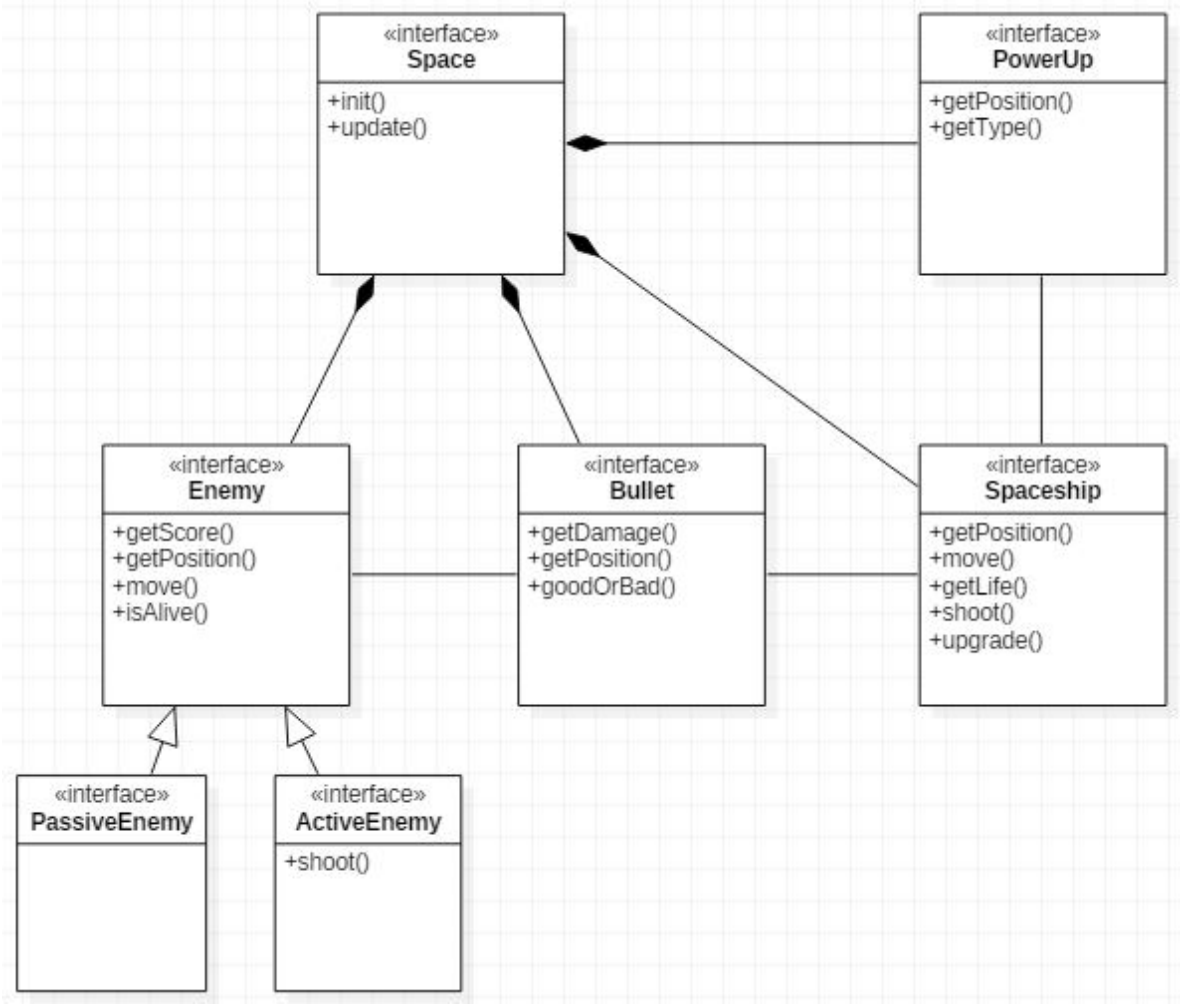
- Raggiungere una soddisfacente fluidità di gioco

1.2 Analisi e modello del dominio

Magnum Chaos è un gioco sparatutto ambientato nello spazio, dove Il giocatore controlla una navicella con la quale può distruggere i nemici, sparando proiettili. I nemici a loro volta cercheranno di abbattere la navicella dell'utente infliggendole danni al contatto o sparando. Quando un nemico viene sconfitto, potrebbe rilasciare un potenziamento al suo posto e, qualora il giocatore raccogliesse questo potenziamento, otterrebbe un aumento ad una delle proprie caratteristiche (vita,danno,etc.).

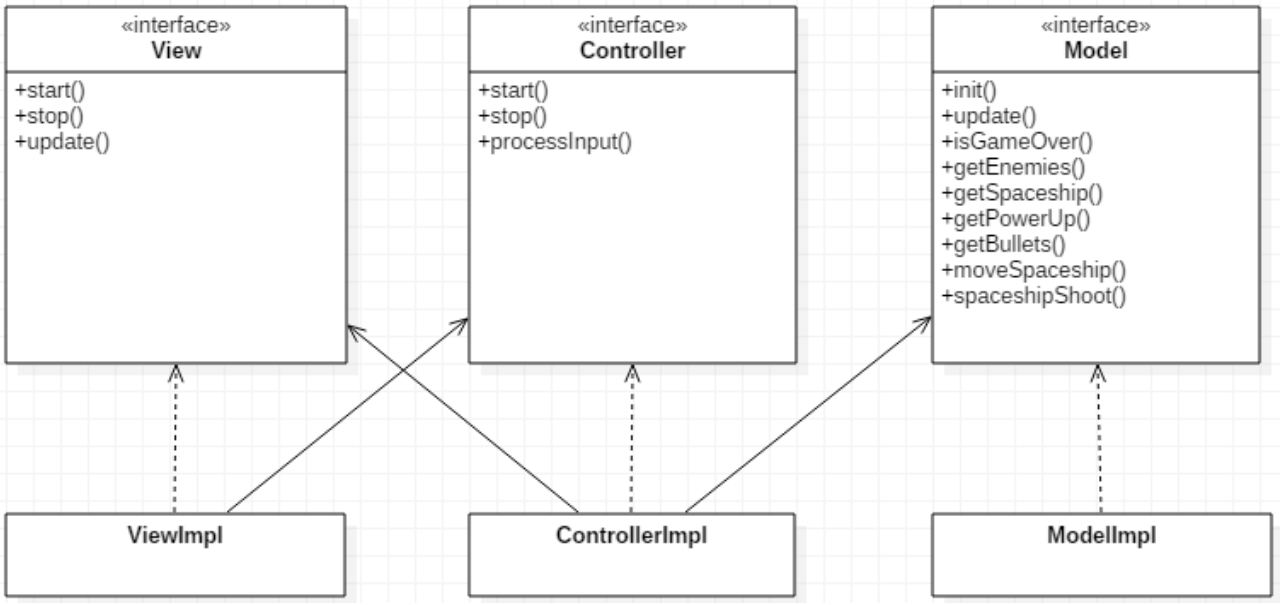
Durante il corso della partita vengono assegnati punti in seguito all'uccisione di un nemico.

La partita termina quando la vita della navicella si esaurisce oppure quando finiscono i nemici.



Capitolo 2
Design

2.1 Architettura



Per la realizzazione di Magnum Chaos abbiamo scelto di utilizzare il pattern architetturale Model-View-Controller. Ciò consente di isolare la logica funzionale che riguarda le tre componenti, le quali non interferiscono tra loro. Così facendo il progetto è meglio organizzato e la modifica di una delle componenti non comporta cambiamenti nelle altre.
Nella nostra modellazione del pattern MVC gli input vengono catturati dalla View, la quale provvederà a notificarli al Controller e successivamente saranno gestiti comunicando opportunamente con il Model. Quest'ultimo contiene l'intera logica dell'applicazione che verrà aggiornata costantemente tenendo anche conto degli input dell'utente. Sarà poi compito della View aggiornare le varie componenti grafiche.

2.2 Design Dettagliato

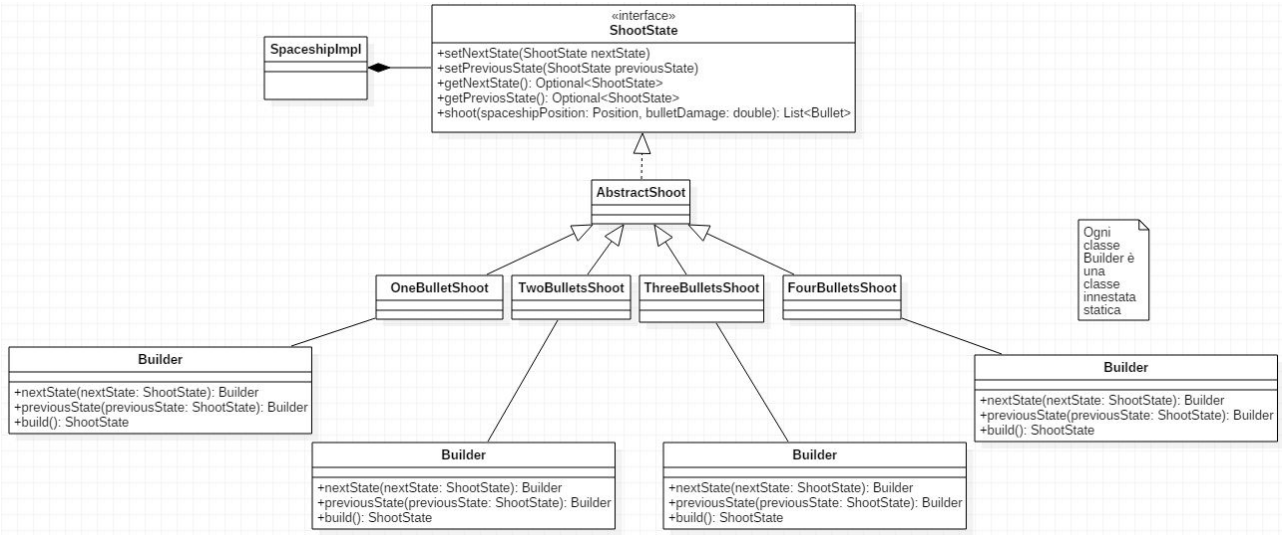
DavideConti

La mia parte è incentrata sulla gestione della navicella del giocatore e sull'input. Sulla parte della navicella spiegherò solamente la parte saliente da me sviluppata.

