Relazione per il progetto del corso di

“Programmazione ad Oggetti”

Buda Francesco
Ceredi Tommaso
Maglia Danilo
Severi Tommaso

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il software, prodotto come progetto per il corso di programmazione ad oggetti, si propone di realizzare un videogioco chiamato Unreal Defense. Una personale reinterpretazione in stile fantasy del paradigma Tower Defense, nel quale il giocatore assume il ruolo di difensore di una zona e deve impedire l'avanzamento dei nemici lungo un percorso, attraverso il piazzamento di torri o altri strumenti difensivi.

Requisiti funzionali

- il software che presentiamo darà all'utente la possibilità di cimentarsi in una mappa di gioco costituita da un percorso, nel quale, a diverse ondate, passeranno orde composte da due tipologie di nemici: gli orchi (lenti ma resistenti) e i goblin (veloci ma deboli).
- il giocatore sarà in grado di piazzare, in punti prestabiliti, 2 tipologie di torri difensive: una colpirà un singolo nemico alla volta, l'altra colpirà diversi nemici con un danno ad area.
- Oltre alle torri difensive sarà possibile utilizzare, per proteggere la propria zona, due incantesimi: uno provoca danno ad area e l'altro rallenta i nemici nel suo raggio per alcuni secondi. Questi incantesimi saranno piazzabili in qualsiasi punto del percorso dopo aver atteso un certo lasso di tempo per il caricamento.
- Il gioco dovrà gestire un semplice sistema di economia attraverso il quale sarà possibile guadagnare valuta eliminando i nemici, per poi spenderla per acquistare nuove difese.
- Verrà gestita la condizione di vittoria e sconfitta del giocatore attraverso un sistema di "vite" che si consumeranno a mano a mano che i nemici riusciranno ad invadere la zona alla fine del percorso. Se il giocatore riuscirà a resistere fino all'ultima ondata di nemici il livello si considererà superato.

Requisiti non funzionali

- Il software dovrà essere ben calibrato dal punto di vista della difficoltà di gioco, facendo in modo che sia necessario un minimo di strategia per riuscire a vincere il livello, rendendo così l'esperienza soddisfacente per il giocatore.
- Il software dovrà essere in grado di eseguire in modo fluido su qualsiasi macchina, senza richiedere hardware particolarmente potenti.
- Il software dovrà presentare un livello di estetica piacevole alla vista e chiaro per aumentare l'immedesimazione del giocatore.

1.2 Analisi e modello del dominio

Il dominio applicativo del gioco Unreal defense si compone di due classi principali: Il mondo di gioco e il giocatore.

Il mondo di gioco è l'elemento collante del dominio, esso contiene il percorso che i nemici dovranno percorrere e un insieme di torri che è possibile piazzare sul percorso. Le torri sono caratterizzate da un loro raggio d'azione e un peculiare pattern di attacco, per il quale necessitano di mantenere l'informazione legata al nemico che costituisce il loro target. Inoltre, il mondo accede ai nemici, separati in ondate ed orde, e controlla il loro ciclo di vita. Quando un nemico attraversa l'area difesa da una torre esso potrà essere colpito e perdere parte della sua vita, se il nemico termina i propri punti vita, viene rimosso dal percorso e il giocatore guadagna delle monete, immagazzinate dal mondo all'interno della banca. Player, costituisce l'entità attiva, e interagisce con il mondo per conto dell'utente. Il player può costruire torri, spendendo monete, e può lanciare incantesimi sul mondo, questi ultimi immagazzinati dal player stesso.

Gli elementi costitutivi del problema sono sintetizzati in Figura 1.1.

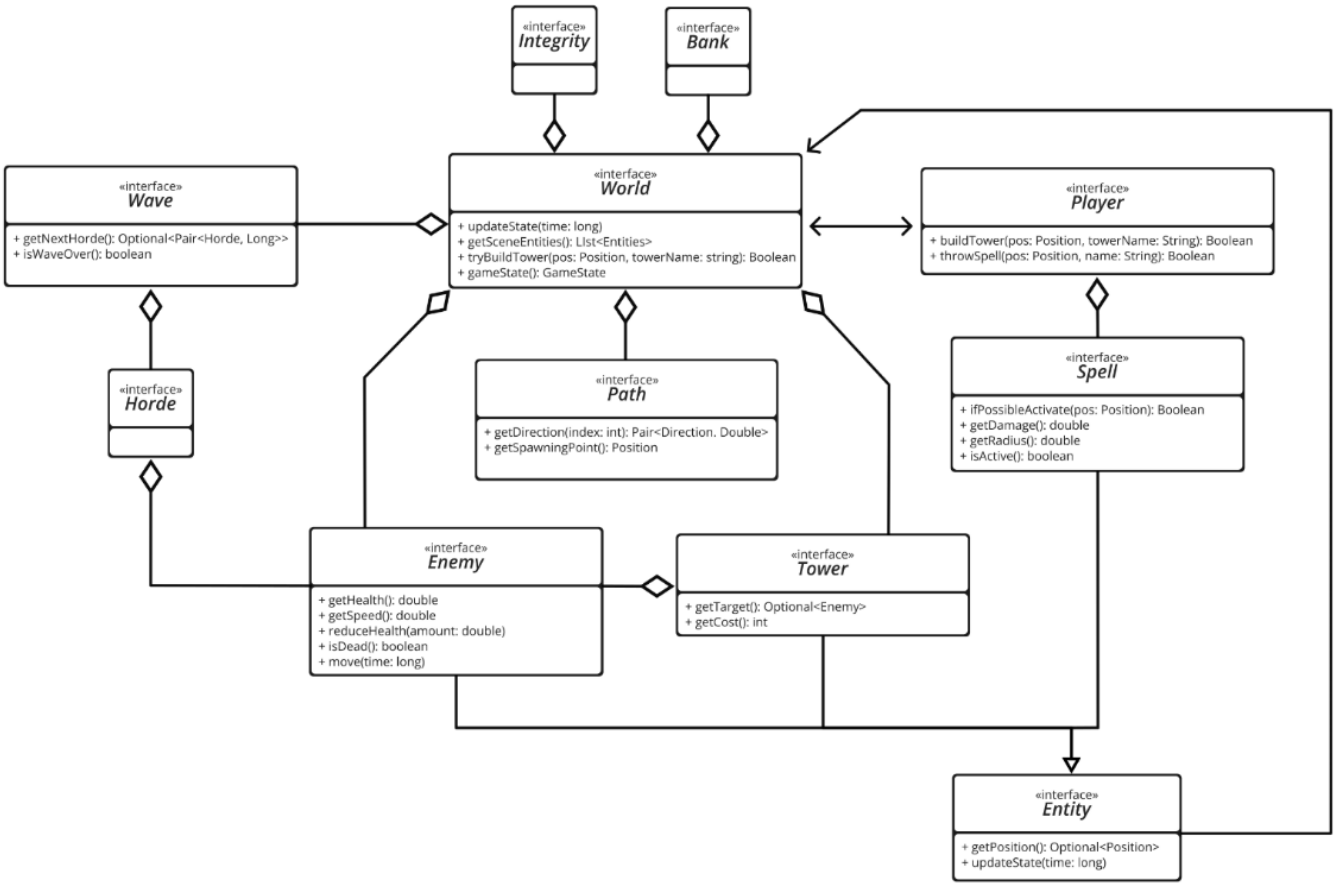

Figura 1.1

La sfida principale che dovremo affrontare è la corretta interazione delle torri con i nemici sul percorso, ciò consiste in un efficace coordinamento tra i vari elementi coinvolti.

Il requisito non funzionale riguardante la calibrazione della difficoltà di gioco richiede uno studio di game design non superficiale che non sarà completamente possibile effettuare nel monte ore previsto. Il raffinamento di questa parte potrebbe essere oggetto di lavori futuri.

