Relazione

Stefano Vanucci, Emiliano Rattini, Lorenzo Rubboli, Nicola Milandri

April 2022

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il software realizzato mira alla riproduzione di un noto flash game con grafica 2D di nome Boxhead, lo scopo di gioco è quello di sopravvivere a ondate di zombie grazie all'utilizzo di armi da fuoco di potenza incrementale e alla raccolta di munizioni e salute.

Requisiti funzionali

- Menù principale che permette all'utente di informarsi sui comandi di gioco, gestire l'audio, iniziare una nuova partita, gestire la pausa di gioco, uscire dal gioco.
- Calcolo direzionale e sistema di sparo a 8 direzioni con visuale dall'alto, sempre ricalcando il titolo sopracitato
- L'uccisione a catena di nemici permetterà di ottenere armi e potenziamenti di fuoco rendendo il gioco più dinamico, compensando l'aumento del numero di nemici generati.

Requisiti non funzionali

- Il gioco dovrà essere funzionale e possedere una buona fluidità

1.2 Analisi e modello del dominio

Il giocatore dovrà sopravvivere ad una serie di orde di zombie evitando i muri disposti sul terreno di gioco. Sia l'utente che i nemici sono forniti di un quantitativo limitato di salute: il giocatore dovrà resistere usufruendo delle armi che sbloccherà durante il corso del gioco, mentre gli zombie andranno a caccia dell'utente causando danno a contatto. Le armi avranno a disposizione diversi potenziamenti per bilanciare l'incremento del numero di zombie con l'avanzare delle ondate, essi saranno ottenibili in base alla killstreak del giocatore.

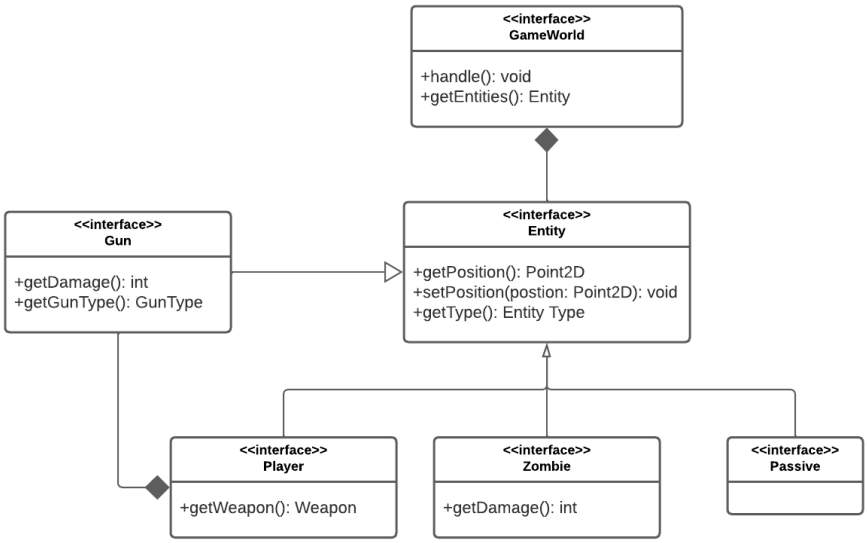

Figura 1.1: Schema UML delle principali entità

Capitolo 2

Design

2.1 Architettura

Il progetto è stato sviluppato seguendo il pattern **MVC** permettendo quindi una suddivisione semplificata ed efficace del carico di lavoro e dello sviluppo. La gestione di tale pattern avviene principalmente nella classe **GameLevel** dove vengono inizializzati i controllers delle varie entità agenti nel gioco. Tali controllers sono fondamentali per la coordinazione di model e view. Per quanto riguarda il mondo e l'input sono state implementate diverse classi atte a gestire le diverse renderizzazioni e l'aggiornamento da tastiera riportando i cambiamenti sui diversi **model** e **view**. La struttura del codice così gestita permette un forte riuso e previene anche errori a cascata semplificando la comunicazione tra le diverse parti del codice e gli attori coinvolti.

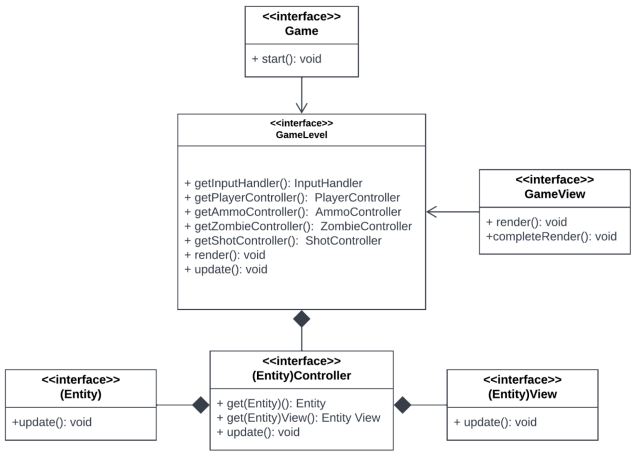

Figura 2.1: Schema UML dell'architettura.