Questa parte riguarda la gestione dei diversi metodi per sparare. All'inizio della partita il giocatore può sparare con la navicella un solo colpo per volta. Raccogliendo power-up per il danno del proiettile, esso sarà aumentato. Una volta che tale danno è arrivato al massimo, al prossimo power-up raccolto la navicella effettuerà un potenziamento e sparerà due colpi per volta. Lo stesso procedimento si ripete per un numero finito di volte. Inoltre se la navicella viene colpita da un nemico viene depotenziata, ovvero, se sta sparando due colpi tornerà a sparare un solo colpo per volta. Dato che il comportamento con cui effettuare l'aggiornamento dipende dal danno del proiettile ho scelto di implementare i diversi "stati" della navicella utilizzando il pattern **State**. In questo modo non ho dovuto realizzare grandi blocchi di codice per le scelte condizionali. Infatti tale pattern inserisce ogni ramo condizionale in una classe separata. Così facendo tutta la logica specifica di uno stato viene incapsulata in un singolo oggetto. Poiché tutto il codice specifico di uno stato è inserito in una sottoclasse di AbstractShoot, definendo nuove sottoclassi si possono aggiungere facilmente nuovi stati e nuove transizioni.

Per la creazione di ogni oggetto ShootState ho deciso di utilizzare il pattern **Builder**. Questa mia decisione è derivata dal seguente motivo: all'inizio avevo pensato di utilizzare il pattern Abstract factory per inserire all'interno della classe factory tutti i possibili costruttori di ogni singolo oggetto ShootState. Dato che tutti i possibili costruttori per un singolo oggetto ShootState sono 4(il primo non prende parametri, il secondo prende il parametro per settare il prossimo stato, il terzo prende il parametro per settare lo stato precedente e il quarto prende entrambi i parametri) all'interno di questa classe avrei avuto n * 4 metodi, dove n indica il numero di sottoclassi di AbstractShoot. In questa classe avrei avuto un sacco di metodi, perciò ho scelto di utilizzare il pattern builder per risolvere questo problema. Inoltre tale pattern consente un migliore controllo del processo di costruzione: a differenza di altri pattern creazionali che costruiscono un oggetto in un "colpo" solo, questo pattern costruisce l'oggetto passo dopo passo esplicitando quello che si sta costruendo. Infine soltanto quando la costruzione dell'oggetto è terminata il Builder

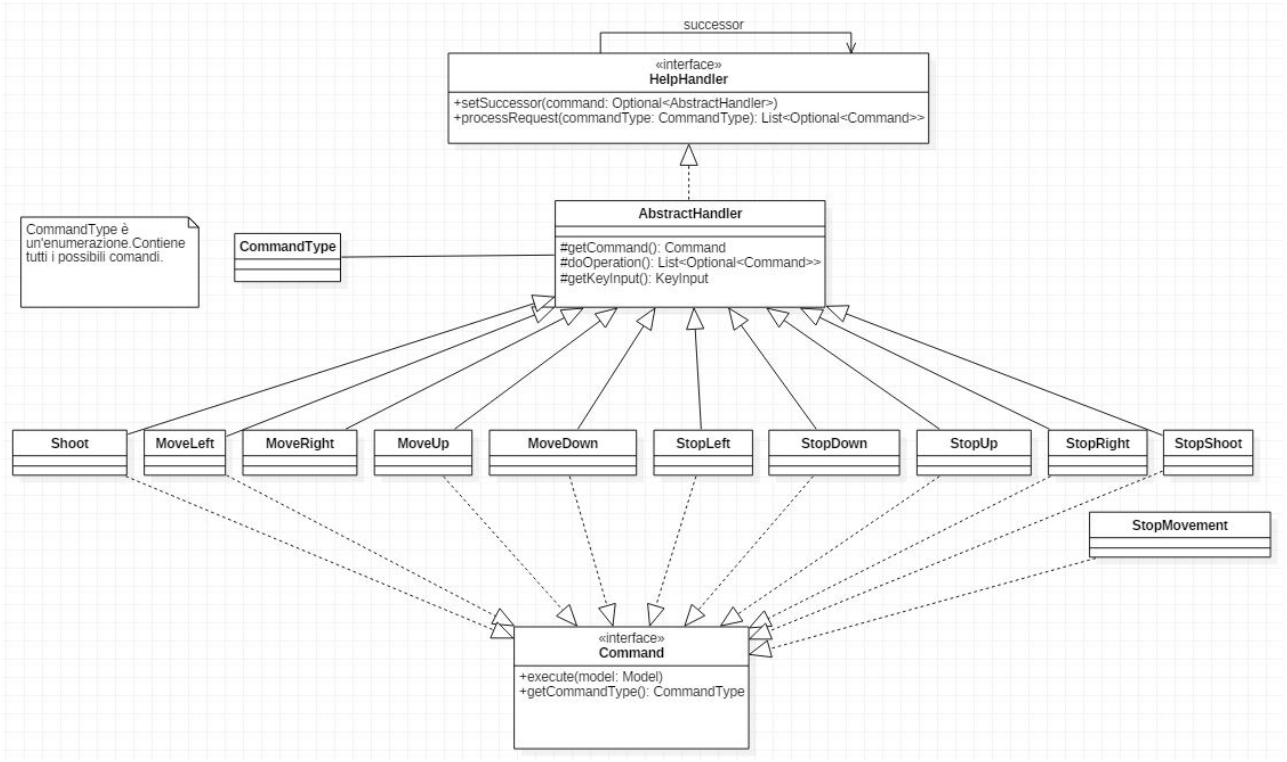
restituirà l'oggetto finale.



Per quanto concerne la gestione dell'input, quando la View rileva che è stato premuto o rilasciato un tasto lo notifica al controller. Il controller deve associare questo evento all'oggetto relativo al comando che dovrà essere eseguito. Per evitare di implementare un grosso switch-case all'interno della classe ControllerImpl ho scelto di utilizzare il pattern **Chain of responsibility**. Questo pattern fornisce una maggior flessibilità nell'assegnare e distribuire le responsabilità tra oggetti. Inoltre fa sì che un oggetto non abbia l'obbligo di sapere quali altri oggetti gestiranno la richiesta. Infatti un oggetto saprà solamente che la richiesta verrà gestita in modo appropriato.

Dato che l'operazione che dovrà essere eseguita, una volta che il tasto premuto/rilasciato è stato associato all'oggetto appropriato, varia a seconda dell'oggetto, ho deciso di utilizzare il pattern **Template method** e quindi creare i metodi astratti protetti doOperation e getCommand (dato che al di fuori della classe e delle sottoclassi non sarebbero stati utilizzati). In questo modo ho implementato la parte invariante dell'algoritmo all'interno della classe AbstractHandler, mentre nelle sue sottoclassi ho implementato gli algoritmi varianti.

Infine per eseguire i comandi, restituiti come risposta dalla chain of responsibility, in particolare dal metodo doOperation, ho deciso di utilizzare il pattern **Command**. Grazie a questo pattern disaccoppio l'oggetto che invoca un'operazione da quello che conosce come portarla a termine. Infine risulta facile aggiungere nuovi comandi poiché non è necessario modificare le classi esistenti.

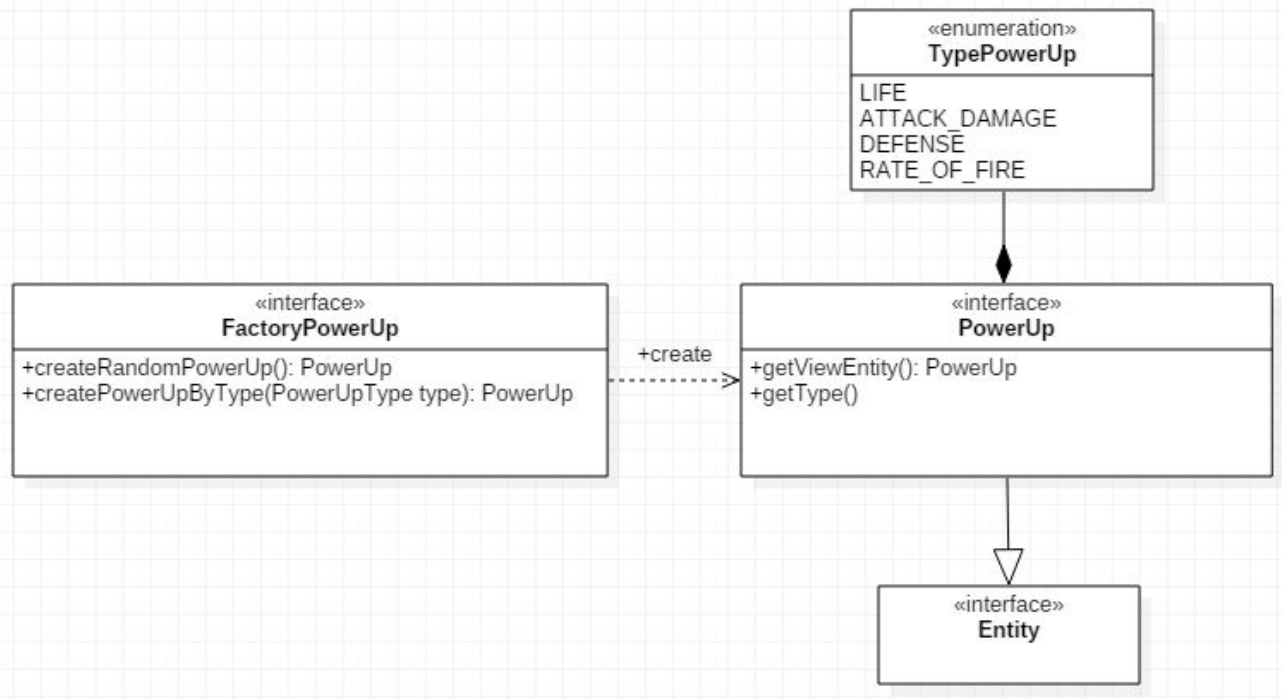


AndreaLavista

Durante la fase di design dettagliato ho progettato le parti del sistema che mi competono: potenziamenti, dati di gioco e dati globali. Di seguito analizzerò più approfonditamente le scelte riguardanti questi ambiti.

