Programmazione ad oggetti

Relazione di progetto: Battle for Honor

Realizzato da:
Alice Conti
Emily Frini
Edoardo Montanari
Olivia Rannick Nguemo

Indice

Capitolo 1

Analisi

Il software, parte della prova per il corso di Programmazione ad Oggetti del Corso di Ingegneria Informatica e Scienze Informatiche all'Università di Bologna, implementa un gioco a turni simile al già esistente Mortal Glory, dove il giocatore affronta uno o più nemici in un'arena. Ispirandosi alle sfide romane di gladiatori contro terribili nemici, il personaggio principale dovrà affrontare avversari via via sempre più feroci e imbattibili. Il videogame, a giocatore singolo, presenta una mappa di gioco dove l'eroe ed i nemici si potranno muovere, aggirare ostacoli e attaccarsi a vicenda. Lo scopo del gioco è riuscire ad uccidere tutti i nemici rimanendo in vita, ottenere sempre più oro e esperienza e avanzare di livello. Il giocatore controlla il movimento dell'eroe all'interno della mappa, ad ogni turno ha a sua disposizione 3 azioni da compiere, per cercare di avvicinarsi e uccidere il nemico. I nemici hanno caratteristiche uniche a livello di attacco e vita, diventando sempre più forti al crescere di livello del giocatore.

1.1 Requisiti

Requisiti funzionali

- All'avvio apparirà un menù principale che permette all'utente di iniziare una nuova partita
- Ad ogni nuova partita verrà generato un terreno di gioco con ostacoli e nemici in posizioni casuali.
- I nemici si muoveranno nella mappa aggirando gli ostacoli e avvicinandosi all'eroe
- Durante il turno di gioco il personaggio potrà compiere un'azione movimento (su, giù, destra sinistra) o un'azione attacco

Requisiti non funzionali

- Il giocatore potrà scegliere gli equipaggiamenti e le armi da usare per migliorare le proprie statistiche
- Il giocatore potrà aumentare di livello una volta accumulato un certo livello di esperienza

- I nemici diventeranno sempre più forti più il giocatore avanza di livello
- Oltre al semplice attacco il giocatore potrà scegliere fra una o più abilità speciali
- Durante il gioco il giocatore potrà raccogliere oggetti speciali
- Possibilità di chiudere la finestra di gioco

1.2 Analisi e modello del dominio

La sfida avviene all'interno di un'arena, generata all'inizio di ogni partita. L'arena contiene ostacoli di diverso tipo disposti in posizioni casuali e nemici che compaiono anch'essi in posizioni casuali con caratteristiche diverse uno dall'altro. Una volta che il giocatore ha ucciso tutti i nemici, la sua posizione viene mantenuta e l'arena viene rigenerata.

Il giocatore dovrà affrontare i nemici, potendo eseguire solamente un numero limitato di azioni per turno. Alla fine del turno del giocatore i nemici svolgeranno a turno le proprie azioni.

Il software deve essere in grado di muovere in modo intelligente i nemici, facendoli avvicinare sempre di più al giocatore. Quando il giocatore si trova accanto ad uno o più nemici avviene l'attacco semplicemente effettuando uno spostamento nella posizione del nemico designato.

- Gli **ostacoli** possono essere di tipo diverso (mud, rock, puddle, fence...), e a seconda della loro tipologia, influenzeranno in maniera diversa il giocatore. Alcuni potranno essere attraversabili, altri potranno causare danno al giocatore, ecc...
- Il **giocatore** è caratterizzato da una vita, un'esperienza e una quantità di oro, che guadagna uccidendo i nemici. Quando il giocatore viene attaccato subisce un danno che dipende dall'avversario e di conseguenza perde punti vita. Il giocatore può usufruire delle sue abilità speciali per compiere un'azione più forte e per diventare più potente.
- Le **abilità**, possono essere utilizzate un numero limitato di volte per partita e permettono al giocatore di effettuare azioni speciali. L'utente sceglie dall'interfaccia l'abilità desiderata durante il proprio turno, usando un punto azione. Una volta finita la partita, verranno ripristinati gli utilizzi di tutte le abilità così da poter essere nuovamente utilizzabili nella mappa successiva.
- Il **nemico** è caratterizzato da vita, attacco, difesa, ecc.. Alla morte di un nemico da parte di un attacco del giocatore, quest'ultimo otterrà oro ed esperienza per poter diventare più forte, in quanto con l'avanzare del tempo i nemici diverranno più difficili da sconfiggere. Nel caso in cui un nemico riesca ad uccidere il giocatore, la partita finisce.