2.2 Design dettagliato

Stefano Vanucci

La mia parte di progetto consisteva nello sviluppo generico delle entità, per poi concentrare l'attenzione sul giocatore, dalle sue caratteristiche al suo comportamento, gestendo anche gli input da tastiera. Inoltre, visto il carico di lavoro non eccessivo, ho sviluppato la parte sonora e la gestione degli sprite. In fase di progettazione abbiamo subito notato come le entità avessero diverse caratteristiche in comune come posizione e velocità. Per tale ragione ho deciso di sfruttare le classi astratte andando a creare AbstractEntity che va a implementare i metodi che riguardano la posizione, caratteristica condivisa sia dalle entità passive che attive. In merito a nemici, giocatore e proiettili, ovvero le entità attive, ci è sembrato opportuno continuare con la strategia delle classi astratte creando così in primis AbstractActiveEntity che aggiunge il movimento e AbstractHealthEntity che si concentra sulle entità dotate di salute.

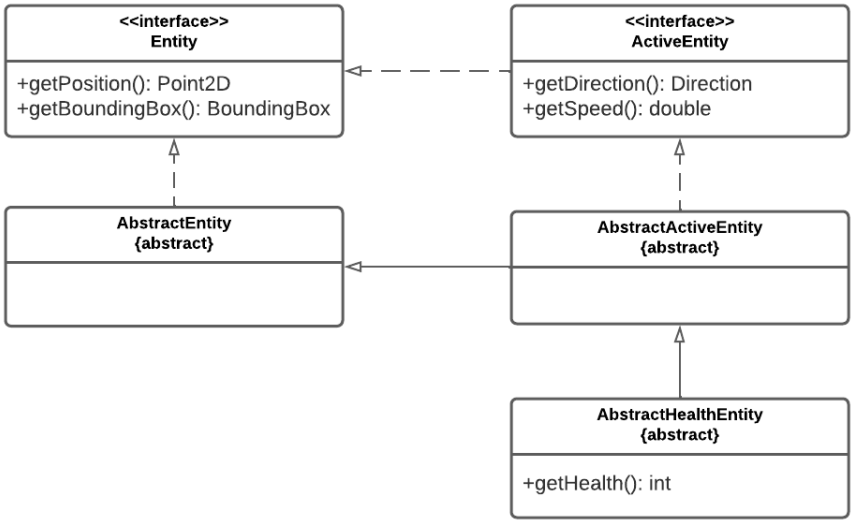

Figura 2.2: Schema UML delle entità

Per il player ho realizzato tre classi principali seguendo il pattern MVC. In buoma parte il model era già implementato ereditando AbstractHealthEntity ed è quindi stato necessario creare una classe Player aggiungendo alcuni metodi riguardanti solo il giocatore, come la gestione delle armi e delle collisioni. La parte di view invece è gestita in PlayerView e si limita a costruire ed aggiornare l'immagine del giocatore. Infine il PlayerController coordina queste due classi per controllare, in base all'input, movimento e azioni del giocatore.

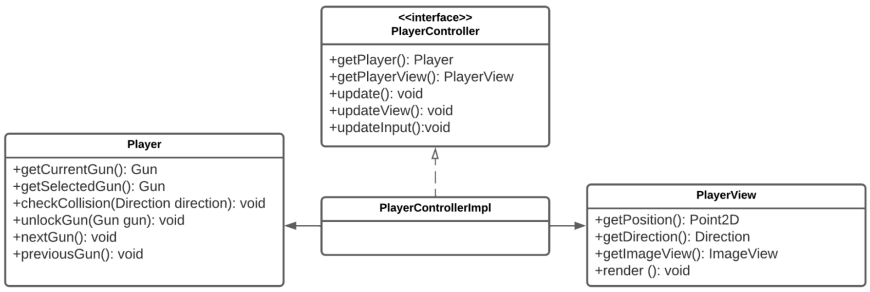

Figura 2.3: Schema UML dell'architettura del Player

Tale input viene gestito grazie ad un InputHandler, classe di supporto necessaria che si occupa di "catturare" e registrare gli eventi da tastiera. In questo modo è possibile tradurre gli input in azioni all'interno di una seconda classe chiamata PlayerInput. Essendo necessario un gran numero di sprite, ho deciso di utilizzare per la loro realizzazione il pattern Factory, creando così una SpriteFactory che genera diversi oggetti Sprite in base al tipo di entità passata. Inoltre Sprite si occupa anche dell'aggiornamento delle immagini degli zombie e del giocatore che cambiano in base alla loro direzione, nel caso di quest'ultimo l'immagine cambia anche a seconda dell'arma equipaggiata.

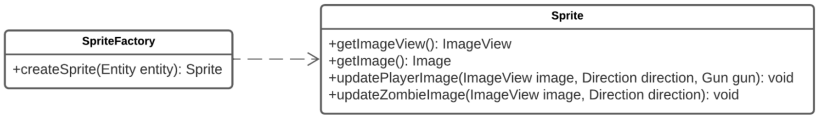

Figura 2.4: Pattern factory usato per la creazione degli sprite

Emiliano Rattini

La parte che mi spettava comprendeva gestione delle armi e dei proiettili. A questi si sono aggiunti anche la gestione del punteggio, del cambio di scene e quindi anche della lettura da file. Ogni arma implementa l'interfaccia Gun che a sua volta estende l'interfaccia Entity, e sono associate ad un Player. Ci sono 3 tipi di Gun, che non vengono rappresentate da classi separate, ma vengono catalogate in una enum dentro Gun e vengono create attraverso la GunFactory (vedi dopo)