Capitolo 2

Design

2.1 Architettura

L'architettura di Unreal Defense adotta il pattern architetturale MVC (Model-View-Controller). GameEngine è l'interfaccia che modella il controller, il quale si occupa di aggiornare costantemente l'applicazione, indipendentemente dagli input provenienti dalla view, rappresentata dall'omonima interfaccia. Al momento della ricezione di un input, il controller lo elabora e lo traduce in un'operazione da eseguire sul model. Le interfacce che costituiscono l'entry point per il model sono Player e World: la prima rappresenta la parte attiva, attraverso la quale il controller notifica l'input al model, mentre la seconda viene interpellata solamente per aggiornare lo stato di gioco e per ricevere informazioni in merito.

inoltre, la view comunica con il model al fine di ottenere le informazioni riguardo le entità attive nel mondo e presentarle all'utente in forma grafica. È da sottolineare che i tre elementi dell'architettura (model, view e controller) sono totalmente indipendenti l'uno dall'altro. ciò significa che è possibile cambiare completamente la resa grafica senza influire su model e controller, e che eventuali modifiche alla logica interna del model non avrebbero impatto sul resto dell'applicazione. La variazione del modo in cui viene ricevuto l'input è supportata, anche se, l'introduzione di un nuovo tipo di input richiederebbe alcune modifiche, principalmente al controller.

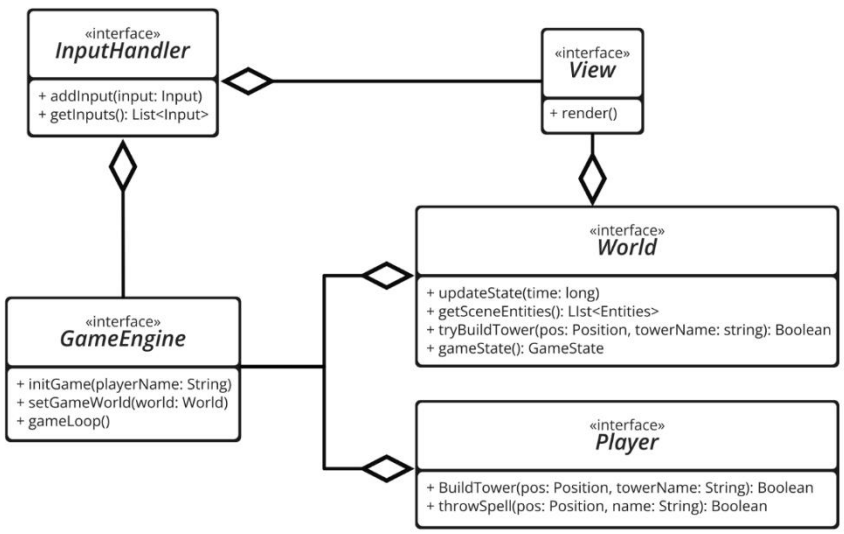

Figura 2.1

2.2 Design dettagliato

Buda Francesco

Il ruolo del mondo di gioco

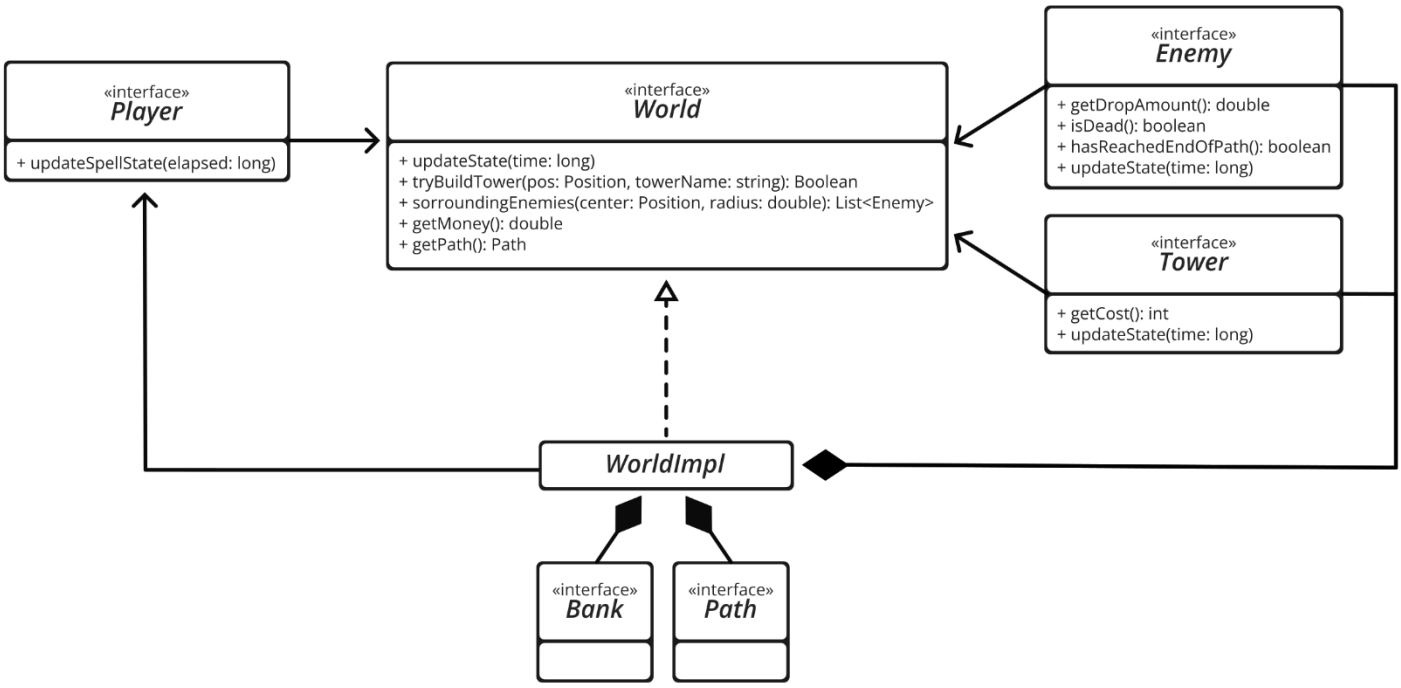

Figura 2.2

Problema

una delle prime questioni venutesi a delineare durante la fase di analisi è stata la gestione delle dipendenze tra le varie classi del progetto. ad esempio le torri, che devono accedere al percorso per identificare le entità presenti su di esso e ai nemici stessi per poterli attaccare, il player, che deve accedere ai nemici per poter lanciare gli incantesimi e alla banca per poter comprare e costruire nuove torri, i nemici, che devono accedere al percorso per cambiare la direzione del loro avanzamento, ecc... . Ci si è subito resi conto che una più intelligente organizzazione fosse necessaria per evitare i grovigli di dipendenze poco chiari e inefficaci che si stavano delineando.

Soluzione

La soluzione che abbiamo adottato è stata la creazione di una classe World che implementasse il pattern Mediator. In questo modo il mondo coordina i rapporti tra tutte le componenti e limita notevolmente le dipendenze caotiche. Ogni elemento attivo, in questo modo, passa dal mondo per interagire con gli altri. la banca e il percorso rimangono incapsulati nel mondo senza essere mai acceduti direttamente. Inoltre, il mondo aggiorna sé stesso e notifica tutte le entità che coordina dei cambiamenti avvenuti, attraverso il suo metodo updateState. Seguendo una strategia simile a quella descritta dal pattern Observer, dove nel nostro caso il mondo costituisce l'observable e le entity sono gli observer.