Potenziamenti

I potenziamenti (powersups) sono delle entità di gioco che compaiono a seguito della morte dei nemici e potenziano la navicella al contatto. Per la loro progettazione ho ritenuto opportuno l'impiego del pattern **Factory Method**, che preserva una certa flessibilità nella creazione degli oggetti di tipo "PowerUp". L'interfaccia PowerUp estenderà l'interfaccia Entity così come tutte le altre entità del gioco (navicella, nemici e proiettili). Il tutto sarà coadiuvato dall'utilizzo dell'enumerazione PowerUpType il cui fine è indicare le differenti tipologie di potenziamenti previste all'interno del gioco, dato che ognuno di essi comporterà modifiche diverse sulla navicella.

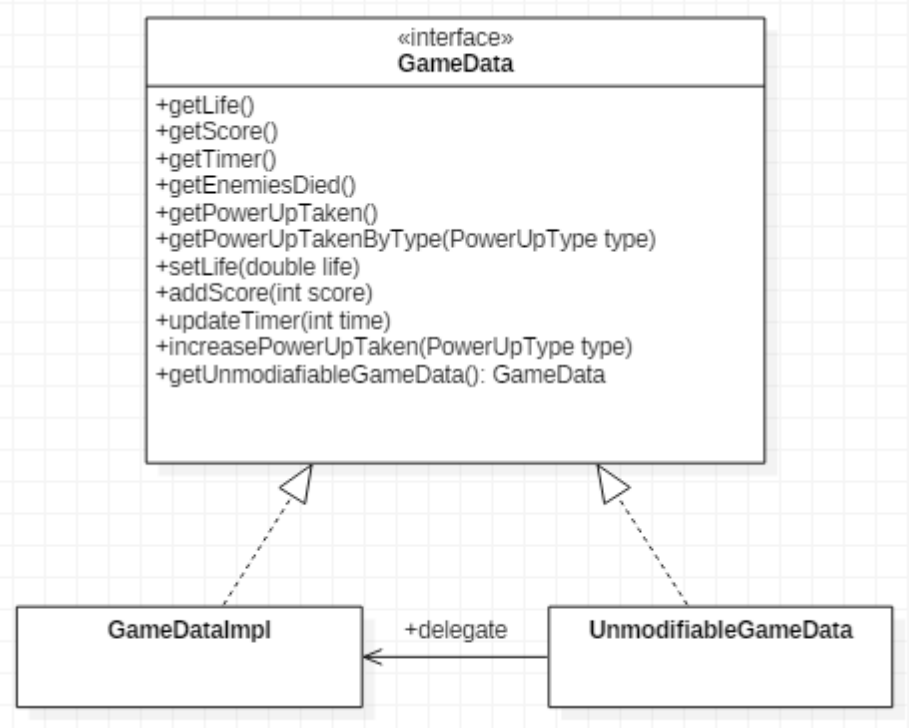


Dati di gioco

I dati di gioco (game data) sono quei dati che verranno visualizzati durante lo svolgimento della partita. Per la loro realizzazione adopererò il pattern **Proxy**. Infatti, lo schema riporta due implementazioni dell'interfaccia GameData:

- GameDataImpl, che contiene i getter e i setter che operano sui dati, la quale verrà usata nel Model per l'inserimento e l'aggiornamento delle informazioni;
- UnmodifiableGameData, ipotizzata per gli accessi in sola lettura che si effettueranno nella View, sulla quale saranno impedite modifiche ai dati.

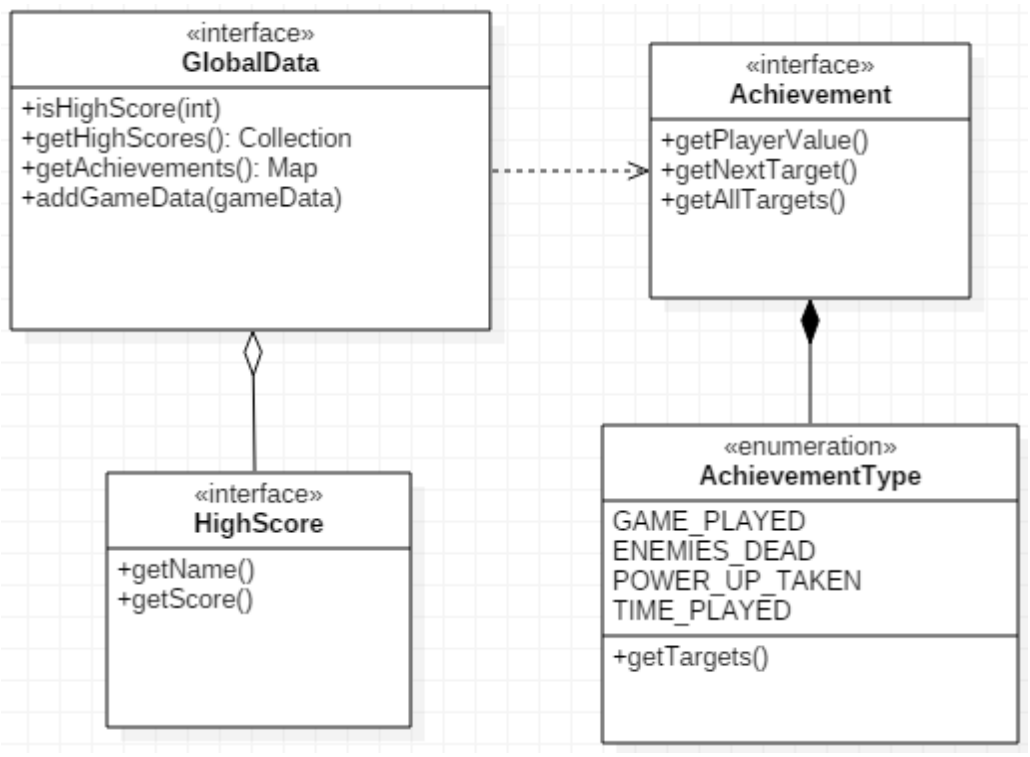
Questa scelta è dovuta al fatto che i dati devono poter essere modificati solo durante l'aggiornamento dello stato del gioco nel Model, mentre la View necessita della sola lettura dei dati e non può alterarli.



Global Data

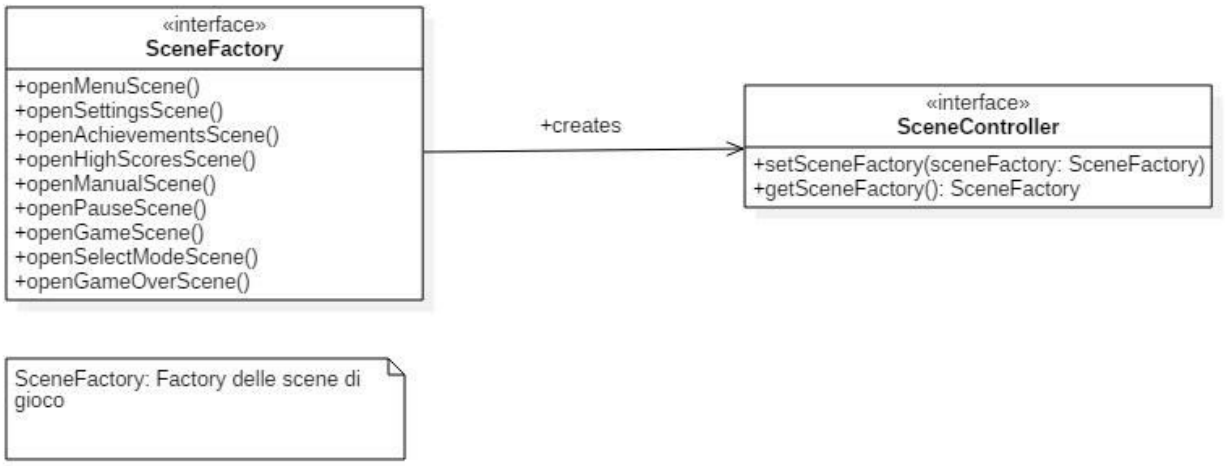
I dati globali del gioco (global data) sono stati previsti per gestire i dati statistici di tutte le partite effettuate, con lo scopo di offrire sfide (achievements) ed una lista dei punteggi migliori (high scores). I dati globali verranno aggiornati al termine di ogni partita, invocando il metodo addGameData, che passerà a GlobalData i dati della partita appena conclusa. L'enumerazione AchievementType invece conterrà tutte le tipologie di sfide, ognuna delle quali avrà più livelli da far raggiungere al giocatore.