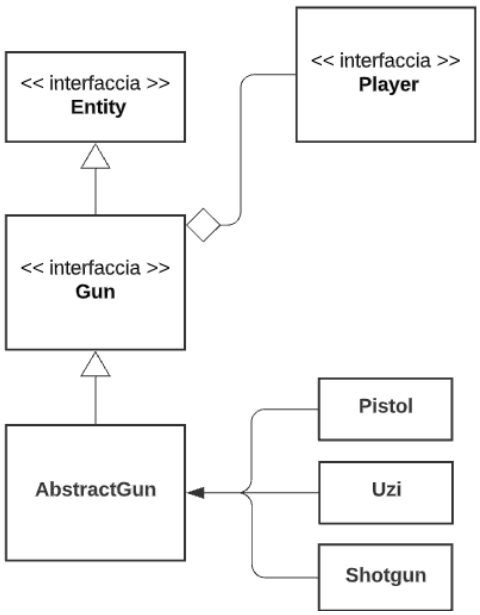

Figura 2.5: Schema UML con le interfacce per la rappresentazione delle armi.

I vari tipi di arma avranno danni, rateo di fuoco e quantità di munizioni differenti e per comodità si è deciso di utilizzare il pattern Builder, che facilita la creazione di armi diverse.

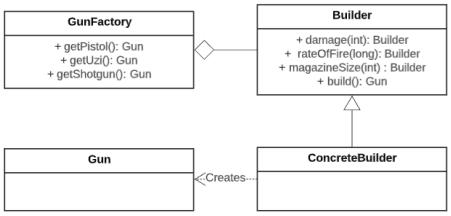

Figura 2.6: Schema UML del pattern Builder.

Quindi, per la generazione delle armi si è utilizzato il pattern Simple Factory, implementato attraverso la classe GunFactory. In questa classe abbiamo un metodo specifico per la creazione di ciascuna arma, che viene istanziata con valori specifici.

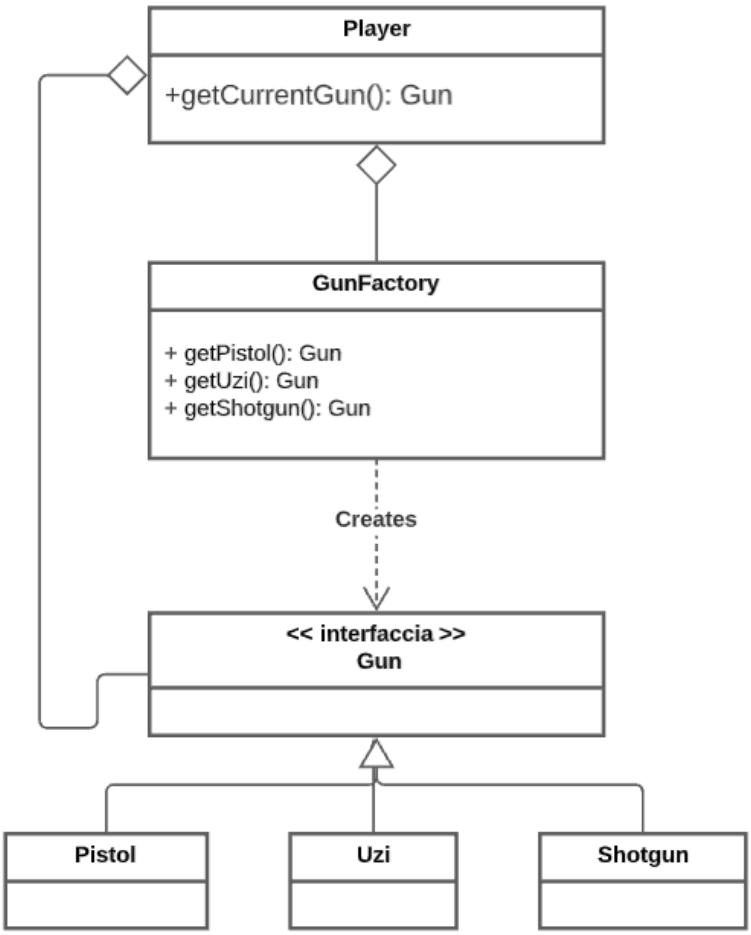

Figura 2.7: Schema UML del pattern Simple Factory per la creazione di armi.

L'azione principale delle armi è quella di sparare, quindi ogni arma può generare dei Bullet. Nel caso specifico Pistol e Uzi generano un Bullet alla volta mentre lo Shotgun ne genera 3. Nell'ottica iniziale si era pensato di avere più tipi di "proiettili" tra i quali Razzi e Granate, ma nel pratico non è stato possibile realizzarli per questioni di tempo. Però si è mantenuto lo schema progettuale iniziale che fa uso dei pattern Template Method e Strategy. L'interfaccia base è Shot da cui poi si ottiene la classe astratta AbstractShot che conterrà tutto il codice necessario per un colpo (per esempio il metodo getDamage() o il metodo getTrajectory()). La gestione del movimento viene però delegata a una Trajectory che diventa di fatto la strategia del pattern. Per questioni di tempo l'unica implementazione è stata StraightTrajectory, quella usata dai Bullett. Rimane comunque possibile sviluppare nuovi tipi di Shot implementando nuove Trajectory (es. ParabolicTrajectory per una Granade).

