Relazione Progetto OOP
"Dimension Holiday"

Lorenzo Prati, Elvis Perlika
Emanuele Dajko, Alessandra Versari

June 9, 2023

Contents

1 Analisi **3**
- 1.1 Requisiti . . . 3
- 1.2 Analisi e modello del dominio . . . 4

2 Design **6**
- 2.1 Architettura . . . 6
- 2.2 Design dettagliato . . . 9
  - 2.2.1 Lorenzo Prati . . . 9
    - Entity e Component . . . 9
    - System . . . 10
    - Engine . . . 12
    - World e Eventi . . . 13
    - Movimento e Posizione . . . 15
    - Collisioni e fisica . . . 17
    - Logica del giocatore e Input . . . 19
    - Schermata vittoria/sconfitta . . . 22
  - 2.2.2 Elvis Perlika . . . 22
    - AI . . . 23
    - Combat System . . . 25
    - HUD . . . 25
  - 2.2.3 Emanuele Dajko . . . 25
    - Introduzione al concetto di gestione delle stanze all'interno del gioco . . . 28
  - 2.2.4 Alessandra Versari . . . 38

3 Sviluppo **42**
- 3.1 Testing automatizzato . . . 42
- 3.2 Metodologia di lavoro . . . 43
- 3.3 Note di sviluppo . . . 46
  - Lorenzo Prati . . . 46

4 Commenti finali **48**
4.1 Autovalutazione e lavori futuri 48
Lorenzo Prati 48
4.2 Difficoltà incontrate e commenti per i docenti 49
Lorenzo Prati 49
Elvis Perlika 49

5 Guida utente **51**

6 Esercitazioni di laboratorio **52**

Chapter 1

Analisi

1.1 Requisiti

Il gruppo si pone l'obbiettivo di realizzare un videogioco roguelike *Dimension Holiday* vagamente ispirato a giochi famosi come *Hades* oppure *The Binding of Isaac*. Il giocatore controllera' un personaggio esplorare un dungeon composto di diverse stanze e affrontare un boss per vincere il gioco.

Elementi funzionali

- Per attraversare tutto il dungeon il giocatore dovrà passare da stanza in stanza, eliminando tutti i nemici presenti. Per farlo potrà usare la spada, lanciare proiettili o utilizzare una magia di fuoco. Giocatore e nemici hanno delle vite (espresse in cuori) e se il giocatore perde tutti i cuori e' *Game over* e si deve ricominciare da capo
- Una volta che il giocatore avrà ucciso tutti i nemici della stanza corrente un portale (*Gate*) che e' presente nella stanza potra' essere attraversato per passare alla stanza successiva
- Dopo un certo numero di stanze, comparirà una stanza shop dove sarà possibile effettuare acquisti usando le monete creando una piccola progressione
- Il gioco si conclude quando il giocatore sconfigge un nemico speciale chiamato *Boss*

Elementi non funzionali

- Ci si pone l'obbiettivo di creare un'architettura del software modulare ed espandibile ad aggiunte future, come l'aggiunta di nuovi nemici, mappe e oggetti.

1.2 Analisi e modello del dominio

Il giocatore potra' muoversi nelle 4 direzioni su, sinsitra, giu', destra tramite i tasti, rispettivamente, W, A, S, D ed effettuare due tipi di attacchi piu' un'abilita' speciale:

- un attacco ravvicinato, usando la sua spada e tramite il tasto sinistro del mouse
- un attacco dalla distanza, usando un proiettile energetico e tramite il tasto destro del mouse
- un attacco caricato, tenendo premuto il tasto Z per 3 secondi e rilasciandolo per sparare una palla di fuoco che fara' piu' danno di un normale proiettile

Il gioco dovra' essere in grado di presentare al `giocatore` una serie di stanze dove affrontare dei `nemici`. Questi potranno avere diversi comportamenti e diverse tipologie di `attacco`. Il giocatore dovrà stare attento ad evitare gli attacchi dei nemici per non perdere cuori. I nemici possono attaccare sia dalla distanza con dei proiettili sia da vicino. Quando un'entita' esaurisce i cuori, essa deve essere rimossa dal `mondo` di gioco; inoltre alla rimozione di determinate entita' il gioco puo' reagire in diversi modi, ad esempio decretando il *Game over* nel caso il player muoia, oppure la vittoria nel caso il boss venga sconfitto. Saranno presenti degli `oggetti` raccoglibili (cuori, monete ecc.) che dovranno applicare degli effetti alle entita' con cui entrano in contatto, ad esempio l'incremento della vita oppure l'incremento della valuta posseduta dal giocatore. Lo shop sara' gestito *in game*, nel senso che non comparirà un'interfaccia grafica che permettera' al giocatore di scegliere i potenziamenti, ma il giocatore dovra' interagire dinamicamente con degli oggetti presenti nella mappa per acquistarli. Anche gli attacchi applicheranno degli effetti con le entita' con cui entrano in contatto, come ad esempio la perdita di cuori. Il `mondo` di gioco sara' composto di una serie di `stanze`. Tramite l'interazione con un oggetto portale, il giocatore sara' trasportato alla stanza successiva senza possibilità di tornare indietro. All'interno delle stanze sono presenti dei muri, che bloccano il passaggio al giocatore. Esistono

tre tipi di stanze: normale, shop, boss. Nella stanza normale compariranno dei nemici, in numero e tipo variabile in base al momento della partita, nella stanza shop invece compariranno gli oggetti rappresentati i potenziamenti acquistabili, e nella stanza boss comparirà il nemico boss.

Una delle maggiori difficoltà consistera' nella creazione di un architettura che permetta la gestione sia di diversi tipi di nemici (zombie, shooter, boss ecc.), ognuno con un proprio comportamento, sia di diversi attacchi utilizzabili sia dal giocatore che dai nemici (proiettili, attacchi meele). Inoltre, si cerchera' di realizzare un sistema di combattimento *action* e *real-time* quanto piu' possibile fluido e responsivo e un'alternanza di mappe e generazione dei nemici in modo tale da far sembrare ogni partita diversa.

Dato il monte ore previsto, si rimanda al futuro una gestione accurata delle performance del gioco.