La banca e l'integrità del castello come componenti a sé stanti

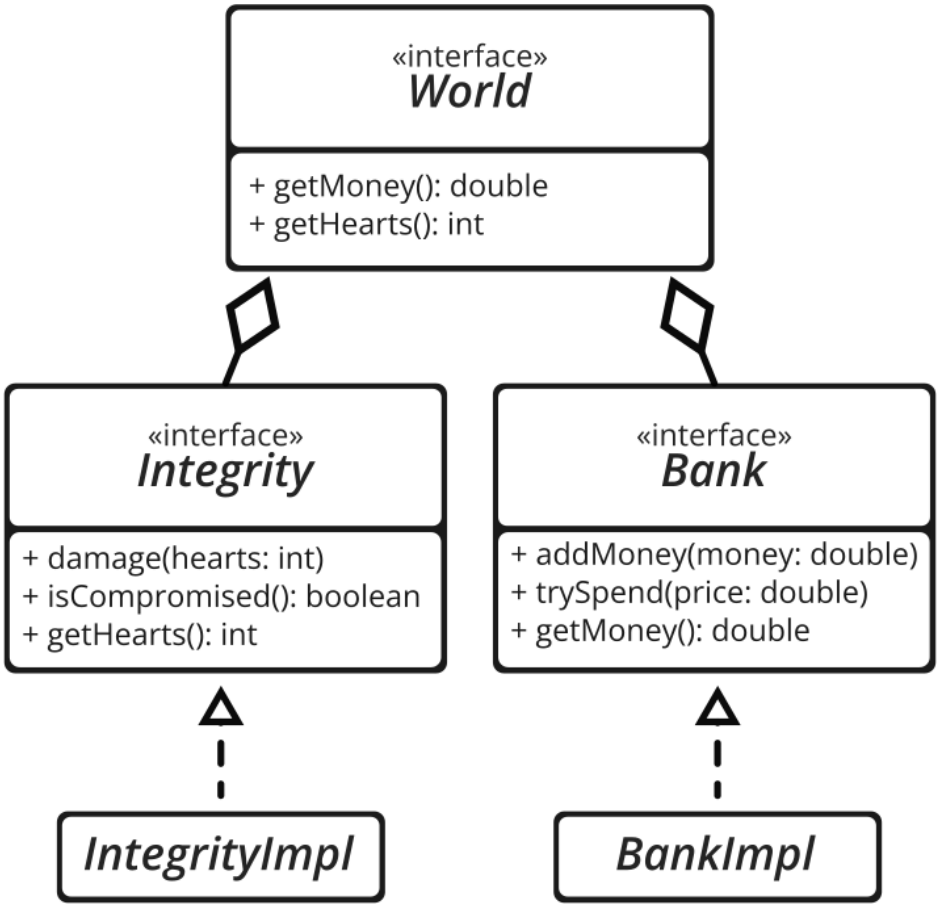

Figura 2.3

Problema

Un'altra questione che si è posta durante lo sviluppo del progetto è stata la scelta del tipo di implementazione per la banca e per l'integrità del castello. Inizialmente si era pensato di aggiungere semplicemente dei campi nel mondo per tenere traccia delle monete e dei cuori, ma poi è sorta la possibilità di scegliere un'implementazione diversa delegando ad un'oggetto esterno queste due funzioni.

Soluzione

Valutati pro e contro delle due strade, la scelta è ricaduta sulla seconda. Infatti questo tipo di implementazione porta con se alcuni vantaggi: innanzitutto alleggerisce la classe WorldImpl di due responsabilità marginali non direttamente legate al ruolo scelto per essa, favorendo il single responsibility principle. In più permette di implementare a piacimento le due interfacce Bank e Integrity. Nel progetto viene fornita una implementazione standard ma nulla vieta in futuro di estendere il set con nuove implementazioni.

Costruzione del mondo di gioco

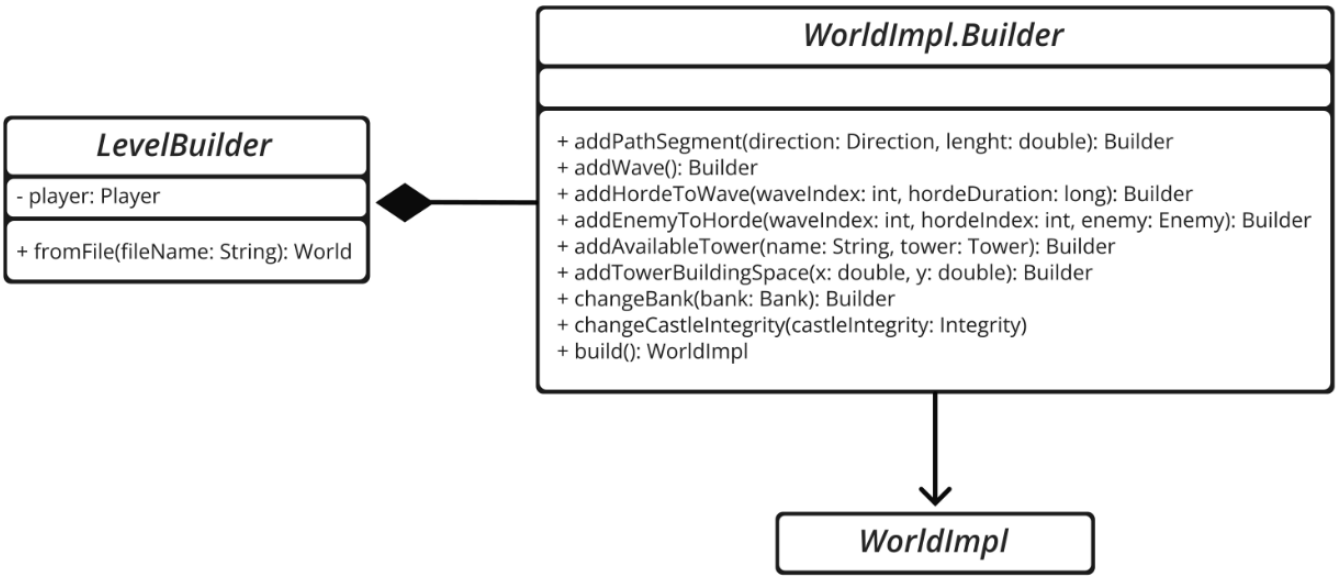

Figura 2.4

Problema

Durante lo sviluppo del progetto ci siamo presto resi conto di quanto il mondo di gioco fosse un'oggetto moderatamente più sofisticato rispetto agli altri, e risultava complesso, non che poco elegante, inizializzarlo per impostare il livello del gioco o per realizzare dei test, automatizzati e non.

Soluzione

La soluzione che abbiamo scelto è stata l'utilizzo del pattern Builder. Abbiamo quindi realizzato un WorldBuilder che non solo permette di costruire in modo agevole un mondo di gioco, ma permette di farlo in modo completamente personalizzato, potendo combinare ad esempio infinite varianti di percorso con infinite varianti di orde, ondate e nemici. Abbiamo inoltre messo a disposizione la classe LevelBuilder che svolge il ruolo del Director e possiede un metodo per costruire mondi preimpostati eliminando all'occorrenza la necessità di doversi interfacciare direttamente con il builder.