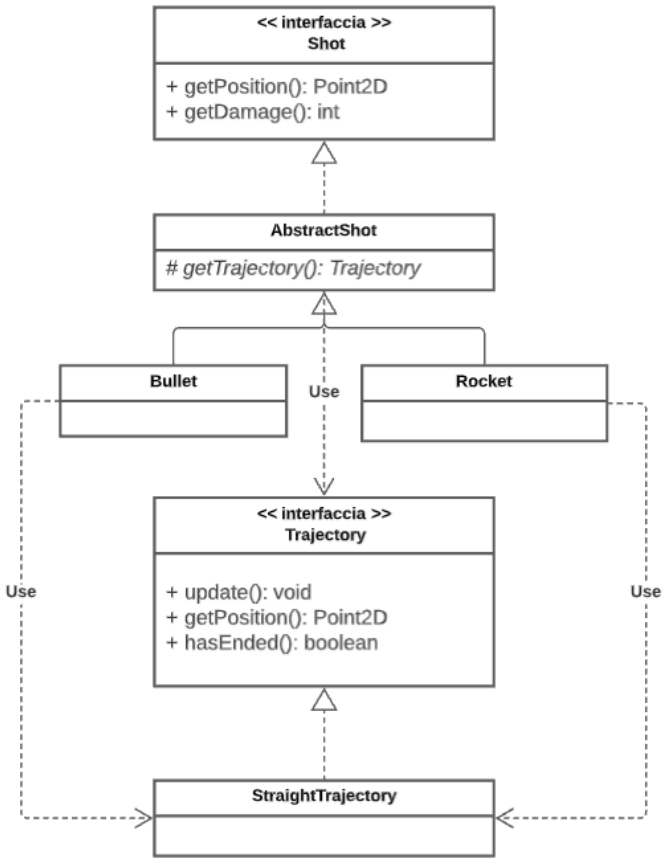

Figura 2.8: Schema UML dei pattern Template Method e Strategy per la gestione dei proiettili.

Dopo la modellazione dei Bullet si è passati alla modellazione dello ShotManager che deve gestire i vari Shot presenti nella mappa. In particolare deve gestire le collisioni degli Shot e fare in modo che diventino ended quando collidono con qualcosa, infliggendo danni se uno zombie.

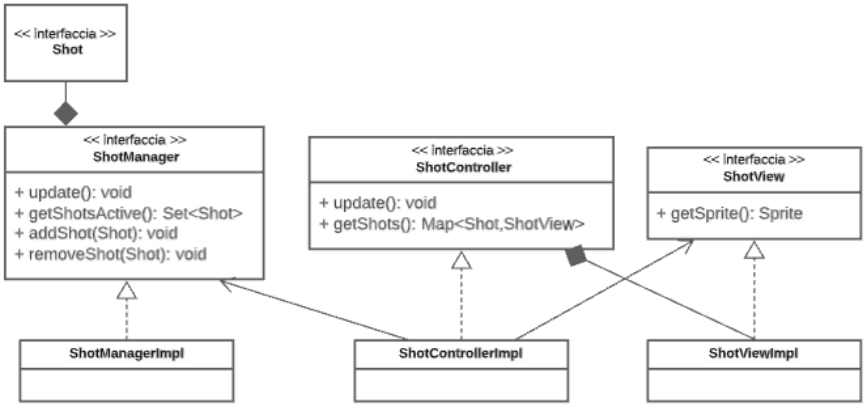

Figura 2.9: Schema UML del pattern MVC per la gestione dei proiettili all'interno della mappa.

Per quanto riguarda la view si è pensato di usare un'interfaccia ShotView e la sua implementazione BulletView. All'interno di essa troviamo uno Sprite e un metodo per settare la direzione dell'immagine. Per le armi non c'è stato bisogno di implementare una view poichè agganciate al Player, e quindi gestite dalla view del Player. In gioco gli upgrade e gli sblocchi di nuove armi vengono gestiti in base ad una streak, che sale ogni volta che si uccide uno zombie e scende in base ad un cooldown decrescente (più sale la streak, minore sarà il tempo per farla scendere). Si è modellato tutto attraverso l'interfaccia Score e la classe GunUpgradeManager, in cui sono salvati tutti i valori di streak per i quali si sblocca qualcosa di nuovo. Dentro Score viene memorizzato il tempo di gioco, gli zombie uccisi e viene gestito il contatore della streak. Si è scelto di tenere lo Score all'interno dello ZombieModel per fare in modo che ogni volta che viene effettivamente eliminato uno zombie la streak viene incrementata.

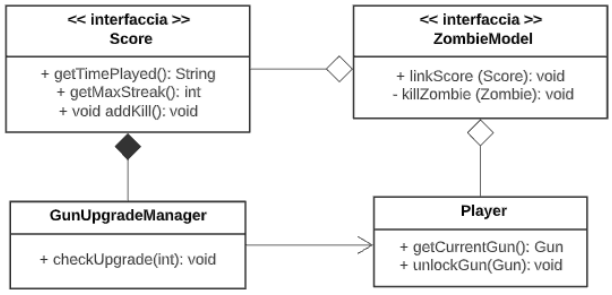

Figura 2.10: Schema UML della gestione della streak e dello Score

Infine ho sviluppato il SceneSwapper che facilita il cambio di Scene all'interno del gioco, tenendole memorizzate in una Map con nome associato a scena. Per implementare la GameView, usata durante il gioco, abbiamo utilizzato FXML, ed è quindi stato necessario implementare un FXMLContainer che permettesse la lettura ed il caricamento di una scena da file.