In seguito si riporta uno schema del modello del dominio

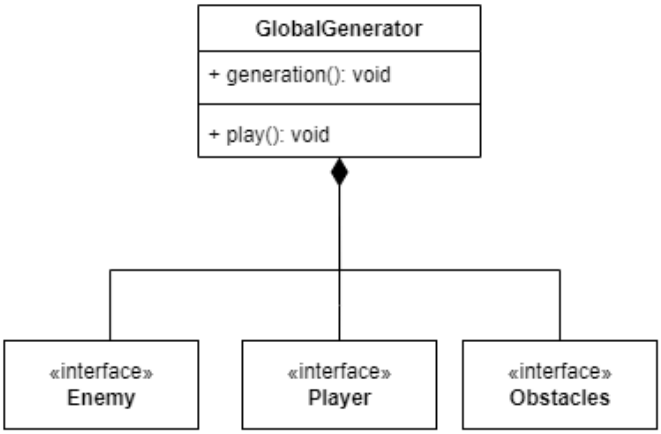

Figura 1.1: Schema del modello del dominio

Capitolo 2

Design

2.1 Architettura

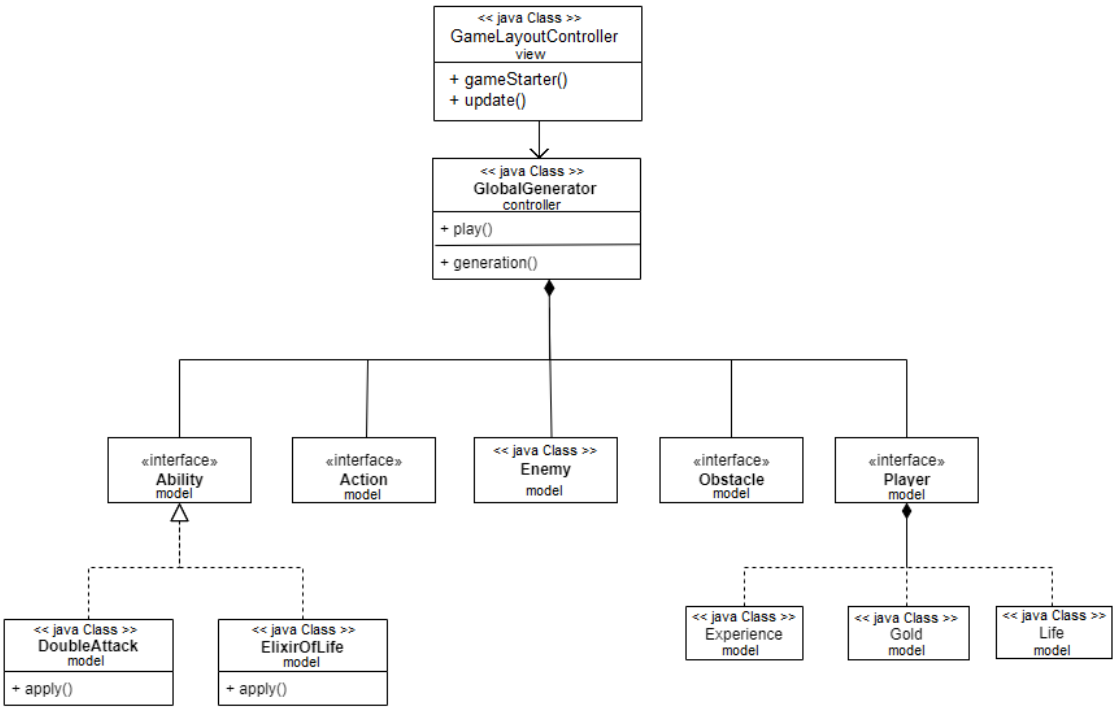

Figura 2.1: Schema UML dell'applicazione

Il progetto Battle For Honor utilizza al suo interno il pattern architetturale MVC (Model View Controller) permettendo una suddivisione chiara della struttura del codice e una gestione in modo efficace dell'interfaccia di gioco, della logica interna e delle entità; quest'ultime in particolare grazie al pattern *Factory Method* vengono create e disposte nell'interfaccia in modo modulare.

Dato che la mappa e tutte le sue entità devono essere uniche all'interno dell'applicazione si è deciso di implementare la classe `GlobalGenerator` con il pattern `Singleton`.

Ogni componente di gioco ha un proprio controller utilizzato per coordinare le singole **model** con la classe principale del nostro progetto `GlobalGenerator`; questa classe contiene tutte le informazioni vitali per il corretto svolgimento della partita, generando le entità e gestendo la meccanica stessa del gioco. L'aggiornamento generale della **view** viene delegato invece a `GameLayoutController` che si occupa del rendering delle entità sulla mappa e di catturare gli input dalla finestra di gioco. Agendo basandosi sul pattern *Observer*, la classe recupera gli oggetti generati in `GlobalGenerator` e rimane in attesa di modifiche su di essi; intercettando tali modifiche tramite opportuni metodi, ne modificherà i valori nel model e refresherà la grafica dell'interfaccia per combaciare con i cambiamenti subiti. In questo modo è possibile aggiungere nuovi input e relative funzioni senza modificare la logica di base dell'applicazione.

In Figura 2.1 si riporta uno schema UML dell'architettura di interazione tra gli elementi principali del gioco

Il controller rappresenta il core del nostro applicativo, svolgendo la funzione di comunicazione tra model e view.

Inoltre ogni entità ha un suo controller dedicato che viene interrogato, affinchè l'azione eseguita venga processata correttamente e di conseguenza visualizzata a schermo.

2.2 Design dettagliato

2.2.1 Alice Conti

Nel corso dello sviluppo del progetto mi sono occupata principalmente dell'implementazione degli ostacoli, della gestione dei punti azione e delle abilità. La logica con cui ho implementato i vari elementi è stata quella di creare qualcosa di facilmente estendibile, rispettando i principi DRY e KISS.

Ostacoli

Nel caso degli ostacoli ho voluto implementare la possibilità di crearne di tipo diverso, delegando il compito di generare ostacoli a un'entità che si occupasse unicamente di quello, rimanendo indipendente dall'implementazione degli ostacoli stessi. Dato che la mia priorità era creare qualcosa di facilmente estendibile che non necessitasse modifiche a cascata sugli altri componenti, ho deciso di utilizzare il **Pattern Factory Method**, secondo quanto riportato nella Figura 2.2

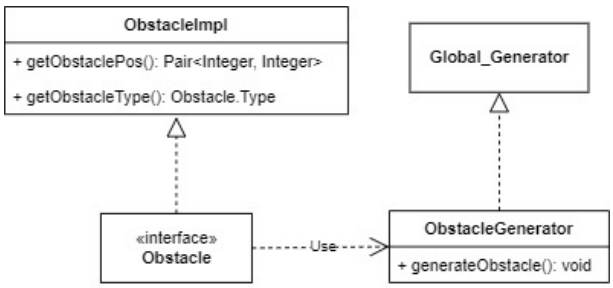

Figura 2.2: Schema UML del Pattern Factory Method nel caso della generazione degli ostacoli

In questo modo `GlobalGenerator` comunica unicamente con `ObstacleGenerator`, che tramite il metodo `generateObstacle()` istanzia gli ostacoli necessari. Per rappresentare il concetto di ostacoli diversi ho deciso di caratterizzare ogni oggetto ostacolo con un tipo `ObstacleType`, siccome tutto il resto è in comune in quanto creare una classe per ogni `ObstacleType` risultava ridondante. In questo modo i personaggi all'interno dell'arena potranno riconoscere il tipo di ostacolo e agire di conseguenza. Inoltre l'aggiunta di altri tipi di ostacoli non richiede eccessive modifiche nel controller. Gli ostacoli ad oggi implementati rappresentanto solo due tipi di ostacolo, *POOL* e *ROCK*, ma non si esclude la possibilità che un'implementazione futura possa prevederne altri tipi. Gli ostacoli, essendo oggetti statici all'interno dell'arena, hanno come uniche caratteristiche la posizione e il tipo. La loro generazione prevede il calcolo di una posizione randomica scegliendola tra le posizioni libere all'interno dell'arena, per evitare sovrapposizioni con gli altri elementi di gioco.

Punti azione

I punti azione rappresentano il numero di azioni possibili per round del giocatore e del nemico. Per rendere la logica di un'azione indipendente dall'entità che effettivamente le usa ho immaginato un oggetto a sè stante, che tramite un contratto potesse essere anche riusata da implementazioni future dell'applicazione. Il contratto quindi specifica tutte le logiche relative all'accesso e alla modifica dei dati.

Siccome dall'analisi non sono risultate difficoltà particolari, non è stato utilizzato alcun pattern specifico. In figura Figura 2.3 si riporta lo schema UML relativo.