Note: La progettazione dei dati globali è stata rieseguita durante la fase di implementazione perché la precedente progettazione si era rivelata poco duttile e creava forti dipendenze tra Model e View. Con l'attuale progettazione invece, modifiche ai livelli delle sfide e aggiunta di nuove sfide, comportano modifiche solo ed esclusivamente alle classi afferenti al Model, lasciando intatte le classi della View.



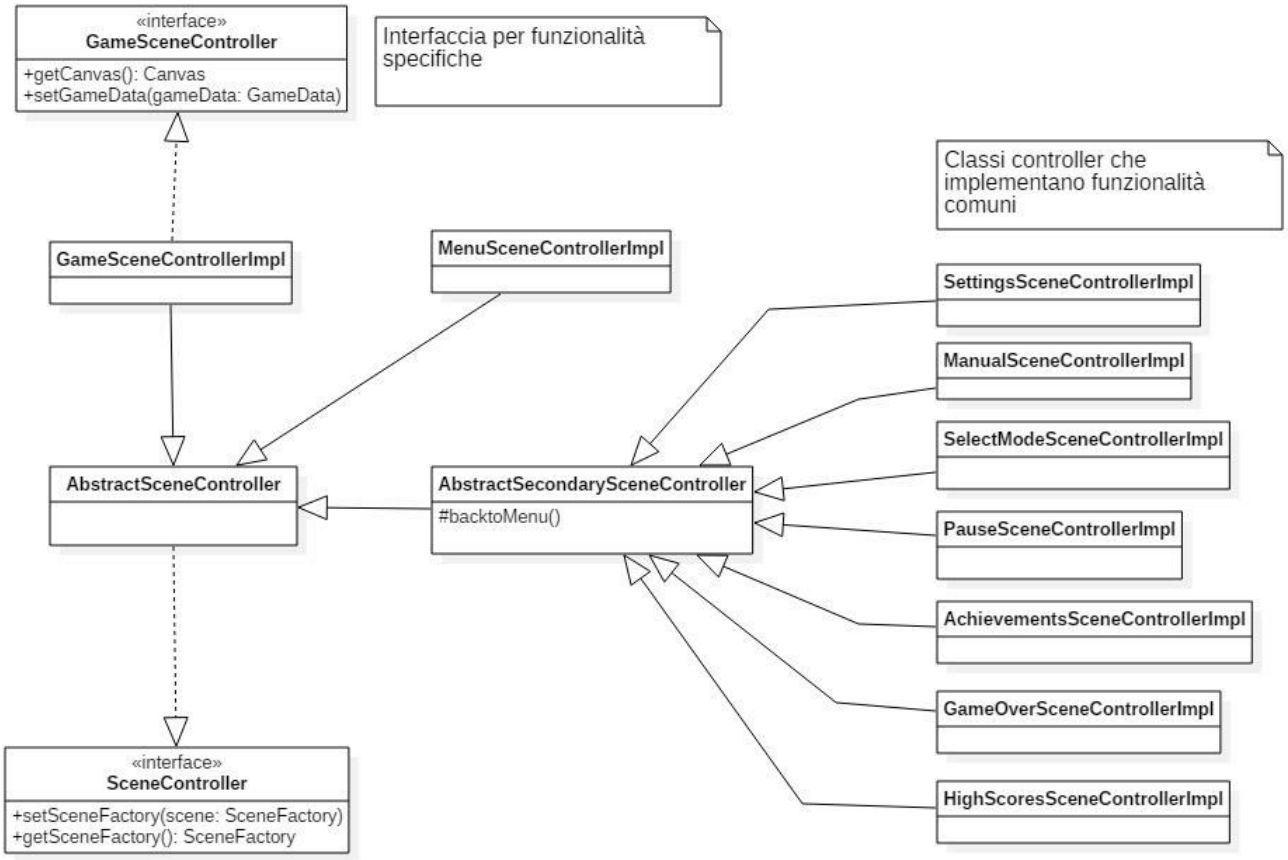
Ivan Mazzanti

Tra le parti della View di mia competenza, quella più considerevole è stata sicuramente la gestione delle varie **scene** dell'applicazione, ovvero definire una struttura logica che regolasse la creazione, la chiusura e la transizione tra le diverse schermate di gioco in funzione dell'input dell'utente e di specifici eventi durante una partita.
Per far fronte a queste esigenze, ho deciso di avvalermi di **JavaFX** con design della GUI descritto tramite il linguaggio **FXML**.
Questo infatti mi ha consentito di separare la parte prettamente grafica della GUI di ogni scena dalla logica che ne definisce il comportamento, garantendomi una completa indipendenza tra le funzionalità delle diverse schermate.
Per risolvere il problema concernente la creazione delle varie scene ho usato il pattern **Factory**, il quale mi ha permesso di scorporare la logica di creazione delle scene e ha reso possibile la facile introduzione di eventuali nuove schermate in caso di integrazioni future.

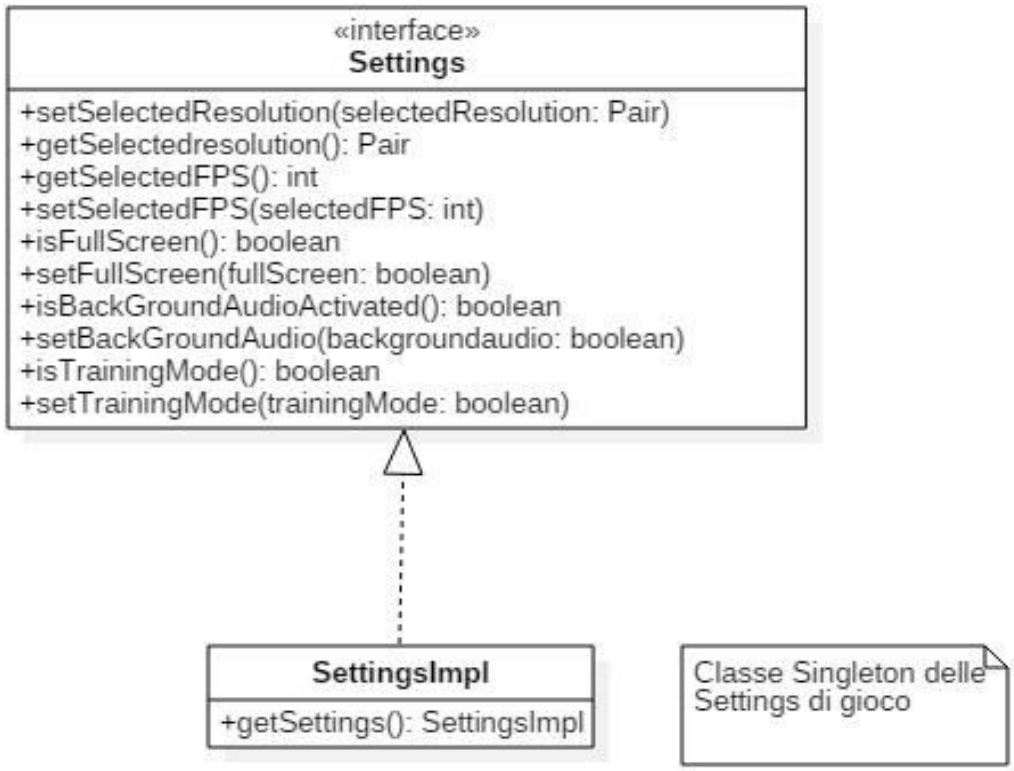


Nota: nello schema sono riportati solo i metodi inerenti all'illustrazione del pattern.

Infine, se una scena avesse necessità di mostrare in output dei dati relativi al gioco o di realizzare una funzionalità in comune con altre scene, le sarà sufficiente implementare rispettivamente una specifica interfaccia e/o una classe astratta.