Lorenzo Rubboli

Gestione degli zombie e gestione del round flow sono stati i miei principali compiti. Mi sono inoltre cimentato nella realizzazione della MenuView e dell'implementazione del GameLevelImpl come aggregatore dei vari controllers. La classe Zombie estende AbstractHealthEntity in cui sono contenuti i metodi di gestione delle entità attive quali player e zombie e la gestione della loro vita. Non è stato implementato alcun Builder in quanto si è deciso di non avere più Zombie type.

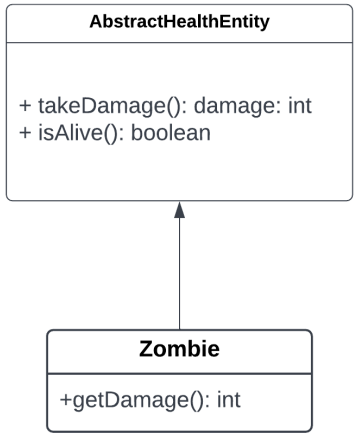

Figura 2.11: Schema UML della gestione della vita degli zombie.

L'intelligenza artificiale degli zombie ovvero il modo in cui computano la posizione del player, in cui si muovono, in cui controllano le collisioni con muri e giocatore è stata creata con l'ausilio del pattern Strategy usando l'interfaccia ZombieAI e la sua implementazione ZobieAIImpl che contiene tutte le "strategie" di comportamento degli zombie. Il model degli zombie è stato implementato nelle classi ZombieModel e ZombieModelImpl in cui si gestisce movimento, eliminazione, score linking e spawn. Quest'ultimo è stato realizzato attraverso Spawn e SpawnImpl che gestisce numero e posizione degli spawnPoints con relative informazioni sugli Zombie compresa la loro velocità di movimento.

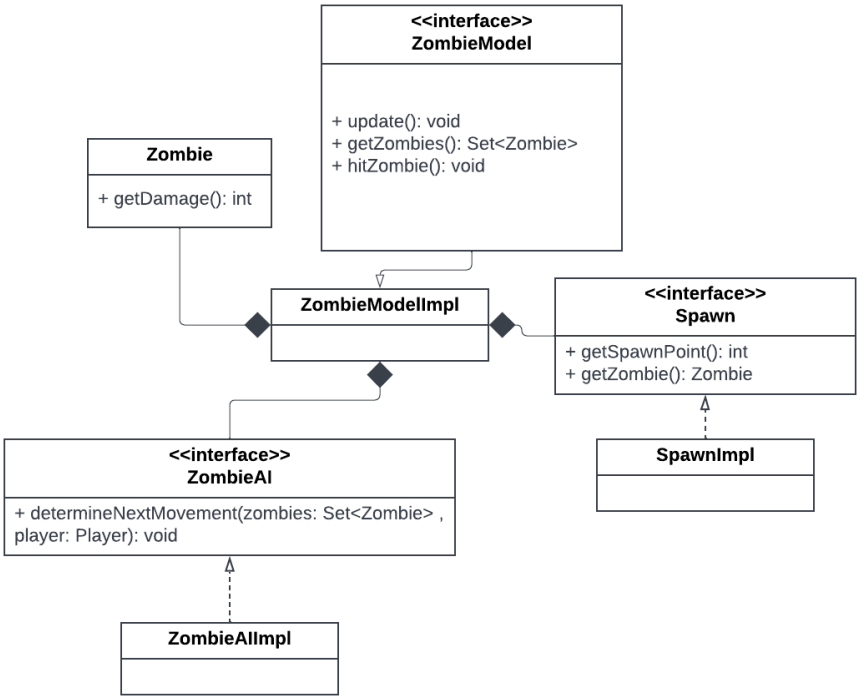

Figura 2.12: Schema UML della Zombie model.

ZombieView e ZombieViewImpl sono invece le classi che contengono i metodi per costruire la "vista" a schermo dello zombie che viene renderizzata dalla GameView sotto direttive MVC dello ZombieController.

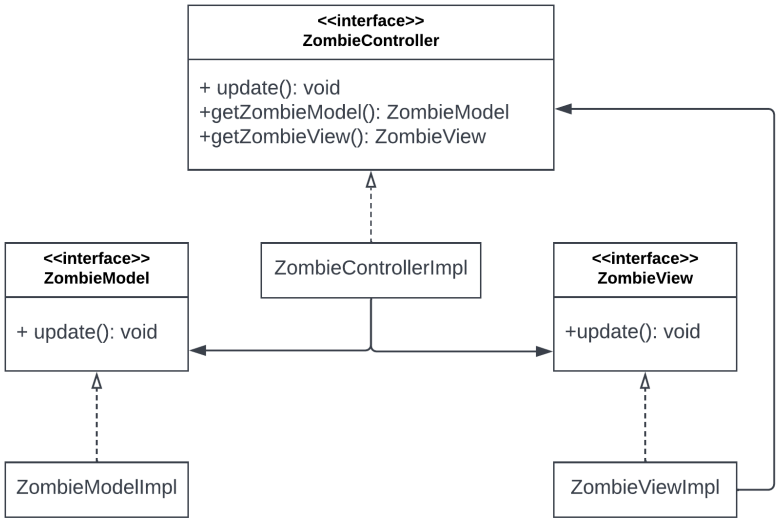

Figura 2.13: Schema UML della struttura zombie.

Ho modellato la gestione dei rounds tramite Round e RoundImpl. Qui si possono decidere il numero di zombie da spawnare ad ogni round con relativo moltiplicatore in base al loro aumentare, un metodo update controlla il loro flow gestendo il tempo che intercorre tra la fine di un round e l'inizio di quello successivo. Quest'ultimo metodo viene richiamato all'interno di RoundController e attraverso il pattern MVC viene gestita anche la RoundView, presentando a schermo le informazioni inerenti.