Figura 2.3: Schema UML delle azioni

Abilità

Le abilità consentono al giocatore di diventare più forte a seconda del tipo di abilità speciale utilizzata. Per catturare il concetto di abilità in modo indipendente dal suo tipo specifico ho immaginato l'interfaccia Ability. Tutte le classi che la implementano dovranno soddisfare il metodo `apply()`, che applica l'abilità. Anche in questo caso ho cercato di immaginare una soluzione meno statica possibile, e che rendesse le abilità facilmente estendibili. Ho quindi scelto di usare il **Pattern Factory Method**, secondo quanto riportato nella Figura 2.4

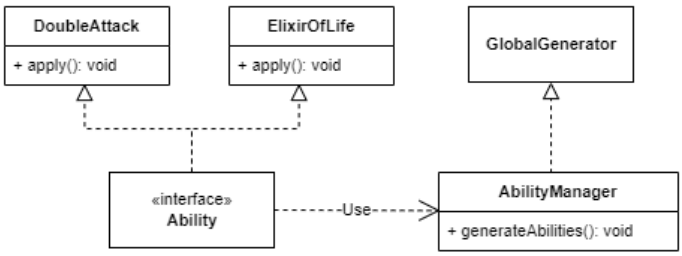

Figura 2.4: Schema UML

In questo modo aggiungere in futuro altre abilità non va ad intaccare le entità che le usano.

2.2.2 Emily Frini

La prima classe a venire caricata nel Main è `CoreLauncher`, chiamata così in quanto visualizza la prima Stage e fa da base per l'apertura dello Stage principale contenente il gioco vero e proprio. CoreLauncher estende direttamente da Application implementando il metodo `start()`; in questo metodo viene caricato lo Stage su cui viene inserita la prima Scena con la sua configurazione iniziale. Essa ha una propria dimensione stabilita e viene caricata assieme ad alcuni metodi di personalizzazione, al suo foglio di stile ed ad un metodo utilitaristico che permette di trascinare lo Stage anche quando il suo `StageStyle` è settato su **undecorated**.

Figura 2.5: Schema UML della schermata di avvio del gioco.

Schermate

Battle For Honor è costituito da sole 2 schermate. Inizialmente sarebbero dovute essere maggiori ma in seguito ad un taglio di contenuti, le corrispondenti schermate sono state droppate. Sono state realizzate utilizzando il tool SceneBuilder che genera automaticamente codice in formato XML (Extensible Markup Language) comprensibile alla libreria JavaFX. Ognuna di queste ha un ControllerFX associato che, oltre a definire i metodi associati agli eventi della GUI, implementa il metodo `initialize()` in cui è possibile accedere ai componenti definiti nel file FXML, taggati tramite @FXML.

Non sono stati utilizzati metodi di `switch` tra le schermate in quanto, oltre ad essere solo 2 schermate, la prima, Figura 2.5, ha scopo puramente estetico e viene caricata esclusivamente al lancio dell'applicazione stessa facendo da "apertura" per la vera e propria schermata di gioco, attraverso il metodo `move()` che "muove" il gioco alla schermata successiva. Inoltre, la grafica dell'applicazione è stata scritta in CSS, in modo da alleggerire le View ed esternalizzare gli aspetti grafici, permettendo di utilizzare una formattazione condivisibile ma non ripetitiva per tutto il gioco, in modo semplice.

Senza dubbio la Scena più importante dell'applicazione è `GameLayout` e il corrispondente ControllerFX. Quest'ultimo fa da ponte tra ciò che viene creato nel `GlobalGenerator` e ciò che verrà poi visualizzato a video. Inoltre si occupa di gestire gli input dell'utente, raccogliendo gli eventi relativi ai vari bottoni. Infine, riconosce determinate situazioni (come ad esempio la morte del giocatore) e le gestisce di conseguenza.

Il metodo centrale di `GameLayoutController` è `gameStarter()`. Chiamato immediatamente dentro `initialize()`, si occupa di generare le varie entità e la griglia dell'Arena, salvando le celle all'interno di due map che mantengono

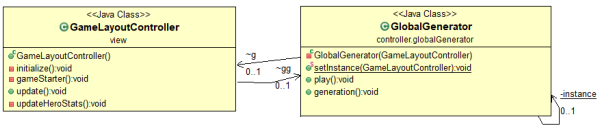

Figura 2.6: Schema UML del rapporto tra View e Controller

il riferimento tra la cella e la sua posizione, utilizzato dalle altre classi per il calcolo dello "spawn" delle entità, del movimento del giocatore/nemici e infine per controllo della collisione. All'interno chiama anche il metodo `update()` che, come suggerisce il nome, si occupa di fare l'aggiornamento della View. In particolare, per ogni cella della griglia viene effettuato un controllo sul contenuto e disegnato il corrispondente *sprite* sulla parte grafica. Per come è stato gestito il gioco, è possibile avere infinite entità nemiche ed è grazie all'uso del controllo sull'ID del nemico (spiegato in un'altra sezione) che si basa il meccanismo di disegno: infatti ad un dato ID è associato un certo *sprite* e se la posizione del nemico corrisponde alle coordinate della cella, allora lo *sprite* corretto verrà visualizzato. Questo stesso procedimento è stato utilizzato anche per la visualizzazione a schermo degli ostacoli (il controllo viene fatto sul tipo di ostacolo) e del giocatore stesso. Richiamando poi il metodo `play()` presente nel `GlobalGenerator` viene inizializzato il primo Round e di fatto comincia il turno del giocatore. Durante il turno del giocatore l'interfaccia rimane in attesa di un input da parte dell'utente. Finché non sono state eseguite 3 azioni (decrementate quando viene premuto uno dei pulsanti movimento o delle abilità), il gioco non avanza e solo quando le 3 azioni sono state eseguite (seguite ogni volta da un `update()`) allora si entra nel turno nemico che viene effettuato e rilascia il controllo all'utente al termine.

In seguito ad ogni update, vengono effettuati non solo *refresh* grafici, ma anche testuali. All'interno di tre componenti testuali è possibile vedere in *real time* ciò che accade nella parte più interna del gioco:

- Una `textArea` si occupa di reperire le statistiche del personaggio, aggiornando dopo ogni movimento i valori (in caso `Experience` o `Level` siano saliti o `Life` sia scesa) e ricordando all'Utente gli attuali punti Azione disponibili.

- Una `Label` mantiene salvati i `Gold` del giocatore. Quando viene utilizzata l'abilità Elixir of Life, all'interno della classe viene effettuato un controllo sui soldi del giocatore e, se sufficienti, il valore viene poi decrementato successivamente all'uso. Quando viene ucciso un nemico invece, l'evento viene intercettato dal `GlobalGenerator` che aggiorna la variabile interna, richiamata poi nella `Label`.

- Una `textArea` in cui vengono reindirizzate tutte le print della console. La maggior parte degli eventi vengono risolti internamente al gioco e non sono effettivamente visibili da fuori. Per i nostri test, è stata usata spesso la print sulla console per controllare lo stato interno della partita ma questa non poteva essere una soluzione finale. Utilizzando `System.setOut(new PrintStream(System.out)` è stato possibile cambiare l'output interno

dell'applicazione, passando dalla console ad un altro output (nel mio caso questa `textArea`), realizzando una vera e propria cronologia degli eventi consultabile dall'Utente.