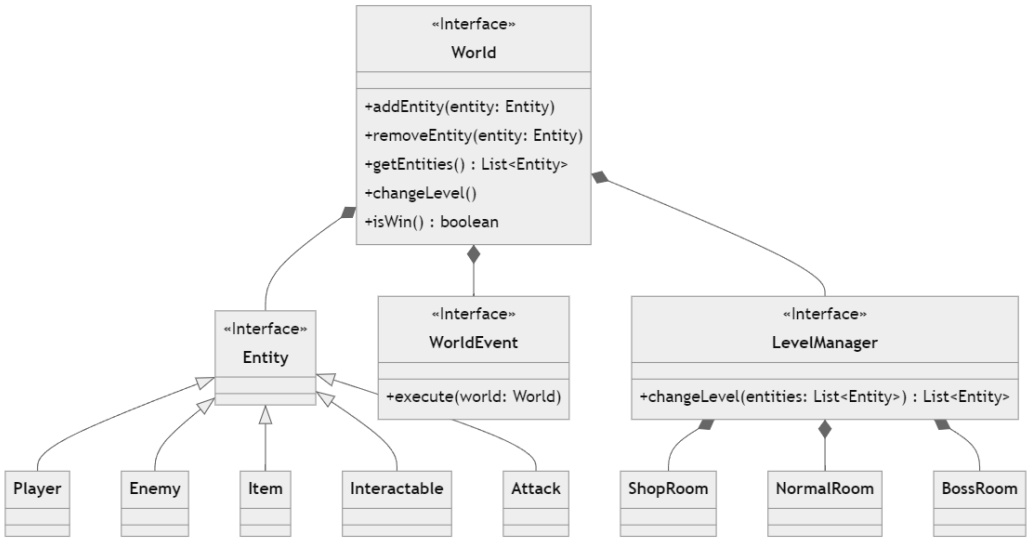

Figure 1.1: Diagramma UML dell'analisi del dominio rappresentante le varie entita' di gioco e i loro legami con il mondo e i livelli

Chapter 2

Design

2.1 Architettura

Abbiamo deciso di utilizzare per il modello del gioco una versione semplificata del *pattern entity-component-system* (*ecs*) mantenendo la parte grafica separata.

Dopo aver tentato un approccio piu' orientato alle gerarchie di classi, ci e' sembrato naturale spostarci verso una visione che fattorizzasse gli aspetti comuni dei diversi attori in gioco in modo piu' modulare, cercando di separare i *dati* (*component*) dalla *logica* che li comanda (*system*).

Questo pattern ci e' sembrato piu' adatto a modellare il nostro gioco perche' supporta la facile creazione di nuovi attori (ad esempio nuovi oggetti, nemici ecc.) garantendo una buona suddivisione del codice e delle responsabilita', un alto riuso e un alto grado di composizione (*composition over inheritance*).

Di seguito spieghiamo le varie parti della nostra architettura:

- `Component`: sono oggetti utili a mantenere i dati che descrivono un certo aspetto del modello di gioco (ad esempio la posizione sul piano, la direzione del movimento, le caratteristiche del corpo ecc.) e in teoria non hanno una loro logica di comportamento.

- `Entity`: sono dei raccoglitori di `Component`. Ogni entita' e' descritta dai suoi componenti che permettono alla stessa di distinguerla dalle altre entita'. Ad esempio, diverse entita' presenti in gioco nello stesso momento potrebbero contenere un PositionComponent, un MovementComponent, un BodyComponent, un HealthComponent e altri, che ne descrivono le proprieta'.

- `GameSystem`: sono la parte dell'architettura che si occupa di operare sulle entita', modificandone i componenti e quindi svolgendo la maggior parte della logica del gioco. L'esecuzione sequenziale di vari sistemi, ciascuno che scorre le entita' e opera su un determinato set di componenti, permette il funzionamento del gioco. Ad esempio, il MovementSystem opera esclusivamente sulle entita' che contengono il MovementComponent e si occupa di muovere tutte le entita'; mentre il CheckHealthSystem si occupa di prendere tutte le entita' che hanno un HealthComponent e di rimuovere quelle che hanno esaurito le vite.

- `Engine` e `World`: queste sono le classi che controllano effettivamente lo svolgersi del gioco. `Engine` si occupa di gestire il *game loop*, mentre il `World` contiene al suo interno le entita', esegue i *system* e passa alla View le informazioni necessarie per disegnare su schermo.

- `LevelManager`: manager dei livelli contenuto nel `World`, che si occupa della generazione e del caricamento degli stessi.

- `View`: e' gestita in modo indipendente dal *pattern ecs*. `Scene` si occupa solamente di disegnare lo stato del modello di gioco, mentre `MainWindow` gestisce diverse schermate di menu (home, opzioni, pausa ecc.). Inoltre sono presenti classi che si occupano di disegnare l'interfaccia grafica e di registrare gli input da mouse e tastiera.