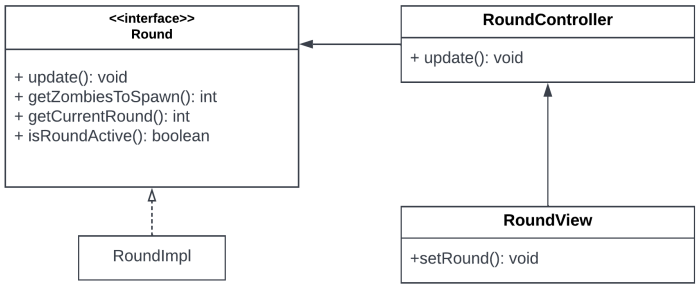

Figura 2.14: Schema UML del round flow.

Nicola Milandri

La parte di mia competenza comprendeva la quasi totale gestione del livello, della generazione della view tramite l'utilizzo di una TileFactory e dell'implementazione delle Ammo.

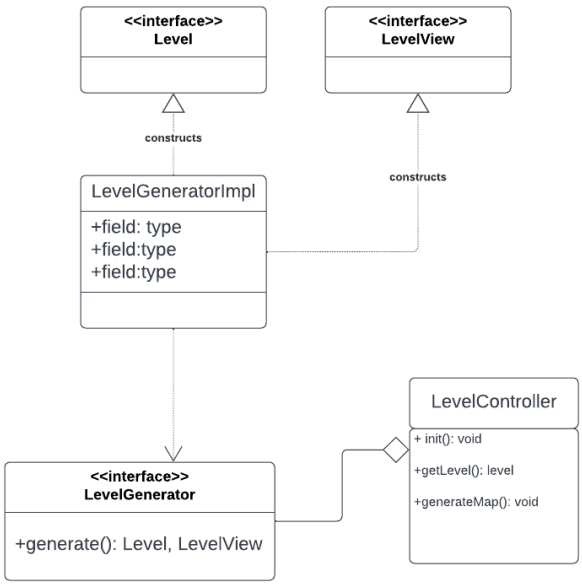

Figura 2.15: Schema UML per LevelGenerator.

Per l'implementazione del terreno di gioco, ho deciso di scomporre la mappa in quadrati detti Tile, i quali avranno una posizione relativa e un tipo associato. I Tile saranno utilizzati principalmente nella fase di View. Ho deciso di generare la mappa partendo dall'inserimento degli indici dei Tile all'interno di un file di testo, in modo da formare idealmente la mappa di gioco che verrà poi letta da un bufferedReader, il quale salverà le informazioni in una Map di Point2D, Integer. Si avrà quindi una separazione frà View (LevelView) e Model (Level) che avranno in comune solo l'associazione di tipo indice-TileType, entrambi sono basati sullo stesso set di dati: la collezione dei Tile. QUesto permette al LevelGenerator di poter generare degli oggetti di tipo Level e LevelView, grazie ai quali sarà poi possibile visualizzare la mappa. Questi ultimi verranno poi gestiti dal LevelController durante l'esecuzione.

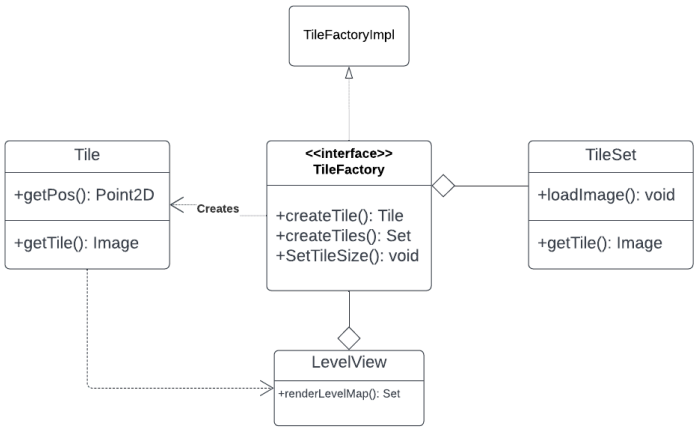

Figura 2.16: Schema UML per TileFactory.

LevelView viene popolata tramite l'utilizzo di una TileFactory, generata attraverso l'utilizzo del pattern Factory, la quale si occuperà di generare gli oggetti Tile, e quindi il terreno, partendo dal set di indici fornitole. Level conterrà le informazioni riguardo alle posizioni degli oggetti statici del gioco quali walls, zombieSpawnPoint e ammoSpawnPoints.