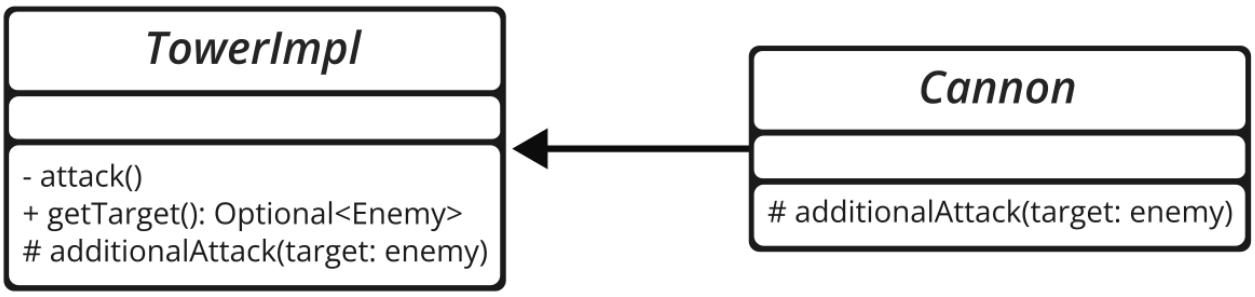

Figura 2.5

Sistema di difesa ergonomico

Problema

Nel gioco sviluppato in Java, tutte le torri del gioco hanno lo stesso attacco base. Tuttavia, una particolare torre di tipo "Cannon" ha un attacco aggiuntivo da eseguire durante l'attacco base.

Soluzione

Per risolvere questo problema si è utilizzato il pattern "Template Method". In particolare, si è creato un metodo astratto "additionalAttack" nella classe "TowerImpl", che verrà implementato dalle classi figlie per definire l'attacco aggiuntivo specifico per ogni tipo di torre.
La classe "TowerImpl" definisce un algoritmo di base per l'attacco delle torri tramite il metodo "attack()", che viene invocato quando una torre ha un nemico nel raggio di attacco. All'interno di "attack()", viene controllato se ci sono nemici nell'area di attacco della torre e viene selezionato il primo nemico nell'elenco per attaccare. Successivamente, viene invocato il metodo " additionalAttack()" specifico della classe figlia, che esegue l'attacco aggiuntivo.
In questo modo, è possibile utilizzare la classe "TowerImpl" come classe astratta per definire le torri standard e estenderla nelle classi figlie per definire le torri speciali come la torre "Cannon", che sovrascrive il metodo "additionalAttack()" per eseguire l'attacco aggiuntivo specifico.
In questo modo è possibile in qualsiasi momento aggiungere nuove torri ed implementare nuove caratteristiche o funzionalità utilizzando parti di codice già in uso.

Maglia Danilo

Entità

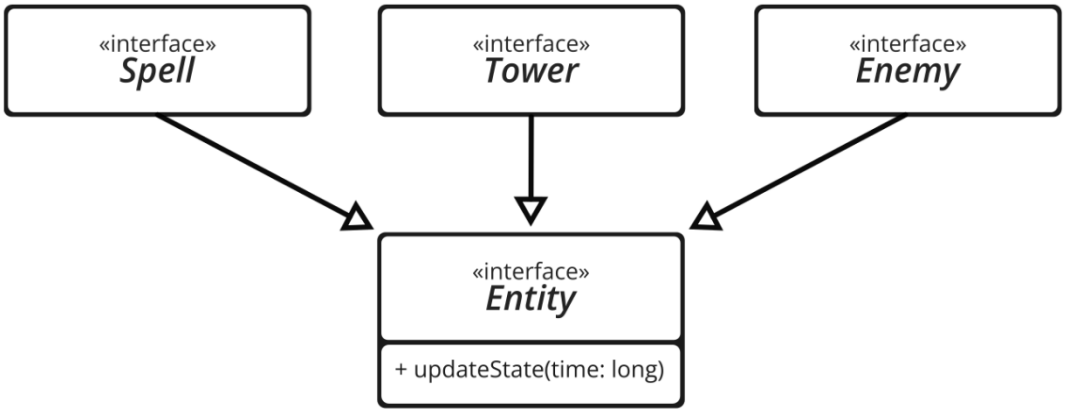

Figura 2.6

Problema

Fin dall'inizio dell'analisi, è emerso che molte classi del gioco presentavano elementi in comune, come la posizione, il nome, il riferimento al mondo di gioco e la necessità di essere aggiornate in base alla logica del gioco con l'avanzare del tempo. Tuttavia, ciascuna di queste classi era stata implementata separatamente, senza alcuna forma di riutilizzo del codice. Era dunque necessario trovare un modo per evitare la duplicazione di codice e semplificare la gestione delle entità.

Soluzione

Per risolvere il problema, è stata creata una gerarchia di classi che si basa sull'ereditarietà e sul polimorfismo.

L'origine della gerarchia corrisponde all'interfaccia "Entity" che definisce il comportamento comune di tutti gli elementi di gioco considerati entità, che nel nostro caso sono i nemici "Enemy", gli incantesimi "Spell", e le torri difensive "Tower". In particolare, viene definito il metodo "updateState" che verrà implementato singolarmente dalle diverse entità a seconda delle loro esigenze. Inoltre, grazie all'ereditarietà abbiamo ulteriormente ridotto la ripetizione di codice nell'implementazione, raggruppando i comportamenti comuni delle entità "difensive" in una classe apposita.

Grazie a questa gerarchia il mondo può interfacciarsi con tutte le entità senza preoccuparsi della specifica implementazione di ognuna di esse e aggiornarle tramite il metodo "updateState". Questo si può ricondurre al pattern "Strategy", dato che ogni classe che implementa Entity definisce a modo suo il metodo updateState, ma al mondo di gioco non interessa come esso viene implementato.

La soluzione proposta si basa quindi sul principio di ereditarietà e polimorfismo, due concetti fondamentali della programmazione orientata agli oggetti, che permettono di scrivere codice più efficiente e mantenibile.

Orde e Ondate

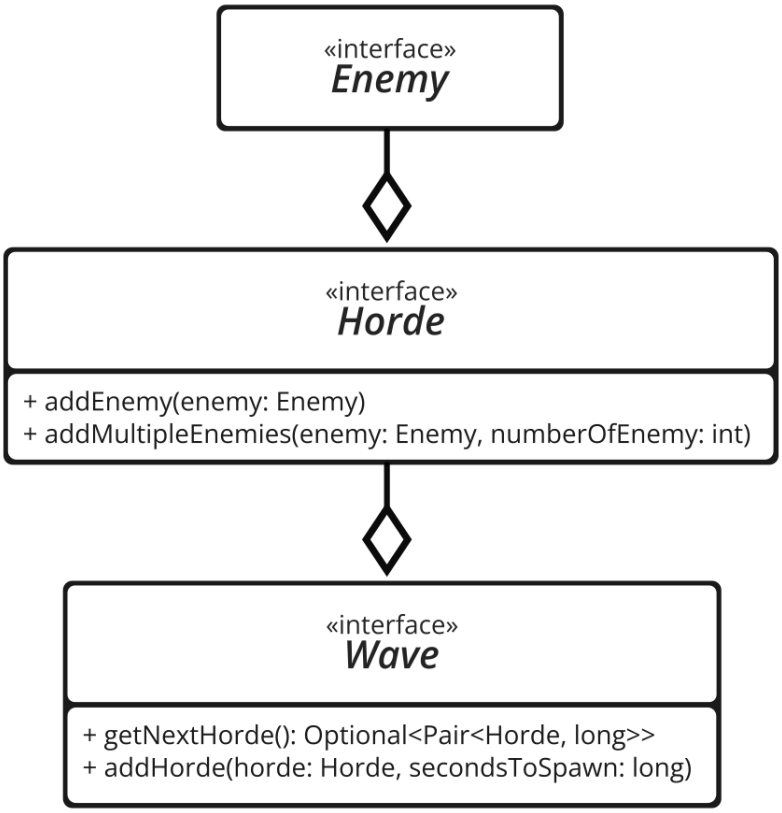

Figura 2.7

Problema

Un problema sorto è stato quello di trovare un modo per tenere traccia di tutti i nemici di ogni mondo di gioco. In modo tale che potessero percorrere la mappa in ordine e suddivisi in diverse ondate.