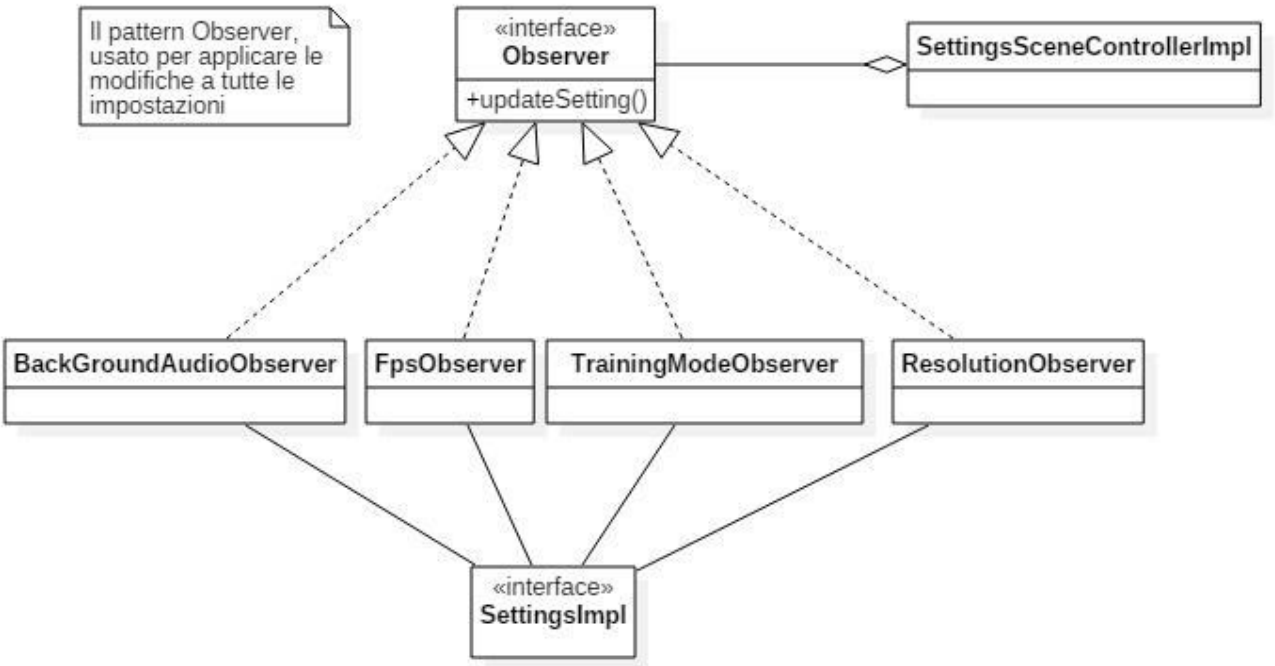


Un’altra parte importante di mia responsabilità era la realizzazione delle impostazioni (“**settings**”) del gioco. In questa sezione, ho ritenuto essenziale che venissero implementate le principali funzionalità che consentono all’utente di regolare le caratteristiche del gioco a proprio piacimento.
La classe contenente tutte le impostazioni di gioco è stata creata usando il pattern **Singleton**, in quanto avere più istanze di questa classe potrebbe portare a farne un uso che comprometterebbe il corretto funzionamento dell’applicazione. Inoltre, questo mi ha permesso di rendere la singola istanza di questa classe facilmente disponibile ove fosse richiesta.

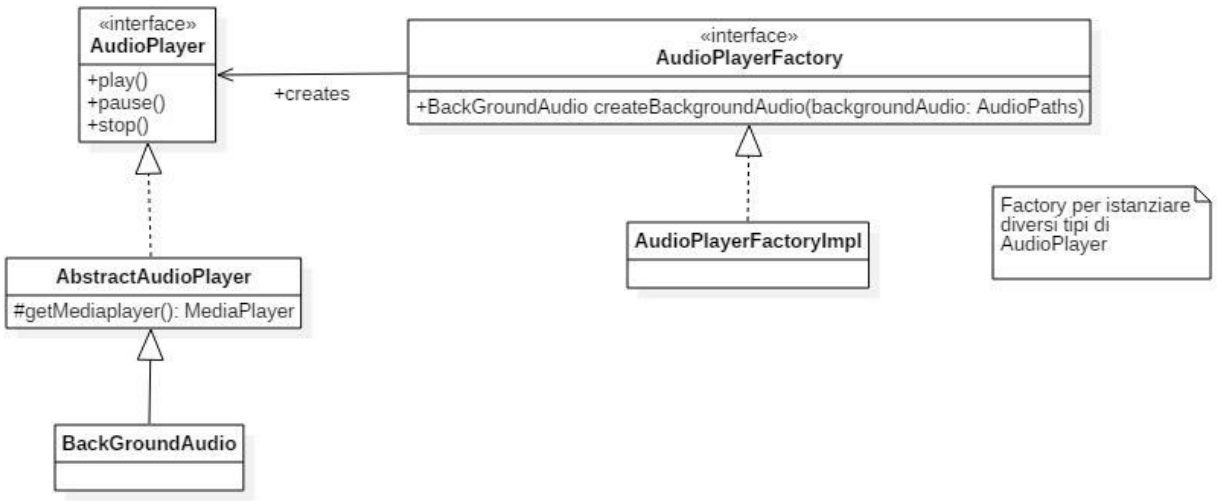


Nota: per motivi di spazio, ho riportato solo i metodi fondamentali per la gestione delle impostazioni.

Per gestire invece le modifiche alle impostazioni, mi sono avvalso del pattern **Observer**. Ho scelto questo pattern in quanto ogni volta che l’utente decide di applicare le modifiche, tutte le impostazioni devono essere aggiornate. Inoltre, questo tipo di organizzazione mi ha consentito di separare la gestione di ogni singola impostazione di gioco (insieme al componente grafico che la rappresenta) in una classe apposita, rendendone più semplice la gestione e la manutenzione.



L'ultima parte di cui mi sono occupato è stata la gestione dell'audio di gioco. Per rispondere a questa necessità, mi sono servito della classe **MediaPlayer** di JavaFX, molto utile e comoda per gestire file multimediali di diversi formati.
Ho implementato il pattern **Factory** per semplificare e scorporare la logica di creazione dei diversi tipi di audio che potrebbero essere necessari durante una partita (audio di background, effetti audio, suoni "semplici", ecc.) dalla loro gestione. Ho previsto inoltre una classe astratta **AbstractAudioPlayer** che fattorizza i metodi principali che ogni AudioPlayer dovrà avere. Eventualmente, sarà ogni sottoclasse a fare override di questi metodi in caso avessero la necessità di implementarli diversamente.

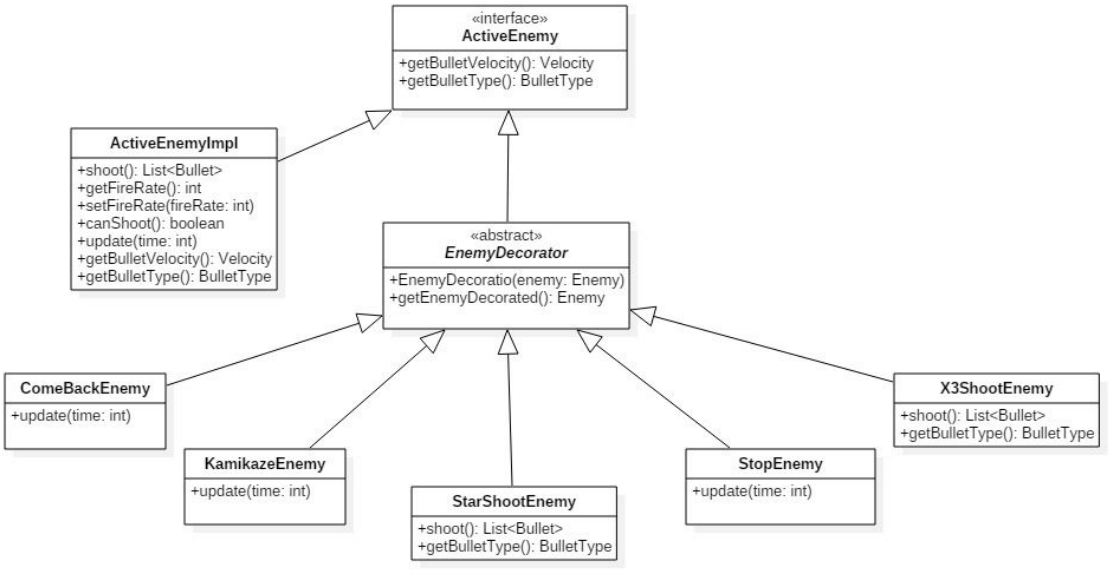


Massimo Donini

Nel progetto la parte principale a me dedicata è stata la gestione dei nemici e di ogni loro logica.

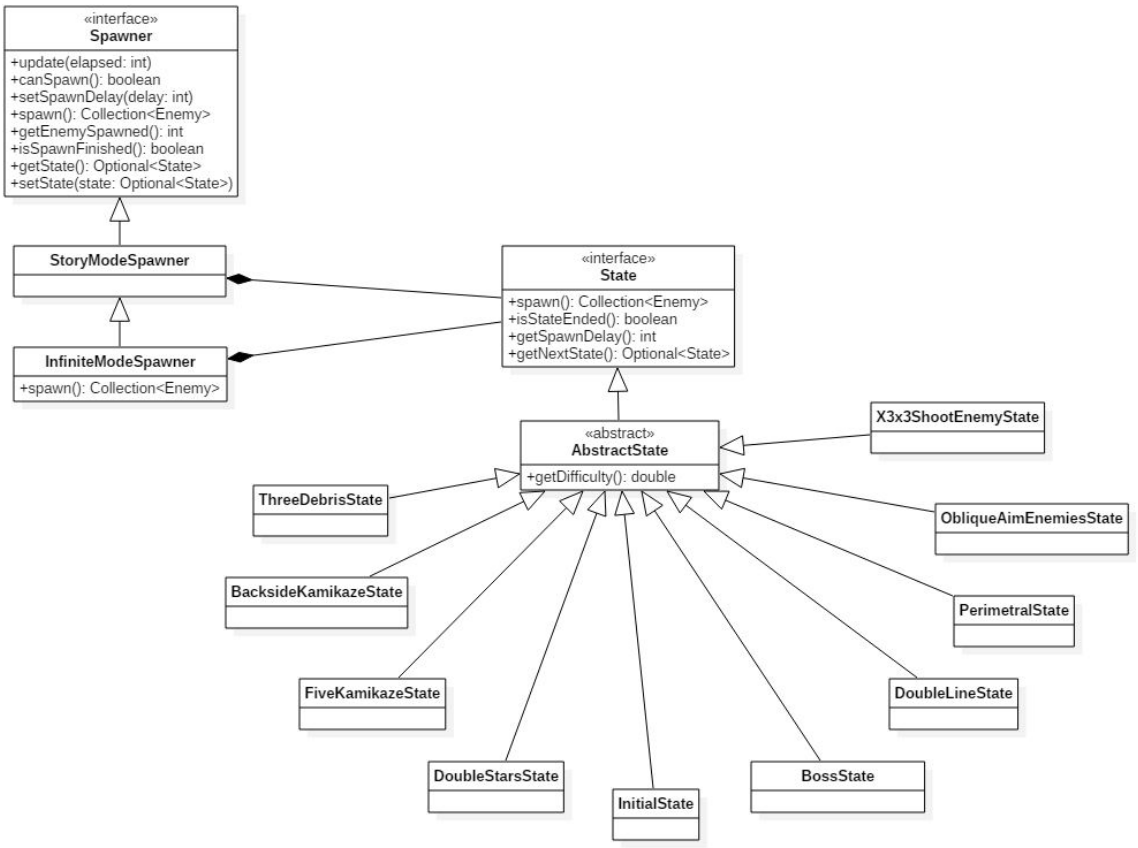
I nemici si possono dividere in due grandi categorie che ho organizzato gerarchicamente: gli attivi, che sparano e i passivi. Essendo il nostro un gioco a scorrimento non ho dovuto gestire il problema di distinguere i nemici tra fermi e in movimento.

