Progetto di Programmazione ad Oggetti
"The Exiled"

Luca Casadei, Francesco Pazzaglia, Marco Magnani, Manuel Baldoni

10 febbraio 2024

Indice

Capitolo 1

Analisi

1.1 Requisiti

L'applicazione è un gioco che presenta un personaggio controllato dal giocatore con la possibilità di muoversi nella mappa in 4 direzioni e di combattere contro dei nemici utilizzando magie elementali. I nemici sconfitti potrebbero rilasciare delle cure o dei potenziamenti che favoriscono il giocatore. Per concludere il gioco, il giocatore deve entrare in possesso di 4 cristalli che vengono consegnati una volta sconfitti i 4 boss del gioco che sono nemici più difficili da sconfiggere.

Requisiti funzionali

- Movimento del giocatore.
- Movimento dei nemici.
- Presenza di oggetti ottenibili dal giocatore (Cure, potenziamenti e cristalli).
- Terminazione del gioco una volta raccolti 4 cristalli o se il giocatore viene sconfitto.
- Posizionamento e distribuzione degli oggetti.
- Possibilità di fare battaglie tra giocatore e nemici.
- Nemici più forti (boss) che se sconfitti rilasciano i cristalli per terminare il gioco.

- Possibilità di utilizzo di magie in battaglia di diverso tipo (Fuoco, Acqua, Fulmine, Erba).
- Aumento di livello tramite guadagno di esperienza sconfiggendo i nemici.

Requisiti non funzionali

- Il gioco deve essere multipiattaforma.
- *TODO*

1.2 Analisi e modello del dominio

All'inizio al giocatore viene chiesto di che tipo elementale sarà il personaggio da controllare, in base a questa scelta gli verrà quindi assegnata una mossa di base di quel tipo e avrà le mosse di quel tipo potenziate fino alla fine del gioco.
Il giocatore può muoversi di una cella alla volta in una mappa a griglia in 4 direzioni (Nord, Est, Sud, Ovest) e una battaglia con i nemici (anch'essi di un certo tipo elementale) inizia quando la cella in cui è il giocatore combacia con quella di un nemico.
Per affrontare un nemico in battaglia il giocatore ha a disposizione massimo 4 mosse (non per forza dello stesso tipo del giocatore). Il combattimento avviene a turni alternati, il primo è il giocatore, poi il nemico e così via finché uno dei due viene sconfitto, se è il giocatore, il gioco termina.
Allo sconfiggere dei nemici viene conferita al giocatore una certa quantità di esperienza che serve ad aumentare di livello e un oggetto casuale, l'oggetto viene salvato nell'inventario del giocatore e potrà essere di diverso tipo, es. oggetto curativo(ripristina un tot di vita), booster di statistiche(per esempio aumenta l'attacco di un certo tipo di mosse) ecc.. L'aumento di livello comporta un incremento generale delle statistiche ovvero attacco, difesa e vita di un valore costante. Ogni 5 livelli verrà presentata al giocatore la possibilità di imparare una nuova mossa di un tipo casuale, se il giocatore ha già 4 mosse potrà decidere di scambiare quella nuova con una di quelle che conosce già. All'aumentare di livello sarà richiesta sempre più esperienza per passare a quello successivo.
Ci sono diverse classi di nemico, alcuni più deboli e altri più forti, come i boss, che se sconfitti consegnano uno dei 4 oggetti necessari per concludere il gioco (i cristalli).

1.2.1 Tipi elementali

I seguenti sono i tipi elementali presenti, ognuno è efficace rispetto ad un altro, quindi quando vengono usate mosse di un certo tipo che risulta essere efficace rispetto al tipo di un nemico, si avranno dei moltiplicatori del danno. Lo stesso vale per i nemici che usano mosse di un tipo efficace rispetto a quello scelto dal giocatore all'inizio.

Ad esempio, nel seguente schema si vede che Fulmine è efficace contro Acqua.