Soluzione

Per risolvere questo problema, è stata creata una soluzione basata su due interfacce: "Horde" e "Wave". Ogni orda contiene dei nemici, che possono essere di diversi tipi. L'interfaccia inolte fornisce dei metodi per semplificarne la gestione, come l'aggiunta di singoli nemici o nemici multipli.

L'ondata invece, tiene traccia di tutte le orde e quanto tempo deve passare prima che il mondo debba far partire la prossima orda.

Il mondo di gioco, quindi, non deve fare altro che tenere traccia delle varie ondate. In questo modo, può sapere quali nemici devono ancora arrivare e in quanto tempo, semplificando notevolmente la gestione dei nemici all'interno del gioco.

L'interfaccia Wave, inoltre, offre vari metodi per la gestione delle orde, come metodi per l'aggiunta e per controllare se ci sono altre orde oppure se l'ondata è finita.

In sintesi, la soluzione basata sull'utilizzo delle interfacce Horde e Wave consente di tenere traccia dei nemici che devono arrivare in modo semplice ed efficiente, semplificando la gestione dei nemici all'interno del mondo di gioco.

Nemici e Percorso

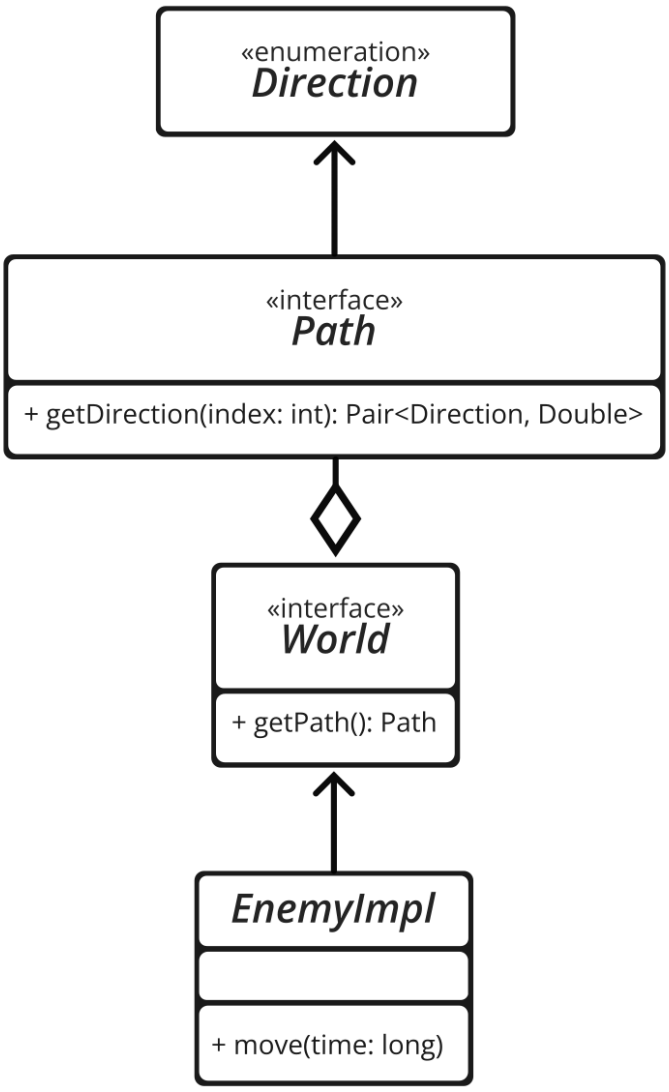

Figura 2.8

Problema

Il problema che ci si pone è come far muovere i nemici all'interno del mondo di gioco seguendo un percorso prestabilito. La soluzione più immediata che viene in mente è quella di creare per ogni nemico una lista di direzioni e lunghezze che deve seguire. Tuttavia, questa soluzione presenta alcuni problemi. In primo luogo, ogni nemico avrebbe una lista di direzioni e lunghezze differente, il che richiederebbe una notevole quantità di memoria. Inoltre, se si volesse modificare il percorso, sarebbe necessario modificare la lista per ogni nemico.

Soluzione

Una soluzione elegante consiste nell'utilizzare un'interfaccia denominata "Path". Questa interfaccia offre dei metodi che facilitano la creazione di percorsi prestabili, come la possibilità di aggiungere direzione e lunghezze. In questo modo, si può creare un percorso una volta sola e renderlo disponibile per tutti i nemici.

Il Path è gestito dal mondo di gioco, quindi i nemici possono semplicemente richiederlo dal mondo e tenersi un indice che indica a quale punto del percorso si trovano e la quantità di strada che devono ancora percorrere prima di chiedere una nuova direzione al percorso. In questo modo, ogni nemico può seguire lo stesso percorso senza dover memorizzare informazioni inutili.

Inoltre, questa soluzione consente di apportare modifiche al percorso in modo semplice ed efficiente. Se si volesse modificare il percorso, sarebbe sufficiente modificarlo nel mondo di gioco e tutti i nemici in seguito richiederanno automaticamente la versione aggiornata del percorso.

In sintesi, la soluzione basata sull'utilizzo dell'interfaccia "Path" è più efficiente, elegante e facile da gestire rispetto all'utilizzo di liste di direzioni e lunghezze specifiche per ogni nemico.

Severi Tommaso

Giocatore e Incantesimi vari

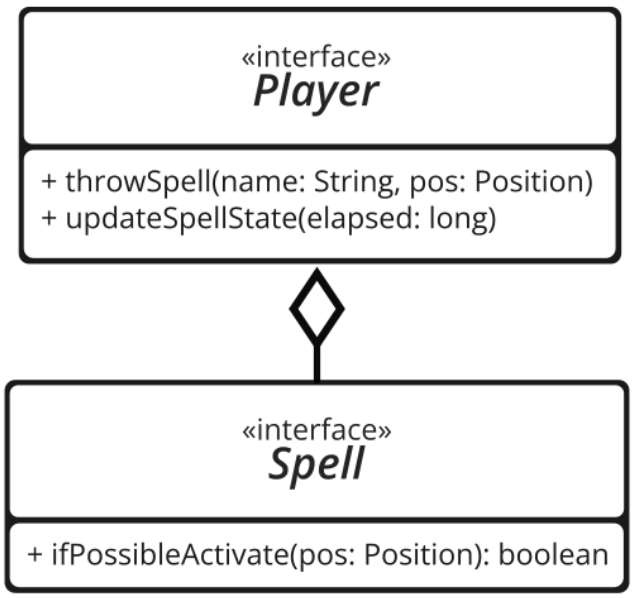

Figura 2.9

Problema

L'applicazione richiede che l'oggetto che rappresenta il giocatore possa contenere incantesimi di diversi tipi e quantità senza che ciò influisca sull'implementazione interna del giocatore stesso.

Soluzione

Per risolvere il problema descritto, si può utilizzare il pattern "Strategy". In questo modo, è possibile individuare l'incantesimo desiderato attraverso un nome identificativo e richiamare il metodo "ifPossibleActivate" associato ad esso. Tale metodo gestisce internamente le proprietà di ogni incantesimo e determina se sia possibile attivarlo. Nel caso affermativo, tutte le conseguenze saranno gestite internamente dall'oggetto dell'incantesimo stesso, liberando il giocatore dalla gestione diretta della logica degli incantesimi.

È inoltre necessario aggiornare lo stato di ogni incantesimo con il passare del tempo, utilizzando il metodo "updateSpellState". Questo meccanismo ricorda vagamente il pattern "Observer", tuttavia non esiste un metodo specifico per dissociare ogni incantesimo dal giocatore, in quanto, per l'applicazione corrente, non risulta necessario. Ogni incantesimo gestisce autonomamente le conseguenze del passare del tempo, senza coinvolgere direttamente il giocatore.