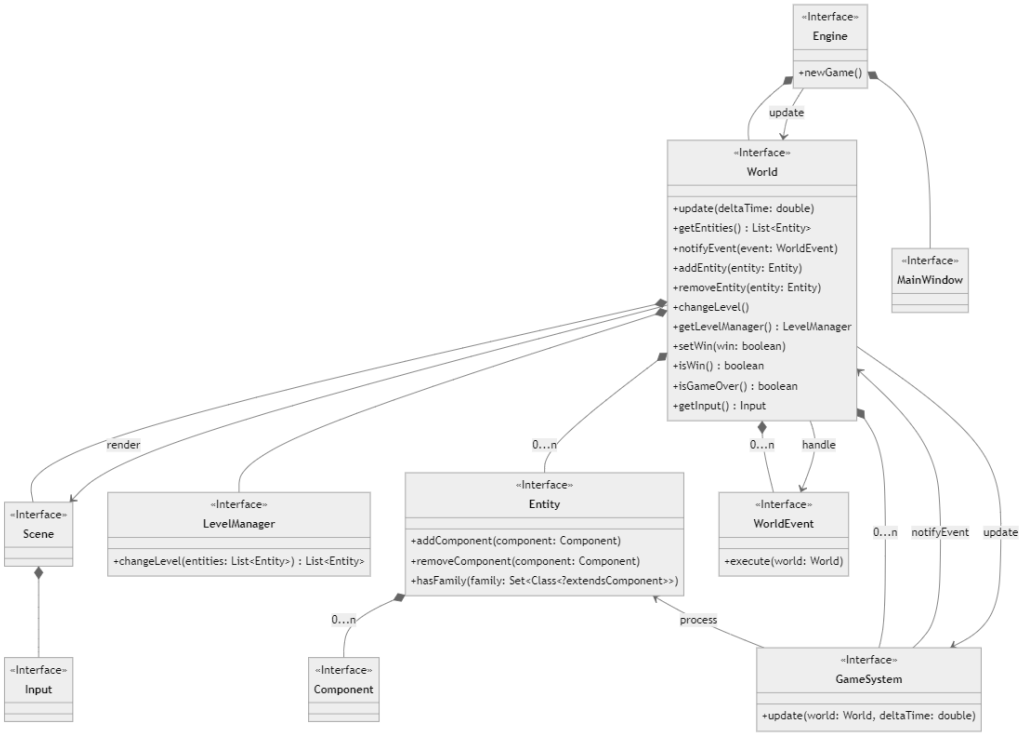

Figure 2.1: Diagramma UML dell'architettura. Le freccie indicano le operazioni ad alto livello che le varie parti svolgono. L'`Engine` fa partire il *game loop* e aggiorna il `World`, che a sua volta aggiorna uno dopo l'altro tutti i `GameSystem`. I sistemi processano le entita' che competono alla loro gestione e possono notificare dei `WorldEvent` al `World`. Il `World`, dopo aver eseguito tutti gli eventi generati dai sistemi, comanda a `Scene` di disegnare il mondo di gioco

.

2.2 Design dettagliato

2.2.1 Lorenzo Prati

Seguendo un approccio bottom-up, di seguito spiego prima il funzionamento della base del *pattern entity-component-system* (*ecs*), per poi passare alla descrizione delle classi fondamentali su cui si poggia il funzionamento dell'applicazione, come Engine e World, e infine dedicarmi ai sistemi e componenti specifici da me realizzati.

Entity e Component

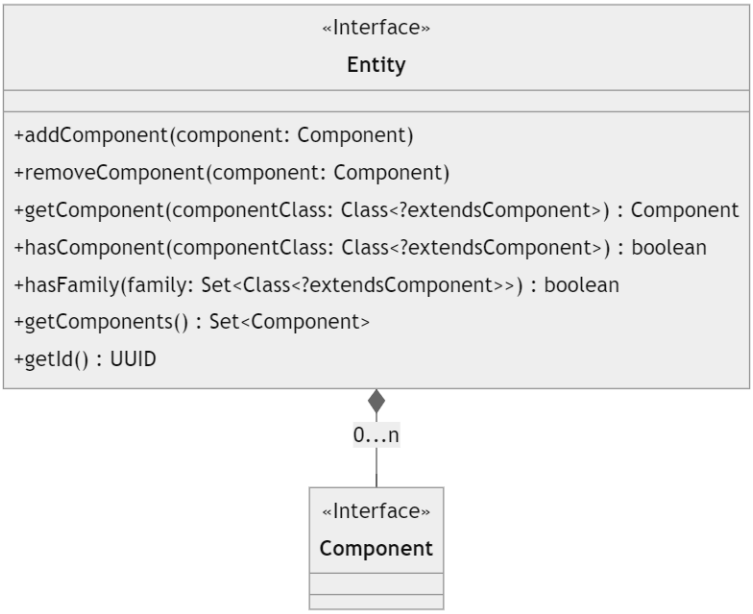

Figure 2.2: Diagramma UML della relazione tra Entity e Component

Problema Modellare il concetto di *entity'* nel contesto del *pattern ecs*. Ogni entita' deve fare riferimento a dei *component* ma non avere una propria logica di comportamento. I componenti devono contenere il piu' possibile solo *dati*. Le entita' devono essere interrogabili sui loro componenti, supportare l'inserimento e la rimozione di componenti, e restituire i componenti richiesti. eventualmente. Nel dominio del gioco, le entita' sono, ad esempio, il giocatore, i nemici, gli oggetti, ma anche gli attacchi.

Soluzione Differentemente dall'approccio classico utilizzato per il *pattern ecs*, che consiste nel rappresentare il concetto di entita' come *id* numerici (interi di fatto) attraverso i quali identificare i vari component, ho scelto di usare un approccio piu' semplice e piu' *object-oriented*, oggetto del corso, realizzando le entita' come classi di tipo `Entity` contenenti un insieme di `Component`. Ciascuna entita' possiede comunque un *id* ma questo e' usato in minima parte e solo per esigenze di gestione che esulano da questo paragrafo e che spieghero' in seguito. Il vantaggio di questo approccio *oop* e' sicuramente nella semplicita' di gestione; infatti liste di entita', ciascuna con relativo insieme di componenti, sono facilmente iterabili e manipolabili all'occorrenza. Di contro, un approccio piu' a basso livello che favorisca l'uso massiccio di *id* e array di componenti avrebbe permesso un notevole incremento delle performance che pero' e' fuori dallo scopo del progetto.

Quindi, l'interfaccia `Component` non ha metodi, e serve a essere implementata da tutti i componenti del gioco. L'interfaccia `Entity`, invece, contiene tutti i metodi fondamentali per manipolare i componenti contenuti in quell'entita', come ad esempio l'aggiunta e la rimozione, e l'ottenimento di uno specifico componente. Tramite il metodo `hasFamily` e' possibile interrogare l'entita' sul possesso di un insieme di `Component` specifici, cosa che risultera' molto utile per i *system*.

Su queste semplici interfacce si basa l'intera struttura del *pattern entity-component-system*, o meglio della parte *entity-component*.

Infine, ho realizzato una classe `EntityBuilder` che, tramite l'uso del *builder pattern*, consente la creazione di entita' in modo dichiarativo semplicemente tramite l'aggiunta sequenziale di `Component` su cui si basano tutte le factory presenti nel progetto. Per creare nuove entita' e' quindi sufficiente aggiungere componenti tramite l'`EntityBuilder` e modificando i parametri di creazione di questi componenti oppure creandone di nuovi, e' di fatti possibile creare molto velocemente nemici, oggetti e attacchi nuovi e dalle caratteristiche diverse.