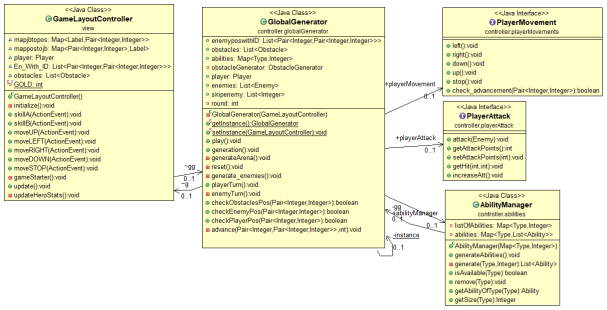

Figura 2.7: Schema UML.

Piccola considerazione per `SharedMethods`, una interfaccia contenente metodi che modificano la parte estetica della GUI. Lavorando per gusti personali con finestre **undecorated**, è stata creata ed estesa poi dalle View, raccogliendo i metodi comuni alle Scene, quali il cursore personalizzato e la possibilità di trascinare la Scena stessa.

Per come è stata realizzata la parte View, essa è completamente sostituibile. Sono in grado di affermare ciò perchè inizialmente avevo preparato una piccola GUI in Java.swing per poter far lavorare i miei compagni su un feedback grafico. Di base, aveva solo un paio di campi per le statistiche del giocatore e una griglia di bottoni non interagibili, in cui veniva cambiato il testo in base al contenuto e l'utente si spostava utilizzando l'input da tastiera sulla console di Eclipse. Quando poi ho creato la Scena in JavaFX ho ricollegato le cose con minimi cambiamenti; di fatti l'unica modifica più importante l'ho avuta all'interno di `GlobalGenerator`, dove il ciclo while, in cui turno Player e turno Enemy si alternavano, è stato riscritto in un metodo iterativo a causa di un problema con il thread di JavaFX. Questo perchè JavaFX è basato su eventi. Se viene utilizzato un ciclo o una logica di lunga durata nel thread dell'applicazione JavaFX, JavaFX non può elaborare i suoi eventi, inclusi quelli che causano la visualizzazione di finestre e quelli che consentono a una finestra di ridisegnarsi e di rispondere all'input dell'utente. Semplicemente non mi era possibile monitorare i progressi del gioco usando il while che in un JFrame non dava al contrario problemi. Tramite uso di interazioni successive ho potuto liberarmi del while e continuare a gestire correttamente il gioco.

2.2.3 Edoardo Montanari

- nemici
- incremento difficoltà
- oggetti
- inventario

La mia parte di competenza all'interno del progetto consiste nella generazione dei nemici all'interno dell'arena, l'eventuale possibilità che alla morte di un nemico venga rilasciato un oggetto, la gestione dell'inventario del giocatore, e la conseguente gestione del tracking delle posizioni tra giocatore e nemico.

I nemici vengono generati con valori incrementali di HP, def, atk, exp a seconda di quanta esperienza totale abbia accumulato il giocatore, in modo tale da mantenere un grado di sfida accettabile anche con l'avanzare della partita nel tempo. Le tipologie di nemici esistenti all'interno dell'arena possono essere di due tipi, semplice e potenziato. I nemici potenziati hanno la principale caratteristica di essere più pericolosi da affrontare per via della loro statistica di attacco superiore rispetto ad un nemico normale, in compenso se sconfitti lasceranno garantito un oggetto o un bonus positivo per il giocatore.

Alla fine della generazione dei nemici, tutte le informazioni relative alla loro posizione vengono inserite in una classe (**Global generator**) consultabile da tutte le altre classi mantenendo in memoria principalmente l'ID e la posizione del nemico all'interno della griglia di gioco.

Successivamente se la generazione ha avuto successo le restanti informazioni vengono aggiunte alla lista dei nemici attivi.

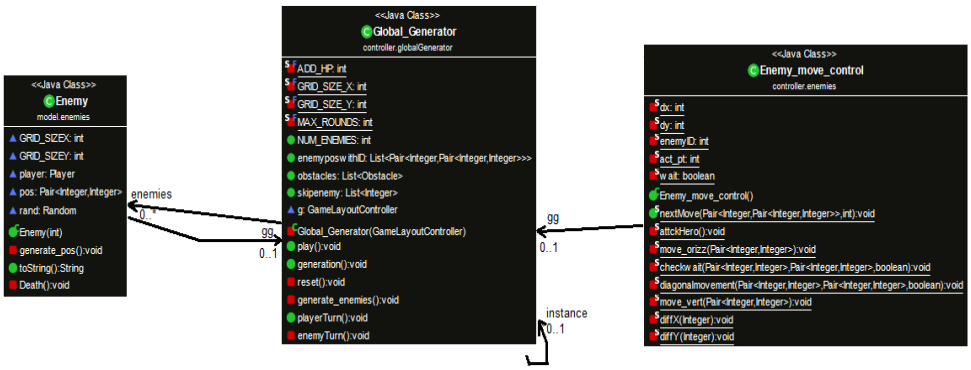

Figura 2.8: Schema UML della generazione dei nemici

La parte più complessa è stata la realizzazione di un algoritmo tale che automaticamente effettuasse la ricerca di passi successivi del nemico che cerca di avvicinarsi al giocatore, in quanto ogni nemico ha a sua disposizione solamente 2 azioni per turno.

In fase di progettazione si è deciso che se una qualsiasi entità all'interno della mappa di gioco non vuole oppure non riesce ad effettuare un'azione, allora l'azione stessa dell'entità viene sprecata, rimanendo ferma.

Ciò si verifica sotto varie condizioni, tra cui:

1. un nemico incontra un ostacolo lungo il percorso.
2. il nemico raggiunge la posizione del giocatore e lo considera come fosse un ostacolo non superabile e in questo caso effettua l'attacco.
3. un nemico incontra un altro nemico.

In special modo la gestione del movimento del nemico contro uno ostacolo presentava una complicazione nel momento in cui il nemico lo volesse aggirare completamente.

La sfida principale è stata quella di effettuare un controllo per cui questa azione fosse possibile solo se essa risulta essere la prima azione che il nemico deve eseguire nel suo turno, e in caso di successo effettuare uno spostamento in diagonale verso destra o sinistra a seconda della situazione, rimanendo successivamente fermo.

Per controllare quante azioni il nemico ha ancora a sua disposizione, è diventato imperativo imporre il passaggio, alla funzione di movimento, di un parametro esterno che indica il quantitativo di azioni a disposizione, in modo tale da renderlo un valore univoco all'interno della classe di controllo.