Riuso del codice per l’effetto degli incantesimi

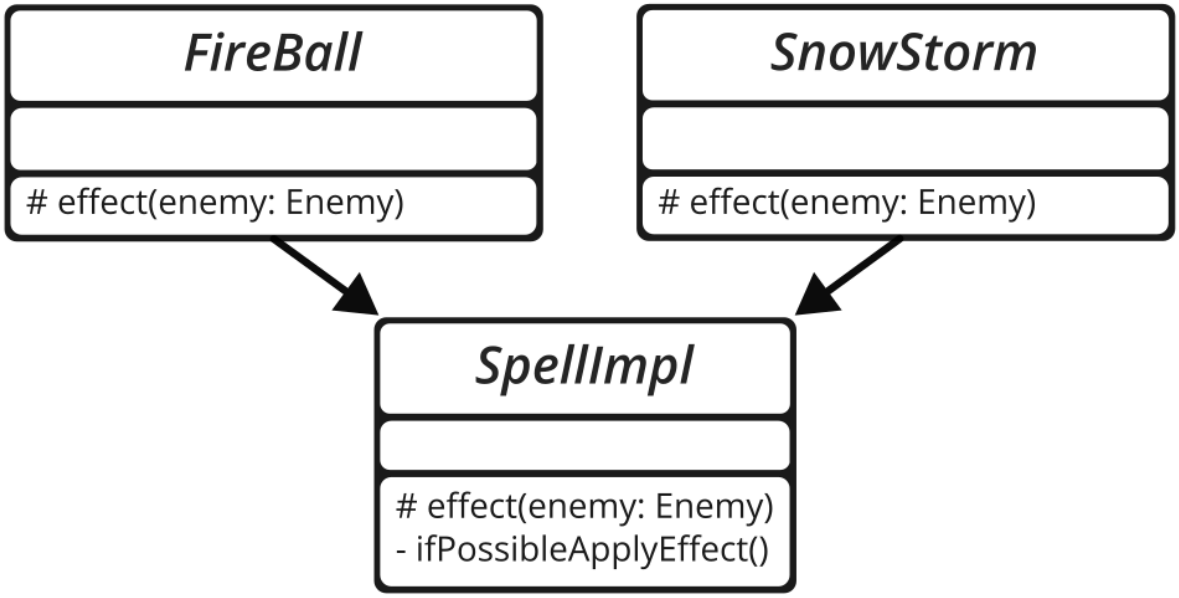

Figura 2.10

Problema

Riguardo all'effettiva implementazione dell'applicazione, sono disponibili due tipologie di incantesimo: "FireBall", che infligge una notevole quantità di danni in un'area circostante, e "SnowStorm", che rallenta i nemici colpiti dal suo effetto. Nonostante i due comportamenti differenti, le due tipologie presentano molte similitudini a livello implementativo, il che potrebbe portare a ripetizioni.

Soluzione

La distinzione fondamentale tra le due tipologie di incantesimo consiste nell'effetto che continuano ad applicare ai nemici dopo essere stati attivati. Pertanto, si è scelto di utilizzare il pattern “template”, introducendo il metodo "effect" astratto e protetto. In questo modo, la classe SpellImpl si occupa di applicare l'effetto dell'incantesimo su ogni nemico che si trova a portata, attraverso il metodo "ifPossibleApplyEffect". L'implementazione di questo metodo è riservata alle classi specifiche, che mantengono variabili speciali interne che variano in base all'effetto applicato. Nel caso di Fireball, viene applicato un danno iniziale comune a tutti gli incantesimi, seguito da una quantità di danno supplementare che si prolunga fino alla fine dell'effetto, sebbene inferiore a quella iniziale. Al contrario, SnowStorm non applica alcun danno iniziale, ma rallenta i nemici del venti percento ogni volta che l'effetto viene utilizzato. Questo effetto viene resettato solamente al termine dell'incantesimo.

Consistente progressione del gioco

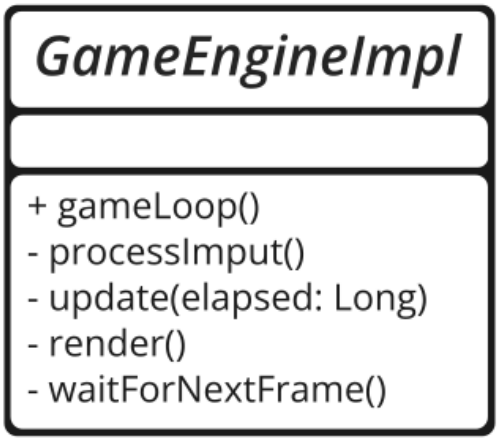

Figura 2.11

Problema

Si richiede che il programma sia in grado di aggiornare la sua logica interna e la sua interfaccia grafica in modo indipendente dall'input dell'utente e dalle capacità del calcolatore su cui viene eseguito, garantendo un'esperienza quasi equivalente per tutti gli utenti.

Soluzione

Il pattern di programmazione di gioco "Game Loop" permette di garantire il normale flusso del programma e l'aggiornamento di tutti i suoi componenti. Il metodo "gameLoop" lo implementa utilizzando i seguenti metodi:

- "processInput": si occupa di processare l'input dell'utente e, se presente, richiede le informazioni sulle interazioni avvenute con l'interfaccia grafica. Se le informazioni sono presenti, la elabore una ad una, partendo dalla meno recente, e la comunica all'oggetto del modello che si occuperà di gestirla. Nel caso specifico di chiusura si occupa personalmente di terminare il programma.
- "update": utilizzando la variabile "elapsed" codificata in millisecondi, comunica al modello il tempo passato dall'ultimo ciclo, in modo che tutti gli elementi che dipendono dallo stesso possano essere aggiornati di conseguenza.
- "render": richiede all'interfaccia grafica di aggiornarsi, osservando le informazioni ricevute dal modello indipendentemente dalla loro variazione.
- "waitForNextFrame": alla fine del ciclo, se necessario, attende un certo periodo di tempo, bloccando il numero di cicli al secondo ad una quantità esatta, per limitare la richiesta computazionale dell'applicazione; purchè il calcolatore sia in grado di mantenere il ritmo proposto.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

- TowerImplTest: viene testato il metodo attack() di una torre. Durante il test viene creato un modo di test e posizionato al suo interno una torre ed un nemico nella stessa posizone. Il test viene superato se il nemico viene ucciso dalla torre.

- EnemyImpltest: viene testato il movimento del nemico in un percorso fittizio creato appositamente per il test. Viene testato sia il movimento quando la velocità è un divisore della lunghezza del percorso, sia quando la velocità non è un divisore della lunghezza e quindi c'è il rischio che il nemico si muova al di fuori del percorso.

- WorldImplTest: vengono testati i metodi sorroundingEnemies e tryBuildTower: nel primo test si verifica che il metodo torni una lista in cui: sono presenti solo i nemici che stanno sul percorso e che rientrano nel raggio specificato; il primo elemento è il nemico che si trova nella posizione più avanzata sul percorso rispetto agli altri. Nel secondo test si verifica che il metodo ritorni true quando si cerca di costruire una torre con un costo abbordabile in una posizione disponibile e che ritorni false nei casi in cui le due condizioni precedenti non sono soddisfatte.

- SpellImpltest: viene testato il metodo di utilizzo tipico di un incantesimo e tutti i metodi al riguardo. Per prima cose si verifica, aggiornando prima per una quantità di tempo sicuramente inferiore a quella effettiva, se l'incantesimo rispetti il suo tempo di caricamento prima di essere effettivamente utilizzabile. Una volta raggiunto il tempo necessario si inserisce nel raggio dell'incantesimo un nemico incapace di muoversi e si aggiorna lo stato dell'incantesimo gradualmente, verificando sia il danno iniziale che il danno (in questo caso esempio) applicato dall'effetto siano effettivamente inferti al nemico. Infine, si verifica che il nemico sia stato sconfitto e che l'incantesimo sia esaurito.