Un esempio di creazione di entita' sfruttando il builder e il *pattern factory method* da me realizzato: (inserire permalink)

System

Problema Le entita' e i componenti non definiscono logica di comportamento propria, quindi e' necessario che vengano manipolati dai *system*. I sistemi devono essere divisi per specifico compito e ognuno deve essere in grado di operare sulle entita' che hanno certi componenti. Ad esempio, devono essere realizzati dei sistemi in grado di muovere le entita', rilevare e gestire le loro collisioni, rimuoverle dal mondo di gioco quando necessario

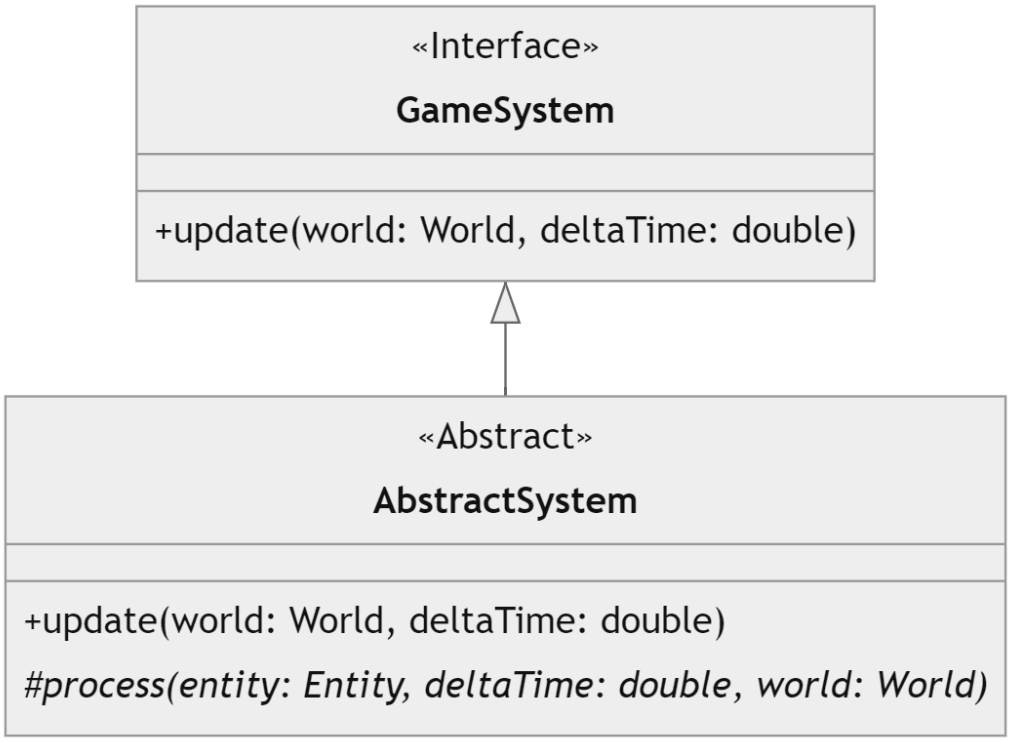

Figure 2.3: Diagramma UML dei *system*

ecc.

Soluzione Ho realizzato un'interfaccia `GameSystem` che modella un generico *system*. Il metodo esposto e' `update` e si occupa di far eseguire la logica del sistema. Visto che ogni *system* deve essere in grado di operare solo su certe entita', per garantire il *riuso* ho scelto di realizzare una classe astratta `AbstractSystem` che all'interno utilizza il *pattern template method* dove il metodo template e' appunto `update`, che fa i dovuti controlli sui componenti che compongono l'entita' e poi richiama il metodo astratto e protetto `process` solo sulle entita' che hanno i componenti richiesti. In questo modo, ogni classe che estende `AbstractSystem` deve solamente occuparsi di definire tramite costruttore l'insieme di `Component` su cui vuole operare e poi implementare il metodo `process` che andra' ad operare solo su `Entity` che hanno i componenti precedentemente richiesti.

I *system* sono in grado, avendo nel loro metodo `process` un riferimento al `World` di notificare, come spieghero' meglio in seguito, degli eventi; ma e' anche possibile un metodo di *signaling* tra *system* diversi. Questo e' pos-

sibile attaccando, in seguito al verificarsi di una determinata situazione, un componente *informativo* all'entita' che si sta processando in modo tale da permettere a successivi *system* di cercare entita' con quel componente *informativo* e gestire la cosa adeguatamente.

In teoria e' quindi sufficiente, per inserire nel gioco una nuova meccanica, costruire nuovi componenti che definiscano nuove proprieta' e un nuovo sistema che operi su di essi senza il bisogno di andare a modificare i sistemi o i componenti precedentemente creati.

Engine

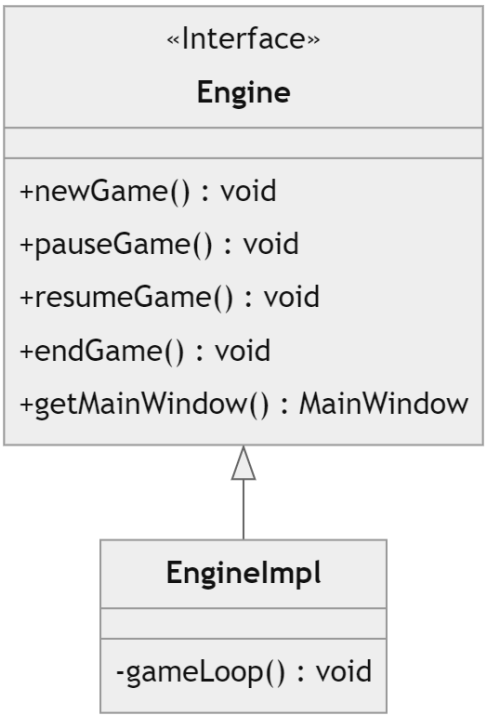

Figure 2.4: Diagramma UML dell'Engine

Problema Realizzare una classe che permetta lo svolgimento effettivo del main loop del gioco, attraverso il quale scandire gli update del modello logico e della rappresentazione grafica, e che contenga tutti gli elementi per mantenere attiva l'applicazione. All'occorrenza, deve anche essere possibile mettere in pausa il gioco e riprenderlo. Deve essere inoltre gestita la fine del gioco.

Soluzione La soluzione e' ricaduta sulla creazione di un'interfaccia `Engine` con relativa implementazione `EngineImpl`. Questa classe fa uso del *pattern game loop*, realizzato in un metodo privato e vengono esposti dall'interfaccia i metodi necessari alle altre classi (ad esempio le schermate della View) per controllare il loop. E' quindi possibile metterlo in pausa, riprenderlo e arrestarlo. Essendo questa classe la prima che viene creata al lancio dell'applicazione, essa si occupa anche internamente di creare il modello logico e la view.