L'ordine di "attivazione" di ogni nemico dipende dal proprio ordine di inserimento nella lista descritta sopra.

Al momento della morte di un nemico viene memorizzato unicamente il suo ID, così da poterne effettuare la conseguente rimozione dall'arena e dall'ordine di attivazione dei nemici dal turno successivo.

Inoltre viene anche chiamato il generatore di oggetti per determinare quale oggetto/bonus debba essere creato a seconda della tipologia del nemico ucciso, in quanto se dovesse morire un nemico potenziato, il giocatore otterrà sicuramente un premio, mentre se a morire dovesse essere un nemico semplice, allora la casistica di oggetti si allarga anche a possibili malus.

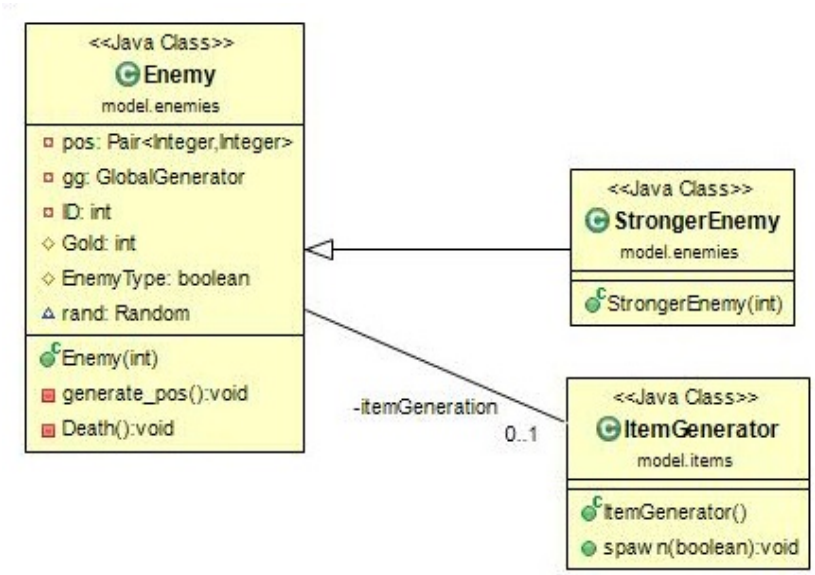

Figura 2.9: Schema UML della generazione degli oggetti

La generazione degli oggetti è gestita dalla classe ItemGenerator che, a seconda della tipologia del nemico che muore, seleziona la lista corretta di scelte effettuabili.

Gli oggetti si dividono in Items ed EquippableItems.

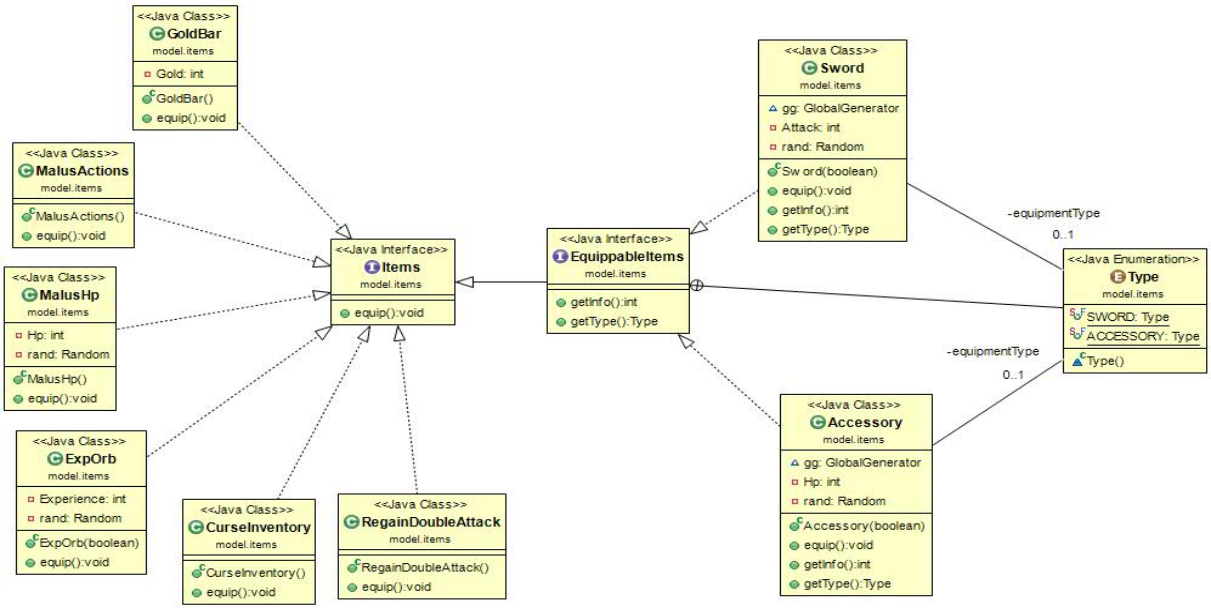

Figura 2.10: Schema UML degli oggetti

Ogni oggetto generico ha il metodo equip() che una volta richiamato applica al giocatore delle modifiche alle sue statistiche sia che esse siano positive o negative con possibile morte del giocatore nel caso quest'ultimo sia molto sfortunato. Gli equippable items hanno la particolarità di essere dei potenziamenti permanenti finché essi rimangono nell'inventario del giocatore.

Non è possibile rimuovere un oggetto di propria volontà; inoltre l'inventario ha una capienza limitata e una volta raggiunta la capienza massima non si potranno ottenere bonus aggiuntivi.

L'unico modo per poter ricominciare ad ottenere oggetti con statistiche migliori è quella di subire il malus della rimozione dell'inventario.