3.2 Metodologia di lavoro

La metodologia di lavoro che abbiamo adottato ci ha portato a seguire un approccio abbastanza classico ma funzionale: in fase di design abbiamo definito le principali dipendenze tra le componenti del gioco ideando delle possibili interfacce da utilizzare. Durante lo sviluppo poi è stato necessario rivedere periodicamente i design iniziali in modo da armonizzare al meglio i rapporti tra le componenti sviluppate separatamente da ognuno di noi, seguendo un approccio a spirale. Il DVCS Git è stato utilizzato per collaborare sullo stesso codice: ogni componente del team ha quindi sviluppato ogni funzionalità su un branch dedicato e il flusso principale ("master") ha svolto la funzione di collante tra tutti gli altri branch che nascono da esso e poi vi si ricongiungono.

Di seguito elenchiamo le classi realizzate da ognuno dei componenti e quelle svolte in collaborazione:

Buda Francesco:

- World
- Bank
- Integrity
- BankImpl
- IntegrityImpl
- WorldImpl
- WorldImpl.Builder
- Sprite
- Position

Ceredi Tommaso:

- Tower
- TowerImpl
- Hunter
- Cannon
- MenuPanel

Maglia Danilo:

- Pair
- Entity
- Enemy
- Horde
- Path
- Wave
- EnemyImpl
- Goblin
- Orc
- HordeImpl
- PathImpl
- WaveImpl
- ErrorDialog

Severi Tommaso:

- Player
- Spell
- PlayerImpl
- SpellImpl
- FireBall
- SnowStorm
- Input
- InputHandler

- InputImpl
- InputHandlerImpl
- GameEngine
- GameEngineImpl
- SpriteAnimation

Buda Francesco e Maglia Danilo:

- SpriteLoader

Ceredi Tommaso e Maglia Danilo:

- EntityImpl

Maglia Danilo e Severi Tommaso:

- View
- ViewImpl
- DefenseButtonPanel

Ceredi Tommaso e Severi Tommaso:

- DefenseEntity

Tutto il team:

- GamePanel
- UnrealDefense

3.3 Note di sviluppo

Buda Francesco

Utilizzo di Stram e Lambda expressions

Utilizzati in vari punti. Un esempio è https://github.com/FourThreads/OOP22-unrl-defense/blob/de93a7a471568c06697f870c1cc0af5216d9bf03/src/main/java/it/unibo/unrldef/model/impl/WorldImpl.java#L121-L126

Utilizzo della libreria JSON-simple

Permalink: https://github.com/FourThreads/OOP22-unrl-defense/blob/33ba719492a04139a3407bc39e3488d5db2a1b0d/src/main/java/it/unibo/unrldef/graphics/impl/SpriteLoader.java#L97-L131

Ceredi Tommaso

Utilizzo di Stream e Lambda

Permalink: https://github.com/FourThreads/OOP22-unrl-defense/blob/33ba719492a04139a3407bc39e3488d5db2a1b0d/src/test/java/it/unibo/unrldef/model/impl/TowerImplTest.java#L42-L43

Utilizzo di Optional

Permalink: https://github.com/FourThreads/OOP22-unrl-defense/blob/de93a7a471568c06697f870c1cc0af5216d9bf03/src/main/java/it/unibo/unrldef/model/impl/TowerImpl.java#L64-L69

Utilizzo di metodi astratti

Permalink: https://github.com/FourThreads/OOP22-unrl-defense/blob/de93a7a471568c06697f870c1cc0af5216d9bf03/src/main/java/it/unibo/unrldef/model/impl/TowerImpl.java#L76

Maglia Danilo

Utilizzo di Stream e Lambda

Permalink: https://github.com/FourThreads/OOP22-unrl-defense/blob/de93a7a471568c06697f870c1cc0af5216d9bf03/src/main/java/it/unibo/unrldef/graphics/impl/GamePanel.java#L186-L199

Progettazione con generici

Permalink: https://github.com/FourThreads/OOP22-unrl-defense/blob/f8df8d81b0371f71d776114530fa2d0b0a908ba1/src/main/java/it/unibo/unrldef/common/Pair.java#L10-L63

Utilizzo di Optional

Permalink: https://github.com/FourThreads/OOP22-unrl-defense/blob/de93a7a471568c06697f870c1cc0af5216d9bf03/src/main/java/it/unibo/unrldef/model/api/Wave.java#L14

Utilizzo di metodi astratti

Permalink: https://github.com/FourThreads/OOP22-unrl-defense/blob/de93a7a471568c06697f870c1cc0af5216d9bf03/src/main/java/it/unibo/unrldef/model/impl/EntityImpl.java#L76

Utilizzo della libreria JSON-simple

Permalink: https://github.com/FourThreads/OOP22-unrl-defense/blob/de93a7a471568c06697f870c1cc0af5216d9bf03/src/main/java/it/unibo/unrldef/model/impl/LevelBuilder.java#L53-L88

Severi Tommaso:

Utilizzo di Stream e Lambda expressions

Utilizzati in vari punti. Un esempio è https://github.com/FourThreads/OOP22-unrl-defense/blob/de93a7a471568c06697f870c1cc0af5216d9bf03/src/main/java/it/unibo/unrldef/model/impl/PlayerImpl.java#L82-L86

Utilizzo di Optional

Permalink: https://github.com/FourThreads/OOP22-unrl-defense/blob/de93a7a471568c06697f870c1cc0af5216d9bf03/src/main/java/it/unibo/unrldef/input/impl/InputImpl.java#L17-L18

Utilizzo di Costruttori multipli:

Permalink: https://github.com/FourThreads/OOP22-unrl-defense/blob/de93a7a471568c06697f870c1cc0af5216d9bf03/src/main/java/it/unibo/unrldef/input/impl/InputImpl.java#L27-L60

Presa ispirazione dalla repository oop-game-prog-patterns-2022 su GitHub di aricci303 in particolare dalla classe step-07-game-over/src/rollball/core/GameEngine.java per lo scheletro della classe di GameEngine e dalla parte di codice GameOver per la schermata finale del gioco.

Considerazioni:

Il motivo per cui abbiamo scelto di utilizzare dei file di config è perché ci siamo ritrovati numerosi magic number, principalmente nella costruzione del mondo e nel caricamento degli sprite. Per questo motivo abbiamo scelto di caricarli da file invece che creare molte costanti. Il motivo per cui abbiamo scelto il formato JSON è perché è molto facile da leggere e scrivere senza essere ambiguo, inoltre è facilmente gestibile a livello di codice soprattutto con l'utilizzo della libreria "json-simple", che offre metodi e classi per facilitare il parsing del file.

Capitolo 4

Commenti Finali

4.1 Autovalutazione e lavori futuri

Buda Francesco

Sono partito con molto entusiasmo verso la realizzazione di questo progetto perché si trattava della prima vera e propria esperienza nella progettazione di un software elaborato e non

banale. A progetto finito, guardando indietro, mi ritengo soddisfatto del lavoro svolto e del mio contributo dato al gruppo. Essendo questa la prima esperienza non ho sviluppato ancora un buon metro di giudizio per valutare il design e l'implementazione di ciò che ho prodotto, sicuramente esistono delle strategie che potrebbero migliorare la coerenza, l'eleganza e l'estendibilità di quanto realizzato, ma credo che questo sia un buon inizio.