Fulmine	$\rightarrow$	Acqua
Erba	$\rightarrow$	Fulmine
Acqua	$\rightarrow$	Fuoco
Fuoco	$\rightarrow$	Erba

1.2.2 UML

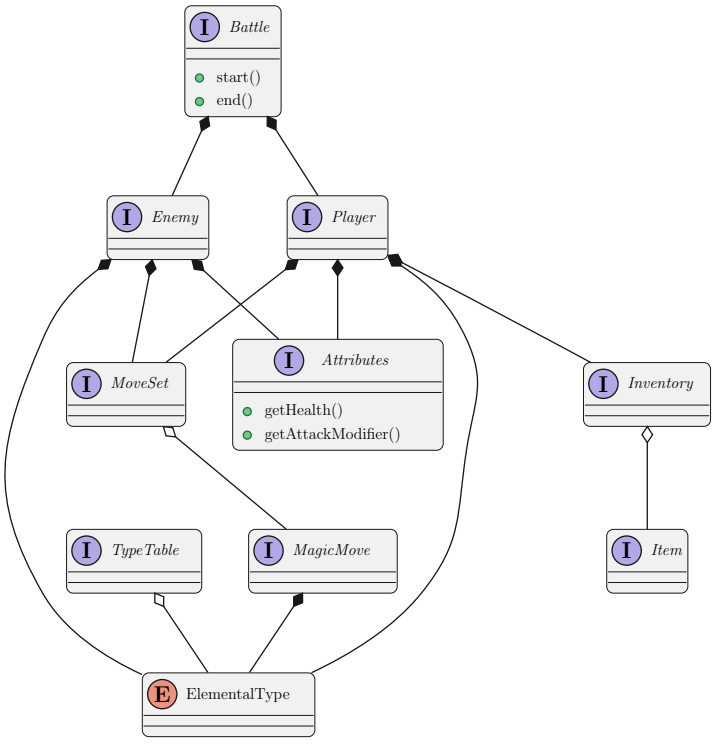

Figura 1.1: Schema UML del dominio.

Capitolo 2

Design

Qui entriamo nel dettaglio della struttura ad alto livello del gioco.

2.1 Architettura

2.2 Design dettagliato

Luca Casadei

Francesco Pazzaglia

La mia parte di progetto si concentrava sull'implementazione dettagliata del giocatore, comprendendo la definizione delle sue caratteristiche quali attributi(come la vita, l'esperienza, la relativa classe...) e le sue meccaniche in gioco. Quindi per facilitare la comunicazione tra il model del giocatore (*Player*) e la view, ho introdotto un controller quindi, utilizzando il pattern *MVC*. Mi sono occupato della gestione del movimento del giocatore e delle altre entità (i nemici) all'interno della mappa di gioco. Pertanto, anche in questo caso è stato fatto uso di controller per consentire la visualizzazione dei nemici in movimento sulla mappa. Infine, ho implementato la gestione delle condizioni di Game Over, che includevano la definizione delle regole e dei possibili eventi che portavano al termine del gioco, nonché la gestione delle rispettive schermate, come la chiusura del gioco o la possibilità di far ripartire una nuova partita.

Player

Nell'ambito del progetto del gioco, ho osservato una serie di caratteristiche comuni tra le diverse entità presenti nel gioco. Le caratteristiche di base del

giocatore sono condivise anche dai vari nemici. Pertanto, in collaborazione con *Luca Casadei*, responsabile della parte relativa ai nemici, abbiamo deciso di introdurre nel model un'interfaccia denominata *GameCharacter*. Questa interfaccia contiene i metodi principali che definiscono le caratteristiche fondamentali comuni a tutte le entità nel gioco, come la vita, la posizione, il movimento e gli attributi. In aggiunta all'interfaccia, è stata implementata la classe astratta *GameCharacterImpl*, che concretizza l'interfaccia *GameCharacter*. Questa classe è stata progettata in modo che il giocatore, e di conseguenza anche i nemici, possano estenderla e acquisire le caratteristiche comuni menzionate in precedenza. Questa architettura ci ha consentito di gestire in modo efficiente le proprietà condivise tra il giocatore e i nemici, garantendo una maggiore coerenza e facilità di estensione del codice senza ripetizioni.