World e Eventi

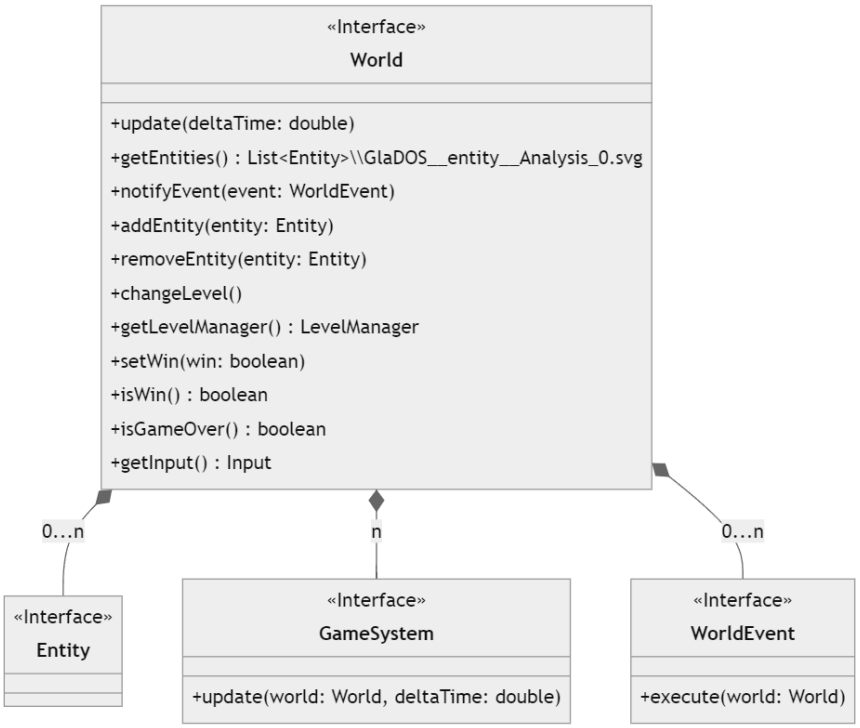

Figure 2.5: Diagramma UML del World

Problema Occorre contenere e mantenere le `Entity`, e comandare la logica che opera su di esse, cioe' i *system*. Allo stesso tempo, e' necessario passare alla View le informazioni necessarie affinche' possa disegnare sia le entita' che la mappa di gioco.

Soluzione Ho scelto di unire in un'unica classe la funzione di contenere le entita' e i sistemi, e la rappresentazione grafica del mondo di gioco (`Scene`), quindi ho realizzato l'interfaccia `World` con la relativa implementazone `WorldImpl`. Ho optato per questa scelta di nome nonostante la funzione della classe sia piu' comparabile a quella di un classico *Controller* del pattern MVC poiche' e' utilizzato spesso nel contesto del *pattern ecs*.

L'interfaccia `World` espone i metodi necessari al controllo del gioco vero e proprio e delle entita', come ad esempio il metodo `update`, che viene chiamato ad ogni ciclo del *game loop*. Questo e' il metodo principale della classe, perche' si occupa di:

- eseguire tutti i `GameSystem`, che equivale ad *aggiornare* il modello
- eseguire tutti gli eventi in coda nel `World`
- comandare alla scena di disegnare

Ho scelto di memorizzare e gestire i `GameSystem` direttamente in una lista dentro il `World` per semplicita' quando di fatti l'ordine con cui essi vengono memorizzati nella lista e quindi eseguiti ad ogni ciclo di *update* potrebbe essere gestito da uno *strategy pattern* ed essere modificabile in futuro con un'altra implementazione; tuttavia ho scelto di semplificare l'approccio perche' in ogni caso non viene gestita in nessun modo l'abilitazione/disabilitazione di sistemi a run-time (come e' comune nel *pattern ecs*) e in generale in un gioco semplice come il nostro i pochi sistemi presenti devono eseguire strettamente uno dopo l'altro in un certo ordine preciso che di fatti lascia spazio a poche variazioni. Per questi motivi, la gestione dei sistemi e' fissa e non modificabile, e per questo l'inizializzazione dei `System` e' gestita internamente al `World` e non e' visibile o modificabile dall'esterno.

Poi il `World` espone i metodi per la gestione delle `Entity`, cioe' che permettono di aggiungere e rimuovere `Entity`, oppure di avere una copia delle entita' presenti in gioco.

Gli eventi sono rappresentati tramite l'interfaccia `WorldEvent` e sono gestiti in modo asincrono. Durante la loro esecuzione, i vari *system* possono notificare il `World` di uno specifico evento, che verra' messo in una coda ed eseguito solo dopo che tutti i `System` hanno finito la loro esecuzione. In questo modo si evita il verificarsi di comportamenti anomali dovuti all'immissione di `Entity` nel `World` nel mezzo dell'esecuzione dei *system* durante la quale, d'altra parte, e' risultato comodo sia immettere che rimuovere le `Entity` tramite eventi dedicati, in modo da reagire all'aggiunta/eliminazione di una specifica entita' (ad esempio per il *game over*). Mi sono occupato io di implementare tutti gli eventi, che saranno pero' lanciati anche dai *system* realizzati dai miei colleghi:

- `AddEntityEvent` aggiunge l'entita' passata come parametro al `World`
- `RemoveEntityEvent` rimuove dal `World` l'entita' passata come parametro, in questo caso identificata tramite *id*. Nel caso si trattasse dell'entita' che rappresenta il giocatore o il boss, viene notificato il `World` rispettivamente della sconfitta e della vittoria tramite il metodo `setWin(boolean win)`
- `ChangeLevelEvent` chiama il metodo `changeLevel()` del `World`, che procede a gestire il cambio di livello aggiornando le entita' con quelle generate dal `LevelManager` per il nuovo livello e passando alla `Scene` anche la nuova mappa.

Il `World` espone anche un metodo per ottenere la classe con la quale i `System` possono sfruttare l'input dell'utente, che spieghero' piu' nel dettaglio descrivendo il funzionamento del giocatore.

Infine, sono presenti vari metodi che servono a interrogare il `World` sullo stato della partita (`isGameOver()`) e il risultato della partita (`isWin()`), utili soprattutto all'`Engine` che deve sapere quando arrestare il *game loop* e gestire la fine del gioco.