Più in generale i nemici possono avere una certa caratteristica che li contraddistingue (come sparare 3 proiettili invece che 1, cercare di schiantarsi contro la navicella del giocatore, etc) o più di queste. Per evitare di creare una specifica classe per ogni combinazione possibile di queste caratteristiche ho utilizzato il pattern **Decorator** che è quindi una soluzione molto più propensa ad ampliamenti futuri e che evita di duplicare il codice.



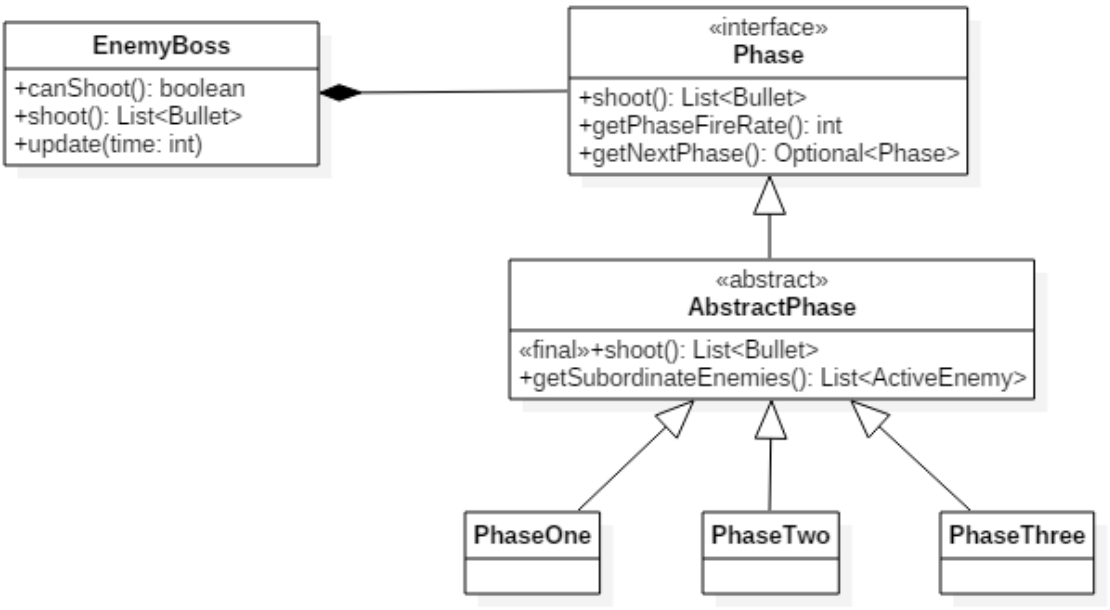
Per motivi di spazio non ho riportato tutti i metodi ma solo quelli che ho ritenuto più importanti per questo pattern.

Per rendere il gioco più interessante, oltre ad avere un'onesta varietà di nemici, mi sono anche posto il problema di come generarli. Fin da subito ho scartato l'idea di stabilire i valori iniziali dei nemici (come posizione, velocità, vita, caratteristiche) in modo pseudo-casuale ma evitando in ogni caso di scrivere un codice che mancasse di robustezza e flessibilità e pieno di condizioni logiche if-else. Ho usato quindi il pattern **State** che mi ha consentito di sviluppare diversi schemi di gioco da me ideati, collegandoli in modo semplice ma dividendone completamente le logiche.

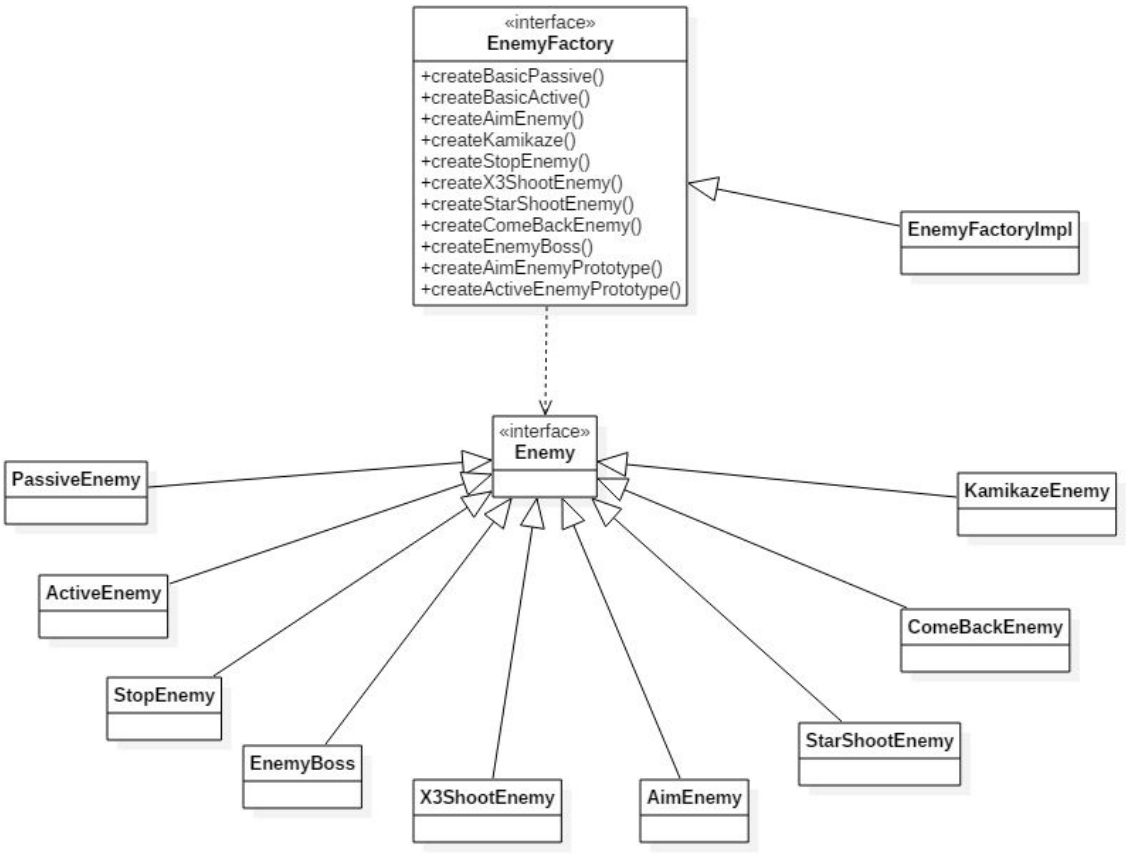


I diversi state sono collegati tra di loro prendendo spunto dal pattern **Chain of Responsibility**, in questo modo ho tolto allo spawner la logica di come alternare tra loro gli state.

Ho utilizzato il pattern **State** in modo "puro" per gli stessi motivi anche per gestire le diverse fasi del boss finale della modalità storia.

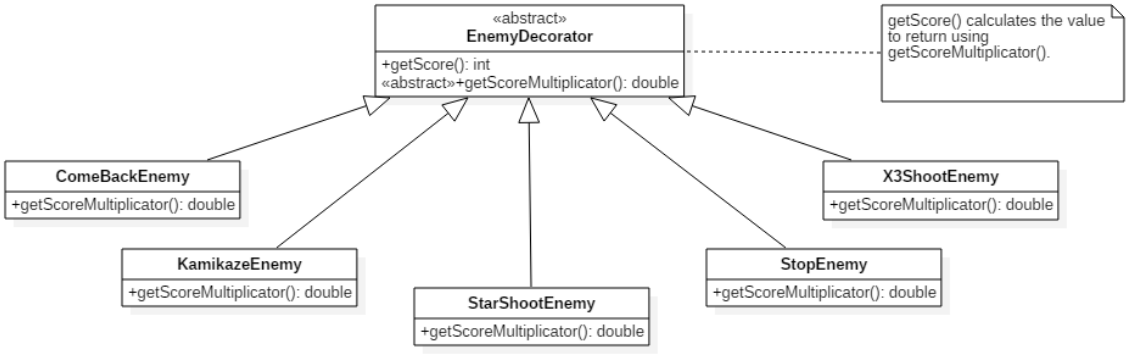


Dovendo creare numerosi tipi diversi di nemici mi è stato utile usare il pattern **Factory**, infatti, sono riuscito così a delegare la loro istanziazione ad una sola classe. Così facendo ho reso il codice più robusto ad eventuali cambiamenti dei costruttori e facile da estendere.



Ho tralasciato i molteplici parametri dei metodi e i valori di ritorno perché non li ho ritenuti utili alla comprensione dello schema del pattern.

Per evitare di duplicare del codice in ogni decorazione dei nemici ho scelto di usare il pattern **Template Method** per risolvere il problema di ottenere punteggi diversi quando si sconfiggono nemici con diverse complessità.