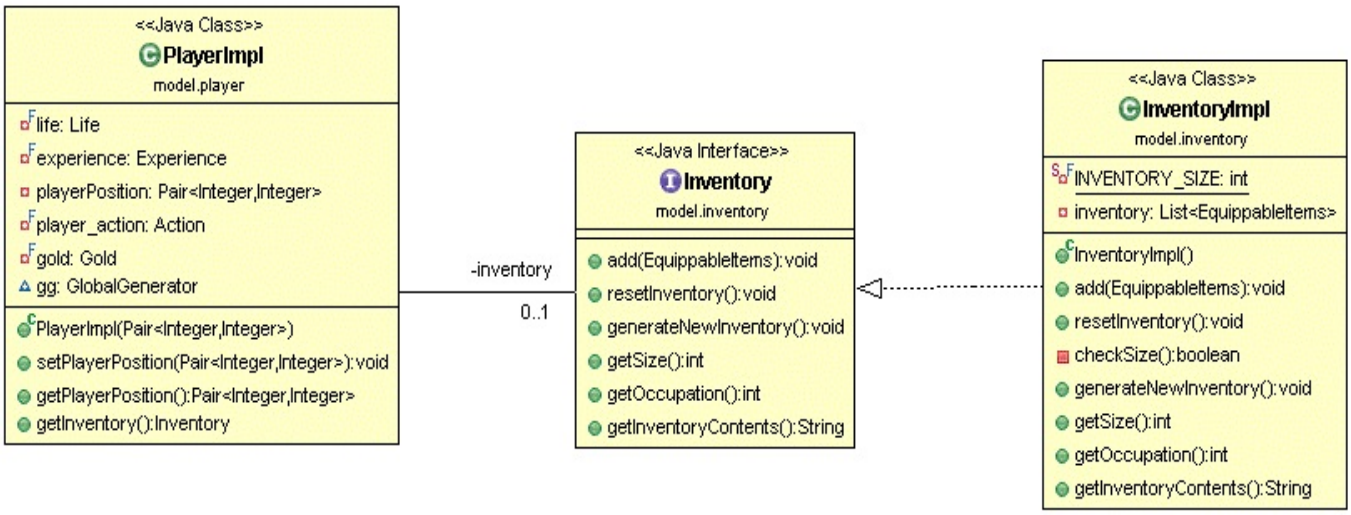

Figura 2.11: Schema UML dell'inventario

2.2.4 Olivia Nguemo

In questo progetto, mi sono occupata della

- gestione dei movimenti del giocatore
- gestione dei punti punti vita, punti esperienza, oro e conseguente implementazione
- aumento di livello del giocatore

Giocatore e punti del gioco

Un giocatore è un'entità che possiede una posizione (x,y) ragione per cui ho utilizzato la classe ***Pair*** per gestire la posizione del giocatore. Le classi ***Player*** e ***PlayerImpl*** rappresentano rispettivamente l'interfaccia del giocatore e la sua relativa implementazione all'interno di PlayerImpl, dove viene anche gestita in dettaglio la posizione del giocatore e il relativo costo in punti azione. Le classi***Life***,***Experience***,***Gold*** rappresentano rispettivamente le classi che gestiscono i punti vita del giocatore, i suoi punti esperienza, e l'oro accumulato uccidendo i nemici. La classe Experience implementa il meccanismo di incremento di livello che controlla se il quantitativo di punti esperienza accumulati fino a quel momento è sufficiente per poter migliorare le statistiche del giocatore. La classe Gold controlla il quantitativo di oro che il giocatore possiede e viene interrogata nel caso si decida di utilizzare l'abilità Elixir of Life, andando a sottrarre una parte dell'oro al giocatore.

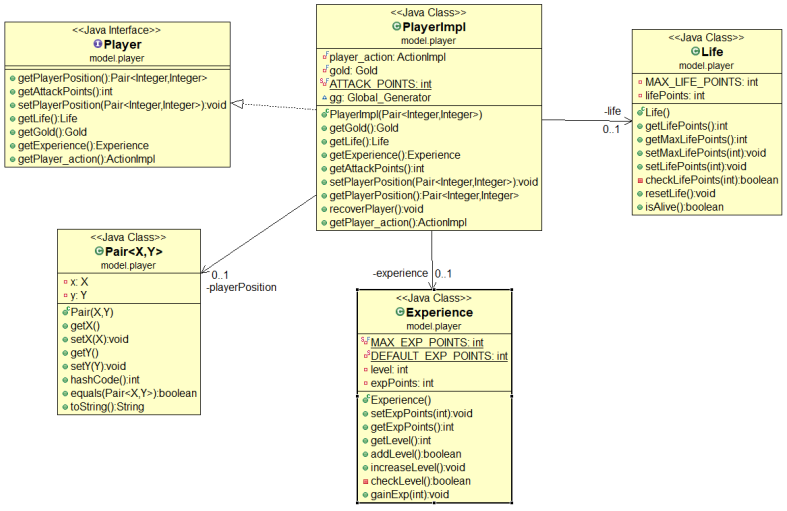

Figura 2.12: Schema UML del player e dei punti del gioco

Movimenti del giocatore

Durante il gioco, il player ha la possibilità di fare alcune azioni che sono "andare a destra, sinistra, davanti, indietro" o "rimanere fermo". Però, prima di muoversi, il giocatore deve assicurarsi che nella posizione che desidera occupare

non ci sia ne' un ostacolo di tipo ROCK (ossia un ostacolo che non si può attraversare), ne' che vada fuori dall'area di gioco. L'ho gestito nella classe ***PlayerMovementsImpl*** che implementa l'interfaccia ***PlayerMovement***.

Nel caso specifico in cui il giocatore volesse spostarsi in una posizione che contiene un nemico, allora si produce un attacco cioè il giocatore affronta il nemico provando ad ucciderlo(finire i suoi punti vita) per poter guadagnare i punti esperienza .

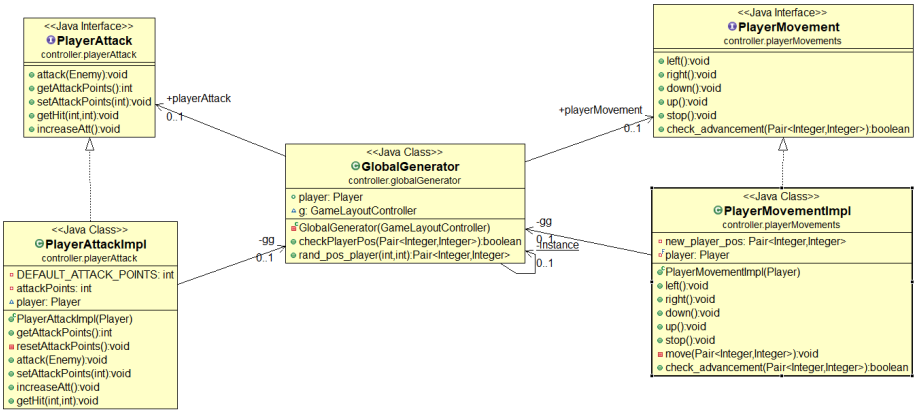

Figura 2.13: Schema UML dei movimenti del player e dell'attacco

Aumento del livello del giocatore

L'aumento di livello del giocatore è vincolato da MAX EXP POINTS punti esperienza (implementato nella classe ***Experience***). Più si aumenta di livello, più i nemici diventano forti(implementato direttamente nella classe ***Enemy***). Anche il giocatore diventa più forte, nel senso che il suo attacco diventa più potente (implementato nella classe ***PlayerAttackImpl***).