Movimento e Posizione

Problema Tutte le entita' di gioco occupano una posizione nella mappa di gioco, che a livello di modello possiamo rappresentare come un piano 2d. Mentre alcune entita' mantengono la loro posizione invariata nel tempo (come gli item), altre (come i nemici e il giocatore) variano la posizione nel tempo muovendosi in diverse direzioni secondo le logiche dell'intelligenza artificiale o un input dell'utente.

Soluzione Ho realizzato una classe `PositionComponent` che ovviamente implementa l'interfaccia `Component` e memorizza le coordinate della posizione di un'entita' in quel momento. Inoltre, al suo interno viene memorizzata anche la posizione precedente a quella corrente, poiche' sara' utile nel momento di gestire le collisioni. La posizione viene quindi memorizzata come dato all'interno del componente, cosi' da poter attaccare ad ogni entita' (tutte di fatto) che hanno una posizione nella mappa un `PositionComponent`, massimizzando il *riuso di codice*.

Analogamente, per rappresentare il movimento ho creato un componente `MovementComponent`, che memorizza la direzione in cui il movimento deve essere compiuto, tramite un vettore 2d. Inoltre, viene qui memorizzata anche

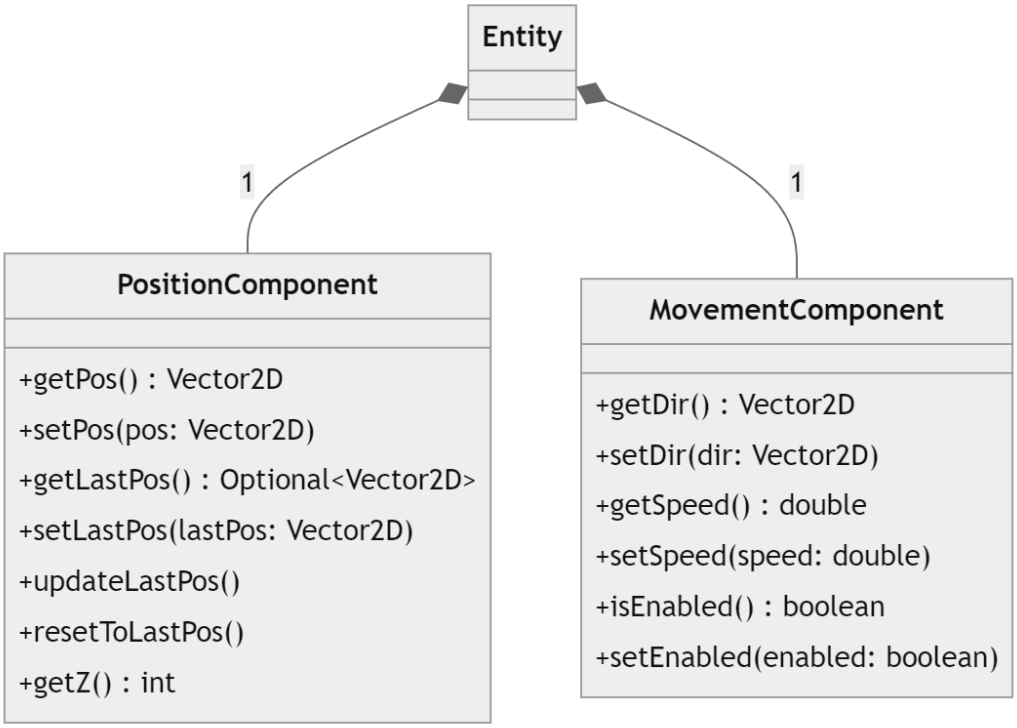

Figure 2.6: Diagramma UML dei componenti della posizone e del movimento

la velocita' con la quale l'entita' dovra' compiere il movimento. Il movimento puo' anche essere abilitato/disabilitato. La scelta di gestire il movimento tramite vettori, nonostante le entita' di gioco si muovano quasi tutte soltanto nelle 4 direzioni, permette in futuro anche la facile implementazione del movimento nelle 8 direzioni, o comunque una sua gestione libera.

Una volta definiti questi componenti di base, e' possibile gestire il movimento di tutte le entita' tramite un *system*. Il `MovementSystem` si occupa infatti di processare le entita' che hanno un componente `MovementComponent`, che significa che *posseggono la capacita' di muoversi e hanno i dati necessari affinche' possano essere mosse*, e si occupa di verificare per ciascuna di queste entita' se il movimento e' abilitato e nel caso aggiornare il `PositionComponent` con la nuova posizione risultante dal calcolo del movimento espresso nel `MovementComponent` applicato alla vecchia posizione. L'esecuzione di questo *system* ad ogni ciclo del *game loop*, permette di muovere tutte le entita' che *possono farlo* previo precedente settaggio della direzione in cui l'entita' intende muoversi.

Collisioni e fisica

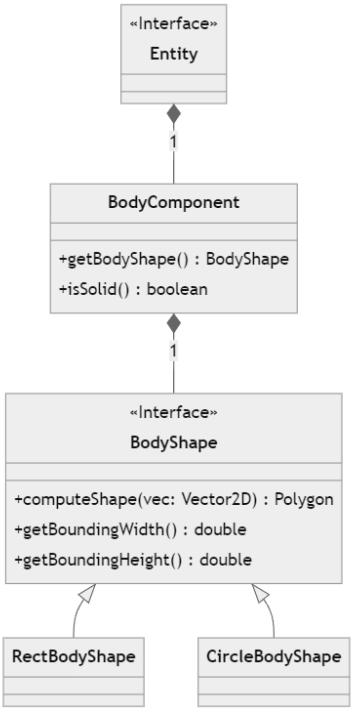

Figure 2.7: Diagramma UML del componente del corpo e delle shape

Problema Alcune entita' hanno corpi *solidi* e devono comportarsi come tali nelle loro azioni di movimento. Inoltre e' necessario registrare quando entita' di qualunque tipo collidono con altre entita', al fine di poterne gestire le conseguenze.

Soluzione Ho realizzato una classe `BodyComponent` che modella il *corpo* di un'entita', definendone proprieta' come la `BodyShape`, ovvero la forma geometrica che il suo corpo occupa nello spazio, e la solidita' (espressa da un booleano).