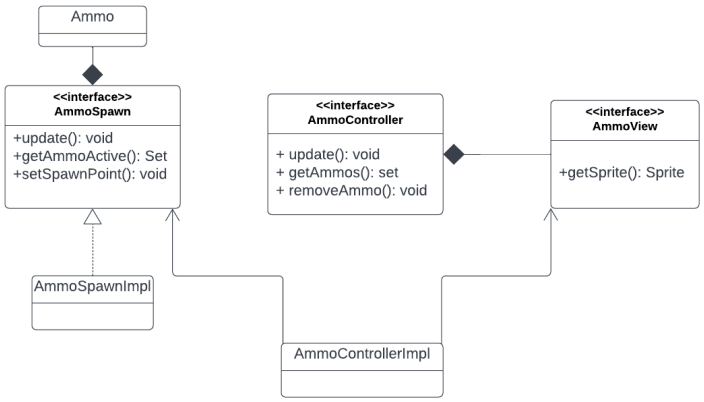

Figura 2.17: Schema UML della gestione munizioni

Per lo sviluppo delle munizioni la classe di partenza è Ammo, che estende la classe AbstractEntity, per le specifiche rimando all'analisi di Stefano Vanucci. La classe permette di modellare delle ammoBox che una volta raccolte ricaricheranno completamente il caricatore dell'arma che si sta utilizzando e rimetteranno in salute il player aumentando 25 HP. Il sistema di generazione delle Ammo è stata realizzata tramite l'interfaccia AmmoSpawn che fornisce i metodi per la generazione e rimozione delle Ammo e la sua implementazione AmmoSpawnImpl. La parte di view è gestita attraverso l'interfaccia AmmoView che permette di generare lo sprite che sarà poi utilizzato dalla GameView. Ammo e AmmoView sono collegati da AmmoController, seguendo il pattern MVC.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Classi di testing sono state utilizzate per valutare l'effettiva efficienza del gioco. JUnit è la libreria che è stata impiegata per la realizzazione dei seguenti tests:

- **PlayerTest**: controllo movimento e collisioni
- **ZombieTest**: controllo movimento, collisioni, spawning e view removing
- **Scoretest**: per la gestione del punteggio a fine gioco
- **GunTest**: per testare gli attacchi e le armi (esaustivo perché testabile visivamente tramite GUI)
- **LevelGeneratorTest**: controllo della corretta lettura della mappa da file di testo.

3.2 Metodologia di lavoro

L'utilizzo di DVCS GIT è stato fondamentale permettendoci di lavorare in maniera organizzata senza dare alcun fastidio al lavoro altrui. Per semplificare abbiamo dedicato molto tempo al lavoro in gruppo mantenendo però un lavoro autonomo sulle classi di appartenenza. Per questo abbiamo scelto di gestire un solo branch per pushare soltanto le modifiche importanti o fix necessari al corretto funzionamento della logica del programma.

Stefano Vanucci

La mia parte di lavoro consisteva nell'implementazione del Player, nella gestione degli input da tastiera e in qualche lavoro di utility tra cui la creazione degli Sprite ed una semplice gestione dell'audio in background. Per ottimizzare il

tempo a disposizione ho anche collaborato alla creazione di alcuni aspetti "secondari" come la gestione dei round. La difficoltà maggiore per me è stata la progettazione iniziale che ha richiesto l'analisi di progetti di altri studenti per capire come e secondo quale logica strutturare il lavoro. Una volta iniziata la scrittura del codice mi sono servito di forum per risolvere eventuali problemi in fase di sviluppo, e della documentazione per capire il corretto funzionamento di JavaFX. Il problema maggiore è stata la gestione degli sprite, in particolare quella del cambio immagine del player seguendo le otto direzioni che in fase iniziale risultava estremamente complicata e non funzionante.

Emiliano Rattini

La mia parte di lavoro consisteva nell'implementazione delle armi e dei proiettili. Mi sono poi spostato sul punteggio, il cambio di scene, la lettura da file. La fase progettuale è stata un po' complessa in principio ed è stato necessario analizzare progetti di altri studenti per ambientarsi. Contemporaneamente ho cercato e imparato alcuni pattern che avrei potuto utilizzare. In fase di implementazione non ho riscontrato particolari difficoltà nello sviluppo delle armi, tuttalpiù ne ho riscontrata qualcuna per la scrittura la file e per il cambio di scene, poichè ho dovuto imparare l'uso di FXML e soprattutto le gestione delle scene di JavaFX. Infine ho aggiunto un paio di View quali la PauseView e la EndView, utili a gestire la pause e la fine della partita, con qualche statistica.

Lorenzo Rubboli

I principali compiti a me assegnati riguardavano la gestione degli zombie, la loro intelligenza artificiale e il round flow. Ho innanzitutto scritto la classe Zombie alla quale ho esteso il model delle entità per ereditare quei metodi utili al setup e alla gestione delle caratteristiche degli zombie. In seguito ho implementato il model dello zombie per gestire danni, kills, spawns e score allegato. L'intelligenza artificiale è stata logicamente la classe che ha richiesto più impegno, lo studio preliminare e l'utilizzo di set, stream, lamba e collection ha reso questa classe la più interessante e al contempo ardua. Lo zombie spawning è stato realizzato in SpawnImpl in cui sono dichiarati i danni, la velocità e la vita da assumere. La parte di view degli zombie è implementata in ZombieViewImpl in cui un update aggiorna la posizione degli sprites e gestisce spawn e rimozione grafica. Model e view sono richiamate dallo ZombieController per l'intero funzionamento. L'ultimo macro-compito è stato realizzare la struttura dei rounds: RoundImpl gestisce il numero di zombie da spawnare, il timing di intercorrenza tra i rounds e il loro corretto flow. Per concludere ho contribuito ad arricchire altre classi come il GameLevelImpl,GameState e MenuView.