Per quanto riguarda il gruppo, dovendo realizzare il mondo di gioco, ho collaborato in modo proficuo con tutti, e credo di aver dato un discreto contributo per la buona riuscita del lavoro. Tuttavia, credo si possa sempre fare di più per sostenere gli altri e distribuire equamente il carico di lavoro. Forse in questo caso avrei potuto sforzarmi maggiormente per cercare di lavorare di più anche su ciò che non riguardava direttamente la mia parte di progetto. In ogni caso ho cercato di aiutare i miei compagni al meglio che potevo e ho anche imparato molto da loro.

Arrivato alla fine sento che le mie competenze riguardanti la programmazione ad oggetti sono migliorate notevolmente, tanto che mi vengono in mente ora diversi e forse migliori modi di implementare la nostra idea iniziale.

Ceredi Tommaso

Personalmente reputo il progetto una ottima sfida a cui abbiamo partecipato per l'esame di programmazione ad oggetti.
In questo modo ho potuto approfondire le mie conoscenze di Java e in generale della programmazione orientata agli oggetti dato che durante il semestre mi erano rimaste delle lacune.

Il nostro gruppo penso che abbia lavorato bene di fronte alle difficoltà incontrate, ma soprattutto c'è stata la giusta comunicazione tra i vari membri per tenerci sempre aggiornati sullo sviluppo. Credo sia stato importante, sia per la mia crescita personale sia per la buona riuscita del progetto presentato, la nostra capacità di essere trasparenti, sintetici e precisi nelle correzioni che vicendevolmente ci siamo scambiati: saper lavorare in un buon gruppo vuol dire anche accettare la possibilità di ricevere critiche costruttive in un'ottica di coesione e scambio reciproco di nozioni e conoscenze.

Per quanto riguarda il mio operato, come dicevo prima, sono soddisfatto di aver migliorato notevolmente le mie conoscenze. Riconosco però che, a causa di limitatissime ore a disposizione nelle mie giornate, l'operato dei miei compagni è più accurato. Sono grato di aver lavorato in un gruppo che mi ha consentito di gestire il mio tempo e il mio impegno, che sono stati comunque costanti e precisi, senza farmi mai pesare il fatto che potessi aver contribuito in maniera non quantitativamente uguale. Sicuramente la qualità del mio lavoro è sempre stata al massimo, nel tempo che avevo da dedicare, e non mi sono mai tirato indietro davanti a critiche o miglioramenti riguardo a quanto ho svolto. Mi sono sempre messo in gioco e in discussione, imparando a valorizzare sia il mio lavoro sia quello svolto dai miei colleghi. Collaborare con delle persone che sappiano stimolare e accrescere interesse e competenze, credo sia uno dei risultati più gratificanti nella realizzazione di un progetto: reputo

soddisfacente il risultato ottenuto non solo per l'effettivo funzionamento del videogioco, ma anche per la qualità del tempo investito e delle persone con cui ho collaborato.

Maglia Danilo

La realizzazione del progetto è stata un ottimo modo per mettermi in gioco e capire veramente le mie abilità nello sviluppo in Java e soprattutto nella collaborazione in un team. Mi sono ritrovato più volte a chiedere aiuto e dare una mano ai miei compagni e penso di essere riuscito a dare il mio contributo nel miglior modo possibile. Non nego che mi sono scontrato più volte con i miei colleghi riguardo idee contrastanti ma tutte le volte siamo riusciti a trovare un modo per risolvere il problema che riuscisse ad utilizzare le idee di tutta la squadra.

Grazie a questo progetto sono riuscito ad approfondire la mia conoscenza del linguaggio e sono riuscito a capire meglio certi argomenti che durante il semestre non avevo capito a fondo, come la GUI o l'utilizzo di pattern per la programmazione.

Sono contento del risultato e soprattutto è stata una sfida divertente che mi ha permesso di migliorare le mie abilità nel lavoro di gruppo. Grazie al progetto mi sento più confidente delle mie abilità e penso di essere in grado di utilizzare le cose imparate anche al di fuori dell'ambito universitario e/o lavorativo.

Severi Tommaso

Ritengo che lo sviluppo di questo progetto abbia costituito una notevole sfida, ma allo stesso tempo ho avuto la possibilità valutare che, se data l'occasione, sarei in grado di fornire un contributo significativo e di poter essere, per quanto più possibile, all'altezza della situazione. Inoltre, ho avuto modo di apprezzare l'importanza del lavoro di gruppo in quanto, spesso, tendo a perdermi in tangenti sviluppativi che a volte, se ripensate insieme ad un collega, risultano, se non inutili, esagerate rispetto alla portata effettiva del progetto.

Il progetto mi ha permesso di approfondire la mia conoscenza del linguaggio di programmazione Java, dimostrandone la notevole flessibilità e potenza. In particolare, ho avuto l'opportunità di consolidare le mie competenze nell'ambito della creazione di interfacce grafiche Swing, in cui in precedenza avevo riscontrato alcune lacune.

Ho sempre cercato di essere quanto più mi era possibile a disposizione di tutto il gruppo e di non tirarmi indietro di fronte a qualsiasi problema che potesse sorgere, indipendentemente se facesse parte della sezione a me interessata. In alcune occasioni, tuttavia, mi rendo conto di aver agito con troppa impulsività e avrei dovuto prestare maggiore attenzione alle opinioni e ai suggerimenti dei miei colleghi.

4.2 Difficoltà incontrate e commenti per i docenti

Ad inizio progetto eravamo davvero molto poco preparati per quello che stavamo per affrontare, da un lato è normale visto che era la prima volta, dall'altro però ci siamo resi conto che forse sarebbe stato meglio se avessimo avuto l'opportunità di fare un numero maggiore di esercizi in laboratorio legati al design. A lezione si parla di ereditarietà, incapsulamento, composizione, pattern... ma in laboratorio capita raramente di dover pensare da zero a come impostare un software semplice in modo che sia ottimizzato al meglio sul lato del design.

Appendice A

Guida utente

Quando viene avviato il software è richiesto al giocatore l'inserimento di un nome utente, il quale renderà disponibile il pulsante per avviare il livello.

Appena caricato il livello, il giocatore avrà a disposizione delle monete iniziali con le quali potrà comprare le prime torri di difendersi dai nemici.

Per posizionare le torri è sufficiente selezionarne una dal menu di destra e posizionarla nelle zone predefinite che compariranno sulla mappa.
Allo stesso tempo è possibile utilizzare delle pozioni da applicare sui nemici per ostacolare la loro l'avanzata, le pozioni si ricaricano a tempo e non è necessario utilizzare mote.

Se un nemico raggiungerà la fine del percorso andrà a danneggiare il castello e gli rimoverà una vita.
Se il catello termina le vite il giocatore avrà perso la partita.

In caso contrario, quando le ondate di nemici saranno terminate, il giocatore vincerà la partita.

Appendice B

Esercitazioni di laboratorio

B.0.1 francesco.buda3@studio.unibo.it

- Laboratorio 04: https://virtuale.unibo.it/mod/forum/discuss.php?d=113869#p169128
- Laboratorio 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=114647#p170054
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=117852#p174139
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=118995#p175123

B.0.2 tommaso.ceredi@studio.unibo.it

B.0.3 danilo.maglia@studio.unibo.it

- Laboratorio 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=114647#p170067

B.0.4 tommaso.severi2@studio.unibo.it

- Laboratorio 03: https://virtuale.unibo.it/mod/forum/discuss.php?d=112846#p168138

- Laboratorio 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=114647#p170050
- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=115548#p171334
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=117044#p173085
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=117852#p174174
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=118995#p174908
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=119938#p176471
- Laboratorio 11: https://virtuale.unibo.it/mod/forum/discuss.php?d=121130#p177602