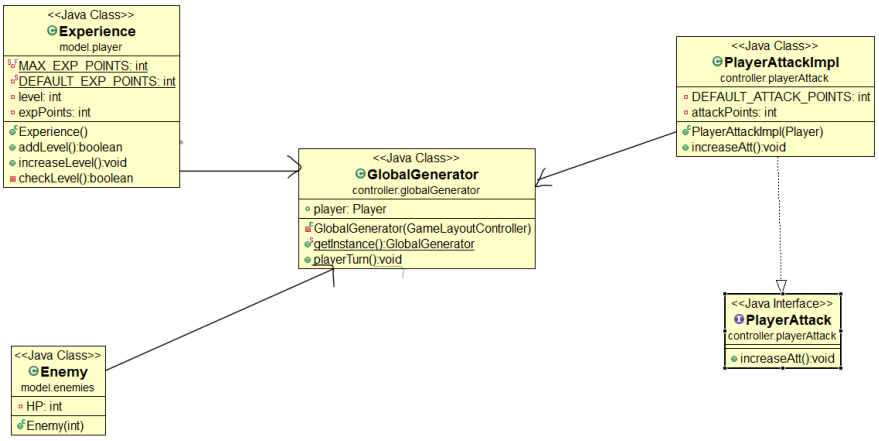

Figura 2.14: Schema UML delle classi che gestiscono l'aumentare di livello

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per testare le funzionalità dei vari componenti realizzati sono stati effettuati più che altro test manuali in cui, lavorando con i file FXML, CSS e i relativi componenti, sono stati controllati come gli *sprite* venivano renderizzati sullo schermo, l'estetica generale dell'interfaccia e infine se la pressione di un tasto comportava i corretti update nello schermo di gioco e nella logica interna.

Per quanto riguarda il controllo del corretto funzionamento del tracking nemico, manualmente sono state fatte prove successive in cui il giocatore si posizionava accanto ad un ostacolo, vicino ai bordi dello schermo oppure lasciandolo fermo, per verificare che in ogni situazione il nemico raggiungesse la posizione esatta da qualsiasi punto di inizio.

Per poter testare i movimenti del giocatore, è stato semplicemente usata la Gui che c'era all'inizio dello sviluppo, ed è stato testato se il giocatore andava fuori dalla griglia, se si sovrapponeva con gli ostacoli o i nemici e se eseguiva i movimenti desiderati cioè up, down, right , left.

3.2 Metodologia di lavoro

Alice Conti

- generazione degli ostacoli;
- gestione punti azione;
- implementazione delle abilità;

Durante la parte iniziale del progetto abbiamo cominciato la fase di analisi e design, per decidere insieme le caratteristiche di gioco e l'architettura generale dell'applicazione.

Successivamente mi sono concentrata sull'implementazione della struttura delle mie parti, cercando di renderle più portabile possibili. In questo forse ho trovato maggiori difficoltà, in quanto era necessario immaginare una struttura che fosse generale e soddisfacesse tutte le possibilità. Parallelamente a questo ho analizzato quale pattern fosse più corretto usare in ogni situazione.

Nell'implementare il codice ho sempre cercato di scrivere un codice pulito e comprensibile anche dagli altri membri del gruppo, decomponendo le varie parti per poterle riusare in caso di necessità.

Emily Frini

- Generazione mappa
- Implementazione interfaccia
- Gestione Menu

All'inizio del progetto ho proposto al gruppo una idea di come il gioco sarebbe dovuto apparire e mostrato loro quali componenti avrei principalmente usato così da coordinarci poi sulla struttura interna delle classi. Ho creato per loro una semplice griglia di bottoni su cui, tramite console, avrebbero potuto testare le loro classi e il corretto funzionamento mentre io lavoravo sull'interfaccia vera e propria. Successivamente parte del mio lavoro è stato cancellato, in quanto alcune funzionalità sono state scartate, comportando così una modifica delle **view** create per accorpare diversamente le cose rimaste. Quando la classe principale del controller, GlobalGenerator, è stata implementata, l'ho modificata per poter connetterla correttamente alle mie classi.

Edoardo Montanari

- generazione dei nemici
- creazione di IA di tracking nemico-giocatore
- incremento statistiche del nemico e conseguente incremento della difficoltà
- generazione oggetti
- inventario del giocatore

Nelle fasi iniziali del progetto mi sono confrontato con i membri del gruppo per decidere come approcciare il problema della generazione dei nemici, ossia decidere se applicare uno schema single thread iterativo, effettuando passaggi successivi prendendo in considerazione un' entità alla volta, oppure applicare un approccio multi thread associando ad ogni nemico un suo thread.

Si è deciso di non applicare la seconda scelta in quanto si è rilevata essere non molto affine alle meccaniche del gioco preso in questione essendo strutturato a turni e di conseguenza ogni entità deve poter eseguire le proprie azioni solo al termine dell' entità precedente.

Successivamente l'aspetto che più mi è risultato complesso è stata la creazione di un metodo che passo passo, controllasse il corretto avanzamento del nemico verso il giocatore, soprattutto tenendo conto delle svariate possibilità in cui esso non debba sovrapporsi con altri elementi di gioco.

Di conseguenza mi sono occupato di controllare che con l'avanzare della partita il gioco non risultasse stagnante ad un grado di difficoltà né troppo elevato né

troppo semplice.

Infine ho deciso che il giocatore dovesse essere obbligatoriamente premiato con un oggetto positivo in caso di morte di un nemico potenziato, mentre con la morte di un nemico semplice volevo che la ricompensa per il giocatore fosse lasciata di più al caso, così da conferire un certo livello di incertezza nell'avanzamento del gioco, atto ad aumentare leggermente il grado di sfida nella schermata di battaglia corrente.

Per quanto riguarda l'inventario del giocatore, ho deciso che dovesse contenere solo un numero estremamente limitato di oggetti, e non rimovibili dal giocatore stesso, perché altrimenti sarebbe risultato fin troppo semplice mantenere di volta in volta solo gli oggetti che conferiscono il bonus maggiore.

Ciò rende l'acquisizione di questo genere di oggetti più interessante, in quanto il giocatore non sa cosa potrà ottenere in futuro: il giocatore infatti cercherà sempre di ottenere il maggior numero di oggetti bonus, e non sapere se ne otterrà uno migliore o peggiore rende il gioco più stimolante.

Olivia Nguemo

Dopo la suddivisione del lavoro, mi sono occupata della

- **gestione dei punti del gioco** in questa parte, mi sono accordata con gli altri membri del gruppo per avere il loro parere sul costo dei diversi punti. Ho cercato di gestire i punti in modo ragionevole in modo che l'utente non trovi il gioco ne troppo difficile, ne troppo facile.

- **gestione dei movimenti del giocatore** la parte la più interessante è stata la verifica della disponibilità della posizione dove il giocatore vuole spostarsi prendendo in considerazione come è stato gestito gli ostacoli e gli nemici.

- **gestione dell'aumento di livello del giocatore** Avevo provato a gestire questa parte in un package a parte ma alla fine, ho trovato meglio aggiungere delle funzionalità per gestirlo nelle classi che esistono già.Ho dovuto prendere in considerazione l'implementazione dei punti vita dei nemici e creare alcune funzioni per rendere alcuni aspetti dinamici come l' incremento dell'attacco.