Nella gestione del giocatore, ho introdotto un'interfaccia denominata *Player*, che estende l'interfaccia preesistente *GameCharacter*. Successivamente, ho realizzato l'implementazione di questa interfaccia attraverso una classe concreta chiamata *PlayerImpl*. Tale procedura mi ha consentito di definire un comportamento specifico, mantenendo al contempo la flessibilità per estendere e personalizzare le funzionalità di base fornite dall'interfaccia *GameCharacter*.

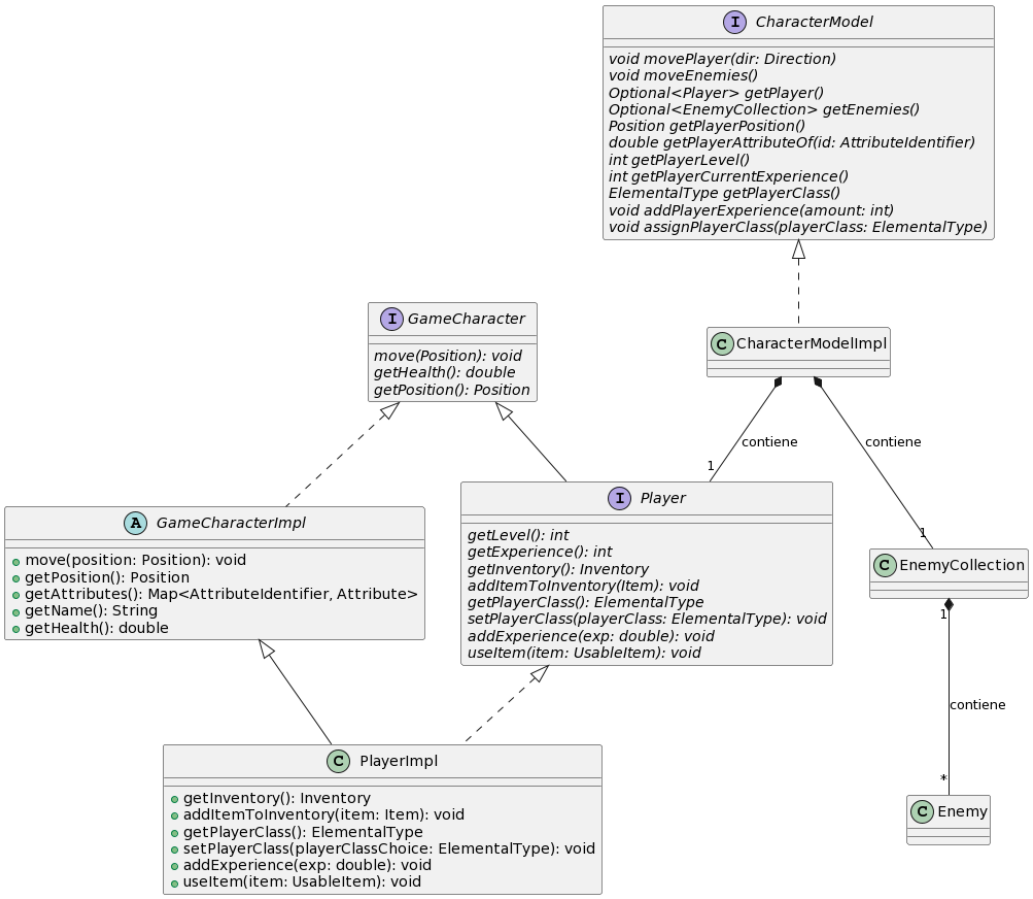

Figura 2.1: Schema UML del Model del Player e del GameCharacter

La struttura del Model è stata progettata seguendo il Composite Pattern. In particolare, si parte dall'interfaccia *CharacterModel* con la relativa classe *CharacterModelImpl* è stata configurata per contenere un elemento dell'interfaccia *Player* e una collezione di nemici. Questa configurazione organizza la struttura del modello come un albero, dove *CharacterModelImpl* funge da componente composito che contiene sia il giocatore che la collezione di nemici al suo interno.