Il `CollisionSystem` si occupa di processare le entita' che hanno un `BodyComponent`. Per ciascuna di queste entita', viene controllata la collisione con *tutte* le altre entita' presenti nel gioco. Se viene rilevata una collisione, calcolata estrendo dai rispettivi `BodyComponent` le body shape e controllandone l'intersezione in date coordinate, allora viene *registrata* la collisione attaccando all'entita' che il system sta processando in quel momento un `CollisionComponent`, un semplice componente che mantiene i dati sulle

collisioni avvenute. In particolare, viene aggiunto un `CollisionComponent` solo nel caso non ce ne sia gia' uno, poiche' altrimenti vengono aggiornate le informazioni di quello gia' presente aggiungendo i dati sulla nuova collisione. Questo e' un esempio, l'unico in realta' realmente presente in questo progetto, di *signaling tra system*. Infatti, i successivi *system* potranno filtrare le entita' che hanno tra i loro componenti anche un `CollisionComponent` e gestire la collisione in modo appropriato. Su questo meccanismo si basano i sistemi che gestiscono le collisioni fisiche, gli item, il combattimento ecc. poiche' la loro gestione si basa sulla precedente aggiunta di un `CollisionComponent` da parte del `CollisionSystem`.

Un `PhysicsSystem`, che esegue subito dopo il `CollisionSystem`, processa le entita' che hanno `BodyCompoenent` e `CollisionComponent` occupandosi invece della gestione vera e propria della collisione fisica, che pero' nel dominio del gioco si traduce in un semplice reset della posizione. Dato che vengono tenute nel `PositionComponent` sia la posizione corrente che quella immediatamente passata, solo in caso di collisione tra corpi *solidi* viene ripristinata la posizione precedente.

Menziono qui anche la presenza di un `ClearCollisionSystem`, un semplice sistema che esegue dopo che hanno eseguito tutti i sistemi che dovevano in qualche modo gestire la reazione a una collisione (filtrando anche per `CollisionComponent`), rimuovendo tutti i `CollisionComponent` dalle entita' che ne hanno uno. In questo modo, al successivo loop di update dei system, non vengono lasciate collisioni non gestite.

Per quanto riguarda ingine l'interfaccia `BodyShape`, essa puo' essere implementata potenzialmente da classi che rappresentano varie forme geometriche. Io ho realizzato una `RectBodyShape`, che rappresenta la forma geometrica del rettangolo, e una `CircleBodyShape`, che rappresenta il cerchio. L'interfaccia espone il metodo `computeShape(Vector2D)` che permette di ricevere il poligono calcolato in base alle coordinate fornite, utile poi al calcolo dell'intersezione con un altro poligono. Qui ho fatto uso di libreria come spiegato meglio nel paragrafo dedicato (riferimento). Inoltre, sono presenti i metodi `getBoundingWidth()` e `getBoundingHeight()` per conoscere il rettangolo che limita il poligono, utili sia alla View che in altri punti del progetto.

Logica del giocatore e Input

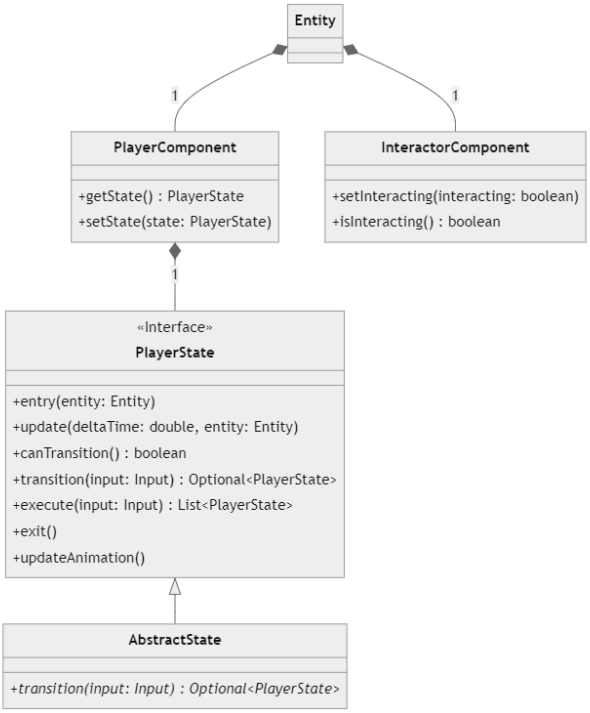

Figure 2.8: Diagramma UML dell'entita' che rappresenta il giocatore e alcuni dei suoi componenti

Problema L'utente controlla un personaggio in grado di:

- muoversi in 4 direzioni (su, giu', destra, sinistra)
- attaccare con la spada
- sparare un proiettile
- caricare una palla di fuoco e spararla rilasciando il tasto
- interagire con oggetti

Ognuna di queste azioni e' rappresentata a schermo da un'animazione differente. Alcune di queste azioni hanno condizioni per poter essere eseguite, oppure possono essere eseguite solo dopo aver compiuto altre azioni; ma in ogni momento il giocatore esegue solo una di queste azioni.

Soluzione Ho risolto il problema utilizzando lo *state pattern* in combinazione con un `PlayerInputSystem`. Infatti, se abbiamo cercato di rispettare il pattern *ecs* il piu' possibile, specialmente cercando di gestire la logica di comportamento delle entita' interamente nei *system* quando possibile, si e' convenuto che non abbia senso forzarlo su ogni aspetto, percio' in questo e in altri casi parte della logica e' stata spostata fuori dai system cercando di semplificare l'aspetto del *behaviour* presente in alcune implementazioni dell'*ecs*.

In questo caso, il giocatore e' un entita' come le altre, definita dall'insieme dei suoi componenti. Il componente che lo distingue maggiormente pero' e' il `PlayerComponent`, che non contiene lui stesso la logica del comportamento del player, ma contiene delle classi che hanno questa logica, cioe' gli *stati* del player (`PlayerState`).

Il `PlayerInputSystem`, che e' il primo *system* a eseguire nel gioco ad ogni loop, processa le entita' che hanno un `PlayerComponent` (lasciando quindi aperta la possibilita' di gestire piu' giocatori). Per semplicita' di spiegazione, assumiamo che l'entita' che rappresenta il giocatore sia una sola. In questo caso, tale entita' viene processata normalmente dal sistema, che ne gestisce il *cambio di stato* in base all'input dell'utente agendo come parte di una *finite state machine*.