Capitolo 3
Sviluppo

3.1 Testing automatizzato

Durante la fase di sviluppo del nostro progetto abbiamo deciso di avvalerci della libreria JUnit per testare in modo automatizzato alcune funzionalità di base. Altre invece, data la loro impossibilità ad essere testate automaticamente, sono state verificate manualmente.
Esponiamo ora le componenti del progetto testate automaticamente.

Model

- Navicella giocatore: test delle diverse funzionalità della navicella quali muoversi, sparare e la corretta interazione con le altre entità.
- Dati di gioco: test del conteggio dei dati di gioco e test della versione Unmodifiable per la sola lettura
- Dati generali: test del calcolo dei dati globali con verifica del corretto avanzamento nelle sfide e del corretto salvataggio dei punteggi migliori.
- Collisioni: testano le collisioni tra le diverse forme geometriche che caratterizzano le entità.
- Decorazioni: test di integrità che controlla che non si possano avere due decorazioni contrastanti su uno stesso nemico (come kamikaze e come back, x3 shoot e star shoot).
- Nemici: vengono testate le caratteristiche tipiche dell'entità.

Tra le funzionalità che sono invece state testate manualmente abbiamo:

- Input: l'effettiva risposta dell'applicazione ai vari input non presenta bug.
- Generazione nemici: i nemici vengono sempre generati all'esterno del campo di gioco e fatti poi traslare al suo interno.
- Ridimensionamento: la finestra di gioco e le risoluzioni disponibili si adattano alle caratteristiche del display e rimangono proporzionate ad esso.

- Audio: corretta attivazione/disattivazione dell'audio a seguito di input dell'utente nel menù delle opzioni e durante la partita.

3.2 Metodologia di lavoro

A seguito della fase di sviluppo, possiamo asserire che la ripartizione dei compiti da noi concordata e confermata nelle fasi precedenti è stata rispettata, risultando piuttosto equa e ben bilanciata. Infatti non sono state necessarie sostanziali modifiche, all'evidenza delle minime dipendenze fra le varie componenti.
Di seguito elenchiamo le funzionalità implementate dai singoli elementi del team:

- Conti Davide: gestione navicella, input giocatore.
- Donini Massimo: generazione e gestione nemici e pattern di gioco, gestione adattabilità allo schermo.
- Lavista Andrea:potenziamenti, dati di gioco della partita e globali.
- Mazzanti Ivan: menu, impostazioni, collisioni, audio, gestione schermate grafiche.

Altre parti sono state invece sviluppate da più componenti del team. Le classi ViewImpl.java, la quale coordina gli aspetti grafici del gioco, e ModelImpl.java, che gestisce parti comuni di logica implementativa, sono tra queste, dato che ogni membro ha contribuito alla loro realizzazione.

Per la messa a punto dei thread, a cui si delegano l'aggiornamento del Model e della View, hanno collaborato Davide Conti, Andrea Lavista e Ivan Mazzanti.

Per gli aspetti comuni tra le entità del modello implementativo (navicella, nemici, potenziamenti, proiettili) sono state create molteplici interfacce di base e classi (astratte e di utilità) a cui hanno cooperato Davide Conti, Massimo Donini e Andrea Lavista.

La gestione delle varie finestre di gioco è stata presa a cura da Ivan Mazzanti mentre Massimo Donini ha contribuito all'adattabilità di queste a schermi diversi.

Nella fase di sviluppo ci siamo avvalsi del DVCS Git. La maggior parte del nostro codice è stato sviluppato sul branch "develop" ma, a fronte di funzionalità o modifiche più delicate che avrebbero potuto compromettere l'integrità del codice, sono stati aperti nuovi branch dedicati.

3.3 Note di sviluppo

Prima della realizzazione del progetto abbiamo avuto modo di assistere all'evento organizzato da S.P.R.I.Te, "Game as a Lab", tenuto dal prof. Ricci in cui egli ci ha mostrato un semplice videogioco, col relativo codice. Da questo abbiamo attinto per comprendere il funzionamento del Game Loop e la gestione dei movimenti degli oggetti di gioco.

DavideConti

Stream: Utilizzati in SpaceshipTest per semplificare e velocizzare alcune operazioni.

Lambda expression: Dovunque fosse possibile ho utilizzato le lambda per diminuire le dimensioni del codice.

Optional: Utilizzati in alcuni campi privati e metodi che potevano ritornare null, in quanto non sempre veniva ritornato un oggetto.

Libreria Common: Utilizzata per la classe Pair e per i metodi equals e hashCode.

Per l'utilizzo di alcuni pattern che ho utilizzato in fase di design dettagliato ho consultato [1].

Andrea Lavista

Lambda: ho impiegato spesso le lambda in diverse delle classi che ho sviluppato, in particolar modo nelle classi attinenti ai dati di gioco della partita e globali, con l'obiettivo di avere codice più compatto e più leggibile.

Stream: adoperati principalmente nelle classi inerenti ai dati globali, per effettuare alcune operazioni (filtraggio, ordinamento, somme, etc.) su delle collezioni.

Optional: utilizzati in quei contesti in cui le richieste avrebbero potuto non ricevere un oggetto, tra cui la lettura da file (in quanto potrebbe non esserci alcun file da leggere, per esempio al primo avvio) e la richiesta del nome al giocatore in caso di punteggio migliore.

Reflection: per il caricamento dell'enumerazione EntityType e delle immagini che contiene. Queste risorse vengono caricate appena viene lanciata l'applicazione, per evitare rallentamenti durante lo svolgimento della partita.

JavaFX: per la gestione delle immagini relative alle entità del gioco.

Ivan Mazzanti

Lambda expression: impiegate per ottenere un codice più compatto, chiaro e di maggior qualità. Usate principalmente per implementare Handler di eventi, filtri degli Stream e scorrere gli elementi delle liste.
Stream: usati principalmente nella gestione delle collisioni, per notificare gli Observer e in molte operazioni che coinvolgessero le Collezioni. Sono uno strumento veramente potente per praticità e leggibilità del codice.
Optional:li ho usati in maniera limitata per gestire il comportamento dei file audio e per evitare di lavorare con dei valori "**null**" dopo aver letto il contenuto di un file.
Reflection: Ho usato meccanismi di reflection per caricare risorse utili al funzionamento dell'applicazione, come i file FXML, il CSS e file audio.
Wildcards: utilizzate unicamente per gestire la letture da file delle impostazioni salvate.
Libreria Common: Ho utilizzato questa libreria per usufruire della sua implementazione della classe Pair.
JavaFX: libreria utilizzata per implementare praticamente tutta la parte relativa alla GUI del progetto. Inoltre, questa libreria è stata usata sia per intercettare le collisioni di gioco che per riprodurre e gestire file audio.
FXML: è stato davvero utile per descrivere il design della GUI e separarlo dalla sua logica implementativa. Mi sono avvalso del programma Scene Builder per creare file FXML in modo più rapido ed efficace.
CSS: ho utilizzato questo linguaggio per rendere le GUI molto più apprezzabili e decorate.

Per velocizzare l'apprendimento delle numerose funzionalità di JavaFX e per sfruttare appieno la loro efficienza, ho fatto riferimento a tutto il materiale a mia disposizione su questa libreria, come le slide del corso, progetti ben valutati degli anni passati e tutorial in rete. Ci tengo però a precisare che l'implementazione è unicamente frutto di mie progettazioni, riflessioni e considerazioni basate unicamente sulla nostra applicazione e che alle volte è capitato di dover rivisitare per adeguarsi ad esigenze nate nel corso dello sviluppo del progetto.

Segue un elenco dei progetti consultati:

- **Like a bullet** (dal quale sono anche venuto a conoscenza del meccanismo di "scale" della grafica e di un efficace modo in cui usarlo).
- **Space Impact**.

I principali riferimenti di JavaFX consultati in rete sono:

- https://docs.oracle.com/javase/8/javafx/api/toc.htm
- https://stackoverflow.com/questions/15013913/checking-collision-of-shapes-with-javafx?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa
- https://stackoverflow.com/questions/23202272/how-to-play-sounds-with-javafx

L'unica parte in cui ho copiato dei pezzi di codice è stata nella stesura del file CSS in quanto ne sono molto poco esperto. Lascio qui il principale riferimento per la grafica dei bottoni e delle immagini di sfondo:

- http://fxexperience.com/2011/12/styling-fx-buttons-with-css/
- https://www.youtube.com/watch?v=urLNNAnQS4Y

Massimo Donini

Stream: utilizzati in sostituzione dei cicli for nei nemici e nello spawner e per disegnare tutte le entità in gioco in ViewImpl.

Optional: utilizzati insieme al pattern State nei metodi che restituiscono lo stato successivo, se presente.

Ho inoltre sviluppato un paio di algoritmi che ritengo interessanti:

Il primo si trova nella classe StaticVelocity e avendo in ingresso la velocità di una certa entità e un angolo permette di calcolare una nuova velocità con la direzione sposta dell'angolo specificato. È stato usato quando ci è servito sparare proiettili con un certo angolo uno dall' altro (nemici che sparano tre proiettili, che sparano a stella e nella navicella).

Il secondo, invece, ha risolto il problema delle risoluzioni del gioco. Queste dipendono completamente dallo schermo in cui si gioca, sono sempre proporzionate ad esso ma lasciano invariate le proporzioni delle componenti al loro interno. Per ottenere ciò è stato necessario applicare un moltiplicatore di dimensione alla

finestra di gioco che fosse uguale per entrambi gli assi e poi aggiungere un eventuale bordo in orizzontale o verticale in base al monitor.

Una volta capito cosa volevo ottenere e in che modo ottenerlo sviluppare il tutto è stato abbastanza facile grazie a come le varie finestre di gioco erano state implementate da Ivan Mazzanti.