Nicola Milandri

La mia parte consisteva nella generazione della mappa di gioco e l'implementazione delle munizioni. La fase di progettazione iniziale è stata particolarmente impegnativa, ha richiesto tempo per poter analizzare progetti di altri gruppi e vari siti per poter apprendere il modo in cui si sarebbe potuto affrontare il lavoro. In fase di sviluppo ho avuto difficoltà con la generazione della mappa, ero partito con una strategia della quale non avevo appreso appieno le caratteristiche, per questo ho deciso poi di virare sulla generazione della mappa attraverso la lettura da file di testo, mentre per le munizioni non ho riscontrato grosse problematiche.

3.3 Note di sviluppo

Stefano Vanucci

- **Stream**: necessarie per lavorare sulle collection
- **Lambda**: utilizzate per le stream
- **JavaFX**: libreria fondamentale per il progetto usata in diverse classi
- **Reflections**: usate per il caricamento delle risorse esterne negli Sprite.

Emiliano Rattini

- **Stream**: utilizzate ove possibile per comodità e leggibilità, come per esempio all'interno dello ShotController e ShotManager
- **Lambda**: utilizzate nei forEach per comodità e leggibilità del codice
- **JavaFX e FXML**: per la gestione della schermata di gioco, e schermata di pausa e di fine gioco
- **Optional**: utilizzati per il ritorno del metodo attack() delle Gun, poichè se si cerca di sparare troppo in fretta viene restituito in Optional vuoto.

Lorenzo Rubboli

- **Stream**: impiegate in ZombieAIImpl per lavorare su set e collection usando le funzione map e collect
- **Lambda**: utilizzate per le stream in ZombieAIImpl e ZombieView
- **JavaFX**: impiegata nella maggior parti delle classi JavaFX.geometry per la gestione dei punti (posizioni) e JavaFX.util per l'utilizzo di Set e Pair.

Nicola Milandri

- **Stream**: utilizzati in particolar modo nelle classi riguardanti le ammo, ma anche nella LevelImpl
- **Lambda**: utilizzate in concomitanza con gli Stream
- **JavaFX**: molto utilizzato soprattutto per la parte di view del progetto.

Capitolo 4

Commenti finali

La creazione di questo progetto di esame è stata affrontata con totale inesperienza dall'intero gruppo, al di fuori di quella didattica. Questo ha permesso un notevole incremento nella ricerca di correlazione di mansioni, analisi, progettazione e sviluppo. L'utilizzo di svariate fonti web, di materiale didattico e di conoscenza personale ha permesso un miglior progresso lavorativo.

4.1 Autovalutazione e lavori futuri

Stefano Vanucci

Mi ritengo piuttosto soddisfatto del lavoro svolto nonostante le difficoltà in fase iniziale. Per quanto mi riguarda, mi sentivo molto inesperto e impreparato ma con un po' di pazienza ho cercato di comprendere a pieno la natura di un progetto di queste dimensioni. Inoltre, il lavoro in gruppo è stato estremamente utile in quanto ogni membro era disponibile per chiarimenti o scambi di idee. La natura pratica del progetto mi ha permesso di affacciarmi sul mondo del lavoro, dandomi un assaggio di ciò che mi potrebbe aspettare in futuro. Sicuramente il codice presenta delle imperfezioni ma tutto sommato con il tempo sono riuscito a migliorare alcune implementazioni, come nel caso della classe Sprite utilizzando gli aspetti avanzati di programmazione.

Emiliano Rattini

Nonostante l'indecisione e le alcune difficoltà iniziali, posso credere di provare abbastanza soddisfazione nel vedere completato il lavoro. La fase di progettazione ha infatti avuto qualche difficoltà a spiccare il volo, anche per l'inesperienza del gruppo. E' stato a mio parere utile vedere i vari pattern e analizzare progetti passati. Non è stato semplicissimo neanche interfacciarsi agli altri e suddividere in maniera il più equa possibile i vari pezzi del progetto. Col tempo ,però

,questa prerogativa è diventata sempre più comoda, poiché suddividendosi il lavoro si risparmia tempo e fatica. Sono sicuro che l'esperienza sia stata alquanto costruttiva e utile a sviluppare la consapevolezza del lavoro di gruppo e sicuramente a guadagnare esperienza nel settore.

Lorenzo Rubboli

Produrre un primissimo applicativo sinergizzando il lavoro di molti nella creazione di un solo prodotto e uscire dai canoni standard di esercizio accademico è stato sicuramente efficace. Progettare e costruire una struttura di lavoro in un progetto più importante di quelli affrontati durante l'anno accademico rende la giusta idea di come porsi al mondo lavorativo. Ho riempito lacune preesistenti in materia e durante lo sviluppo se ne sono prodotte di nuove, portandomi ad approfondire ulteriormente da problemi di casi passati in rete. E' stata confermata la buona volontà da applicare nella gestione del codice sorgente e delle risorse del progetto. Essere chiari e ordinati in modo tale da facilitare il lavoro a interessati e collaboratori è il modo più funzionale per approcciare a questi sviluppi.

Nicola Milandri

Il risultato finale non è esattamente quello sperato, ma mi ritengo abbastanza soddisfatto date le numerose difficoltà affrontate date dall'inesperienza nell'affrontare una sfida di questo tipo. Credo però che questo sia stato un ottimo banco di prova per cimentarsi nella creazione di un progetto complesso, abbiamo avuto la possibilità di saggiare con mano le varie problematiche date dalle varie fasi richiesta per ottenere poi il prodotto finale, per questo motivo non credo valga la pena andare avanti con lo sviluppo di questo progetto, ma, piuttosto, grazie all'esperienza acquisita, preferirei avere la possibilità di verificare l'apprendimento in un nuovo percorso.