Nel contesto della View sia il giocatore che i nemici sono stati gestiti attraverso una singola classe chiamata *CharacterView*. Questa classe è stata progettata per utilizzare immagini al fine di rappresentare graficamente le entità del gioco. In particolare, sono state gestite delle animazioni mediante l'utilizzo di due immagini per ciascuna direzione (SU, GIÙ, DX e SX). Questo approccio è stato adottato per migliorare l'aspetto visivo del gioco e rendere più coinvolgente l'esperienza di gioco.

Per garantire la comunicazione tra il Model e la View del *Player* è stato introdotto un Controller, sotto forma di intefaccia, denominato *CharacterController*. La relativa classe che la implementa invece è stata chiamata *CharacterControllerImpl*, dove, al suo interno sono stati definiti i metodi per il controllo del *Player*, tra cui *movePlayer(Direction)*, *assignPlayerClass(ElementalType)*, *addPlayerExperience(int)*, nonché i relativi metodi Getter necessari per l'interfaccia grafica.

Gestione del movimento

Nell'ambito dell'implementazione del movimento delle entità all'interno della mappa di gioco, ho adottato due distinzioni principali. Per il movimento del giocatore, ho progettato un sistema basato sull'input dell'utente tramite la tastiera(tramite i tasti W,A,S,D ci si può muovere nelle 4 direzioni). D'altra parte, per i nemici, ho sviluppato un meccanismo autonomo di movimento. In pratica, i nemici si spostano all'interno di un'area definita dalla loro classe, e quando il giocatore si avvicina entro un determinato raggio, i nemici iniziano a inseguirlo. Questo comportamento è stato implementato attraverso due metodi: uno per calcolare la distanza tra il giocatore e un nemico (il metodo *calculateDistance*), e un altro per determinare la direzione in cui il nemico deve muoversi per inseguire il giocatore (il metodo *calculateChaseDirection*). Questo approccio consente un comportamento dinamico e coinvolgente per i nemici, rendendo il gioco più interessante e sfidante per il giocatore. Considerando la decisione di non impiegare i thread nel nostro progetto, i nemici si sposteranno esclusivamente in risposta all'input dell'utente. Di conseguenza, i nemici si muoveranno solamente quando l'utente eseguirà uno spostamento sulla mappa di gioco.

In particolare, ho utilizzato il *CharacterModel* per definire il comportamento del movimento sia per il personaggio che per i nemici. Successivamente, all'interno del *CharacterController*, ho implementato un metodo di movimento che ha richiamato i metodi precedentemente definiti nel Model. Questa scelta è stata fatta per mantenere la separazione delle responsabilità e organizzare il movimento all'interno del pattern *MVC*. Inoltre, nella classe *GameKeyListener* della View, ho gestito l'input dell'utente associando i movimenti alle direzioni corrispondenti ai tasti premuti (ad esempio, W per su, S per giù, A per sinistra e D per destra)

Marco Magnani

Manuel Baldoni

Capitolo 3

Sviluppo

3.1 Testing automatizzato

3.2 Note di sviluppo

Luca Casadei

Francesco Pazzaglia

Marco Magnani

Manuel Baldoni

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.2 Difficoltà incontrate e commenti per i docenti

Appendice A

Guida utente

Appendice B

Esercitazioni di laboratorio

Elenco dei Permalink delle consegne nel forum studenti del modulo di laboratorio.

Consegne

B.0.1 luca.casadei27@studio.unibo.it

- Laboratorio 06: `https://virtuale.unibo.it/mod/forum/discuss.php?d=146511#p208401`
- Laboratorio 07: `https://virtuale.unibo.it/mod/forum/discuss.php?d=147598#p209323`
- Laboratorio 08: `https://virtuale.unibo.it/mod/forum/discuss.php?d=148025#p209782`
- Laboratorio 09: `https://virtuale.unibo.it/mod/forum/discuss.php?d=149231#p211336`
- Laboratorio 10: `https://virtuale.unibo.it/mod/forum/discuss.php?d=150252#p212475`