Per lo studio di pattern non affrontati a lezione ho fatto affidamento al sito www.journaldev.com

Davide Conti, Massimo Donini, Andrea Lavista

JavaFX: libreria utilizzata per rappresentare e manipolare le figure delle entità di gioco all'interno del Model (navicella, nemici, proiettili, potenziamenti).

Capitolo 4
Commenti finali

4.1 Autovalutazione e lavori futuri

Davide Conti

Personalmente, mi ritengo soddisfatto del risultato ottenuto.

La parte che mi ha attratto con maggior interesse è stata la fase iniziale, di analisi e di progettazione, in quanto mi ha dato modo di capire con maggior profondità quanto sia importante definire il dominio e progettarlo nel miglior modo possibile onde evitare di perdere un tempo indefinito nella fase di implementazione.

Ho migliorato la mia capacità di programmazione, principalmente grazie ai design patterns dei quali, prima di questo corso, ne ignoravo l'esistenza, e in generale di tutto ciò che concerne un codice di più alta qualità.

A causa del poco tempo a disposizione non credo di voler portare avanti il progetto. Sicuramente è stata un'ottima esperienza che mi sarà d'aiuto per progetti futuri.

Andrea Lavista

In generale sono piuttosto soddisfatto del lavoro svolto. Durante la fase di sviluppo le mie priorità sono state quelle di produrre codice duttile e leggibile facilitando eventuali modifiche future. In questo senso ritengo che ciò da me prodotto, seppur certamente migliorabile in alcuni punti, generalmente rispetti questi canoni. D'altra parte devo riconoscere di aver impiegato più tempo del necessario nell'implementazione a causa di errori e manchevolezze verificatesi nelle fasi antecedenti.

Con gli altri componenti del gruppo c'è stata collaborazione ed un continuo scambio di informazioni sulle difficoltà incontrate. Nella fattispecie ho dato dei suggerimenti su alcuni aspetti concernenti parti di loro competenza (gestione delle scene e dell'input). Inoltre, spinto dalla curiosità, mi sono interessato anche del lavoro altrui; ciò, unito al fatto di aver dovuto realizzare componenti che spaziassero da Model a View, mi ha consentito di approfondire la conoscenza di buona parte del sistema software, seguendone le evoluzioni. Contestualmente questa visione ad ampio raggio mi è tornata molto utile, consentendomi di poter prevedere ed intercettare

eventuali criticità, assicurarmi della coerenza delle singole parti prodotte e valutare la fattibilità e l'impatto di potenziali modifiche, facilitando altresì la successiva fase di integrazione.

Non credo che questo progetto avrà un futuro, ma sicuramente mi è stato molto utile per comprendere meglio le dinamiche di lavoro in un team nonché l'importanza delle fasi di analisi e progettazione.

Ivan Mazzanti

Sono complessivamente soddisfatto del lavoro che ho prodotto. Ritengo che molte delle mie parti avrebbero potuto essere progettate, o quanto meno implementate, in modi migliori, tuttavia sono felice di esser riuscito a costruire tutta la GUI di un progetto dandole una struttura ben definita, usando una libreria che non conoscevo e avendo una limitata quantità di tempo a disposizione.

Credo che sia possibile aggiungere delle novità in questo progetto, ma marginali: ad esempio si potrebbero aggiungere nuove modalità di gioco, diversi tipi di nemici o effetti audio aggiuntivi. Per innovazioni più grandi credo sia opportuno documentarsi su librerie (sia grafiche che non) predisposte appositamente allo sviluppo dei videogiochi.

Massimo Donini

È stata la mia prima volta in cui ho partecipato ad un progetto di gruppo ed è stata un'esperienza che ho apprezzato molto. Non penso che questa applicazione avrà un futuro ma sicuramente mi sarà d'aiuto per quelle future. Sono soddisfatto sia di come ho lavorato, usando pattern che mi sono stati di grande aiuto e scrivendo un codice facile da estendere, sia dell'effettivo risultato finale del gioco, di come si presenta e del livello di difficoltà.

Ad aver avuto più tempo avrei aggiunto qualche nemico, stati per lo spawner e modalità di gioco in più ma non penso fosse questo l'obiettivo del progetto e che lo stato attuale sia già abbastanza buono.

Le parti che ho preferito sono state quelle dello sviluppo dei vari nemici e degli stati per lo spawner.

4.2 Difficoltà incontrate e commenti per i docenti

Ivan Mazzanti

Nonostante JavaFX sia una libreria davvero efficiente, pratica e utile, ho riscontrato alcune difficoltà durante il suo utilizzo, che mi hanno dato non poche complicazioni. La prima problematica riguarda la gestione dei numerosi elementi grafici di una schermata quando a questa viene applicato uno "scale" (ovvero un ridimensionamento) dei suoi componenti. È stato per me molto difficile trovare una disposizione degli elementi che mi garantisse il corretto mantenimento dell'allineamento di tutti i componenti grafici. Infatti, quando veniva applicato un scale abbastanza alto, i componenti perdevano completamente il loro allineamento mentre con uno scale basso, i nodi grafici si sovrapponevano tra loro ben prima di quanto mi aspettassi. Alla fine, credo di aver trovato una particolare disposizione dei componenti che permette di mantenere il più possibile una struttura definita e un corretto allineamento della grafica.
È stato anche problematico implementare il full screen delle scene: usando il metodo setFullScreen() di JavaFX durante il passaggio tra una schermata e l'altra si notava un rimpicciolimento della schermata prima che questa venisse reimpostata a schermo intero, un effetto brutto e fastidioso. Anche in questo caso ho dovuto aggirare il problema al meglio delle mie possibilità, usando soluzioni alternative.

Il secondo problema riguarda l'utilizzo della classe MediaPlayer, che ho usato per riprodurre i file audio. L'istanza creata infatti, sembrava non rispondere bene alle chiamate di stop e pausa, causando numerose volte un comportamento scorretto del file audio. Inoltre, quando ho provato a testare l'audio avviando l'applicazione da un file Jar, l'audio si fermava inspiegabilmente dopo che veniva messo in pausa e fatto riprendere mentre su Eclipse il funzionamento era corretto. Ho provato quindi ad utilizzare l'interfaccia **AudioClip**, ma questa dava problemi nell'esecuzione in loop di un file audio se questo era stato precedentemente messo in pausa, inoltre l'unico formato audio che supporta è il **wav** che è davvero troppo pesante. Ho tentato perciò di scaricare ed usare **JLayer**, una libreria open source creata appositamente per gli MP3 (http://www.javazoom.net/javalayer/javalayer.html), ma ho poi scoperto che questa non crea in automatico un thread per riprodurre il file audio. Quindi, dato che il tempo per completare il progetto stava ormai per finire, sono tornato ad usare la classe MediaPlayer, cercando di sistemare tutte le problematiche che trovavo, riuscendo alla fine ad ottenere, a mio parere, un buon risultato.

Appendice A
Guida utente

Appena avviata l'applicazione si aprirà il menù principale e qui l'utente potrà scegliere tra diverse opzioni:

- *manuale*: qui viene esposto un riassunto dei principali meccanismi di gioco.
- *settings*: in questa schermata è possibile regolare a proprio piacimento le impostazioni di gioco. Queste modifiche verranno salvate e saranno caricate ad ogni successivo avvio del gioco.
- *achievements*: sono mostrati gli obiettivi raggiunti durante la totalità delle partite con i relativi avanzamenti.
- *highscores*: sono elencati i punteggi migliori ottenuti nelle diverse partite.
- new game: qui è possibile iniziare una nuova sessione di gioco, sarà possibile scegliere fra due diverse modalità:

- *STORY MODE*: il giocatore dovrà sconfiggere ondate di nemici a difficoltà crescente ed infine un boss. La partita termina quando non ci saranno più nemici sullo schermo o quando la navicella esaurisce la vita a disposizione.
- *SURVIVAL MODE*: Il giocatore dovrà cercare di sopravvivere il più a lungo possibile affrontando ondate di nemici a difficoltà crescente. La partita termina quando la navicella esaurisce la vita a disposizione.

Una volta iniziata la partita l'utente potrà muovere la navicella con le frecce direzionali e sparare con il tasto "X". Per mettere in pausa il gioco è sufficiente premere il tasto "ESC".

Durante la partita sarà possibile raccogliere dei potenziamenti che verranno rilasciati in modo casuale dai nemici uccisi.
I potenziamenti sono:

- *ATTACCO*: di colore rosso, incrementa il danno dei proiettili della navicella.
- *VELOCITÀ D' ATTACCO*: di colore viola, aumenta la velocità d'attacco.
- *VITA*: di colore verde, incrementa la vita della navicella.
- *SCUDO*: di colore azzurro, dona alla navicella uno scudo che la rende immune ai danni per un certo periodo di tempo.

La navicella potrà aumentare le proprie caratteristiche fino ad un limite fissato. Dopo un certo numero di power up d'attacco raccolti, la navicella incrementerà il numero di proiettili che può sparare, bilanciando il danno. Tuttavia, se la navicella subisce danni in questa fase, il numero di proiettili sparati diminuirà.

A fine partita, verranno aggiornati gli achievements; qualora il giocatore avesse totalizzato un nuovo punteggio migliore, potrà salvarlo associandoci un nome. Nelle impostazioni, l'utente può attivare la "Training Mode", una modalità in cui la navicella non subisce alcun tipo di danno mentre l'utente può allenarsi e studiare i diversi comportamenti dei nemici. In questa modalità non è possibile né sbloccare achievements né salvare il punteggio a fine partita.

Bibliografia

[1] Design patterns di Gamma, Helm, Johnson, Vlissides.