3.3 Note di sviluppo

Alice Conti

- utilizzo di `stream` e `forEach` per operare più facilmente su collezioni di dati
- utilizzo di `Lambda Expression`
- creazione di `enum` per rappresentare i tipi di ostacoli e di abilità
- sviluppo di algoritmi per controllare la sovrapposizione delle varie entità

Emily Frini

Per realizzare l'interfaccia grafica con JavaFX mi sono appoggiata a SceneBuilder per lo scheletro fxml.

Per lavorare sulla parte più meccanica del gioco, ho fatto uso di:

- `HashMap` per salvare le informazioni riguardo le entità e la loro posizione all'interno dell'area di gioco
- `forEach` per iterare la griglia di gioco e controllare successivamente i
- `Pair`, utilizzati per salvare le entità e la loro posizione.

Sono stati utilizzati anche svariati componenti e i relativi attributi della libreria JavaFX, quali anchorPane, Label, Button, etc.

Edoardo Montanari

- **List** soprattutto utilizzate per avere un contenitore accessibile nei casi in cui si debba effettuare check di controllo per posizioni e ordine di "attivazione" nei turni di gioco.
- **Pair** utilizzato per racchiudere le informazioni sintetiche da utilizzare nella interfaccia grafica ossia identificativo e posizione

Olivia Nguemo

- **Pair**: Ho molto usato la struttura di un pair(X,Y) per rappresentare la posizione del giocatore invece di usare due componenti separati per i componenti x e y.
- **Stream e Lambdas**: per scorrere la lista degli ostacoli nella funzione `checkAdvancement()` della classe PlayerMovementImpl.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Alice Conti

Malgrado le difficoltà incontrate mi sono divertita molto ad immaginare diverse soluzioni architetturali, consapevole del fatto che le scelte prese sono ancora migliorabili. Inoltre mi è piaciuto approfondire le potenzialità di DVCS come Git e dei pattern.

Non essendo un'appassionata di videogame alcune logiche di gioco non mi erano del tutto immediate, ma grazie al continuo confronto con gli altri membri del gruppo ho risolto facilmente tutti i dubbi.

Emily Frini

Sono convinta che avrei potuto fare di più per quanto riguarda la mia parte all'interno del progetto, è stato un peccato dover droppare parte del mio lavoro. I giochi a turni non mi piacciono particolarmente, avrei sicuramente preferito una cosa più dinamica, soprattutto per poter lavorare maggiormente sull'utilizzo degli *sprite*, creazioni di mappe e l'interfaccia giocatore in generale.

Edoardo Montanari

Essendo uno dei progetti personalmente più complessi a cui ho partecipato, mi ritengo soddisfatto del lavoro svolto, nonostante le iniziali difficoltà. Creare un' intelligenza artificiale mi ha molto appassionato a livello progettuale e ha rappresentato per me una sfida per la mia crescita personale. Penso che rimarrà un' esperienza lavorativa che mi potrà dare spunti creativi per lavori futuri.

Olivia Nguemo

Questa è stata la mia seconda esperienza all'interno di un Team di lavoro, e mi ritengo molto soddisfatta del risultato ottenuto. Ho imparato a confrontarmi con persone con pareri diversi dal mio e questo credo sia uno degli aspetti più rilevanti della mia esperienza col mio gruppo. Essendo la mia prima esperienza con un videogioco, i miei compagni hanno condiviso le loro conoscenze con me. è stato un piacere lavorare con questo team.

4.2 Difficoltà incontrate e commenti per i docenti

Alice Conti

Per me è stato particolarmente difficile l'utilizzo di Git, siccome utilizzarlo con altre persone non era stato troppo approfondito a lezione e alcuni comandi non sono banali e immediatamenti intuitivi.

Emily Frini

La difficoltà principale è stata settare Gradle perchè a lezione non è stato fatto vedere come fare esattamente; anche capire come JavaFX funzionasse è stato particolarmente complicato, in particolare come il suo Thread interno lavora. Soltanto verso la fine e dopo ore di tentativi e lettura di guide sono riuscita a capirne le base e a lavorarci più velocemente.

Edoardo Montanari

Una delle difficoltà principali incontrate durante la realizzazione del progetto, oltre a quelle già elencate e spiegate nei capitoli precedenti, è stata la complessità di utilizzo di git, in quanto non sempre riusciva ad eseguire e a caricare correttamente i miei file del progetto, come anche le operazioni che dichiaravo. Ciò ha messo a rischio ben più di una volta perdere parte del mio lavoro svolto online (e anche in locale) oppure quello dei miei compagni di progetto.

Olivia Nguemo

All'inizio del progetto, la prima difficoltà è stata di capire come funziona javaFX che non è stato molto dettagliato a lezione. E poi, quando è venuto il momento di fare il merge delle singole parti del lavoro, non è stato banale risistemare tutto prendendo in considerazione le parti degli altri. Alla fine, ho imparato ad usare meglio GitHub per il workflow in gruppo sopratutto evitare i merge conflict.

Appendice A

Guida utente

All'avvio dell'applicazione viene mostrata un'interfaccia con il menù iniziale di Figura A.1, da cui si può scegliere:

- **Play**: permette di iniziare una nuova partita
- **How to play**: apre una finestra con le istruzioni di gioco
- **Exit**: consente di chiudere l'applicazione

Figura A.1: Finestra del menù di gioco

Scopo del gioco è sopravvivere a infiniti nemici usando strategicamente l'ambiente e le proprie abilità. Durante il turno possono essere eseguite 3 azioni, tra cui muoversi, attaccare o usare le abilità. Per muoversi si possono utilizzare

le frecce presenti nell'interfaccia di gioco. Per attaccare un nemico è sufficiente avvicinarsi e muoversi verso la sua direzione.

Le funzionalità delle abilità si possono leggere passando il mouse sopra i bottoni della colonna a sinistra.

Una volta che tutti i nemici sono stati uccisi verrà ricreata una nuova arena.

- Muoversi costa un'azione.
- Superare una pozzanghera costa due azioni.
- Andare contro una roccia fa perdere punti vita.
- Curarsi con l'Elisir della vita costa 25 di oro.

Figura A.2: Finestra di gioco

Appendice B

Esercitazioni di laboratorio

Olivia Nguemo

https://github.com/olivia237/OOP-lab-05.git laboratorio 05

https://github.com/olivia237/OOP-Lab06.git laboratorio 06

https://github.com/olivia237/OOP-Lab07.git laboratorio 07

https://github.com/olivia237/OOP-Lab08.git laboratorio 08

https://github.com/olivia237/OOP2021-Lab09.git laboratorio 09

https://github.com/olivia237/OOP2021-Lab10.git laboratorio 10