Il `PlayerComponent` contiene lo *stato* corrente in ogni momento, quindi viene estratto tale `PlayerState` e interrogato sulla possibilita' di poter effettuare un cambio di stato in base all'`Input`; se possibile, quindi, il `PlayerState` corrente restituisce il prossimo stato calcolato sempre sulla base dell'input e il system procede con la transizione di stato, sostituendo lo stato corrente nel `PlayerComponent` con il nuovo stato calcolato. Se lo stato corrente puo' transitare, allora viene anche chiamato il metodo `execute`, che potrebbe generare nuove entita', ad esempio proiettili o attacchi, e queste vengono poi aggiunte al `World` tramite evento. Il cambio di stato piu' nel dettaglio e' gestito dai metodi `entry` ed `exit` che gestiscono rispettivamente le operazioni da effettuare nei due momenti per quello stato. Infine, il system aggiorna l'animazione (vedi spiegazione animazioni).

Ciascuno stato estende una classe `AbstractState` che fattorizza alcuni metodi dell'interfaccia `PlayerState` come `canTransition()` (che si occupa di controllare che lo stato non sia bloccato in un animazione non cancellabile), `update(double, Entity)` (che si occupa soprattutto di aggiornare il tempo passato nello stato) e altri che sono utili alle sottoclassi come `setAnimationState(String)`. Inoltre, vengono fornite delle implementazioni di default dei metodi d'interfaccia `entry(Entity)`, `exit()` e `execute(Input)`, overridabili a piacimento dalle sottoclassi per definire comportamenti piu' complessi. Invece, il metodo `transition(Input)` e' lasciato astratto da im-

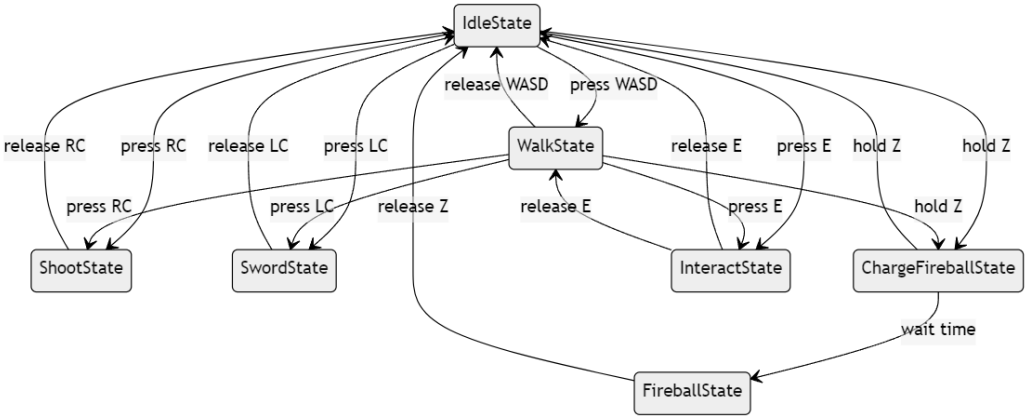

Figure 2.9: Diagramma degli stati del player. LC ed RC indicano rispettivamente il tasto sinistro e destro del mouse. Il player puo' trovarsi solo in uno di questi stati alla volta

plementare per ogni singolo stato poiche' ognuno definisce logiche proprie di transizione verso altri stati, che non descrivo nei dettagli poiche' credo gia' sufficiente esplicate nel diagramma sopra. Infine, ogni stato implementa il metodo `updateAnimation()` settando una specifica animazione (riferimento alla speigazione delle animazioni).

Di seguito gli stati:

- `IdleState` rappresenta lo stato di *idle*, cioe' in cui il giocatore e' fermo, e si occupa solo di disabilitare il movimento in entrata.
- `WalkState` rappresenta lo stato di camminata, e si occupa di abilitare/disabilitare il movimento in entrata/uscita e di settare la direzione del `MovementComponent` coerentemente con l'input dell'utente.
- `SwordState` rappresenta lo stato di attacco ravvicinato, che si occupa di restituire l'entita' che rappresenta l'attacco ravvicinato, tramite relativa factory
- `ShootState` rappresenta lo stato di attacco dalla distanza, che si occupa di restituire l'entita' che rappresenta il proiettile, tramite relativa

factory

- `ChargeFireballState` rappresenta lo stato di carica della fireball, che si occupa di gestire il fatto che si rimanga nello stato fino a quando il giocatore tiene premuto il tasto, ma, se e' passato un certo tempo e solo se il giocatore ha anche rilasciato il tasto, si passa allo stato `FireballState`

- `FireballState` rappresenta lo stato di attacco dalla distanza con una fireball, che si occupa di restituire l'entita' che rappresenta la fireball, tramite relativa factory

- `InteractorState` rappresenta lo stato in cui il player puo' interagire con oggetti che hanno un `InteractableComponent` (power up dello shop, gate ecc.); lo stato si occupa solamente di abilitare/disabilitare l' `InteractorComponent` in entrata/uscita

L'interfaccia `Input` e'ottenibile tramite getter dal `World` e interrogabile sui tasti premuti dall'utente tramite dei semplici getter; in questo modo i vari stati sono in grado di sapere quali azioni sta attualmente cercando di realizzare l'utente.

Ho realizzato una classe `InputListener` che, tramite i metodi di Swing, registra l'input da mouse e tastiera e chiama dei setter su un'istanza di `Input`. Questa istanza viene poi passata al `World` tramite una copia, garantendo in questo modo al modello di gioco di operare con una classe completamente separata dalla View.

Schermata vittoria/sconfitta

Ho realizzato anche una schermata di View, cioe' la schermata di vittoria e sconfitta. Qui, tramite il passaggio di un parametro booleano che indica la vittoria/sconfitta, viene semplicemente visualizzata un'immagine di sfondo e un messagio finale differente.

2.2.2 Elvis Perlika

In questa sezione si approfondirà la parte di *AI* dei nemici ed il *Combat System* tra gli stessi nemici e player.
(Tendenzialmente affronterò la descrzione delle soluzioni con un approccio contrario a quello del mio collega Lorenzo Prati, cioè Top-Bottom)

AI

Obbiettivo Progettare nemici con caratteristche differenti, in modo da offrire al giocatore una varietà di situazioni e strategie da affrontare. Tuttavia, nonostante le differenze tra di loro, l'obbietivo dei nemici deve rimanere quello di eliminare il giocatore.

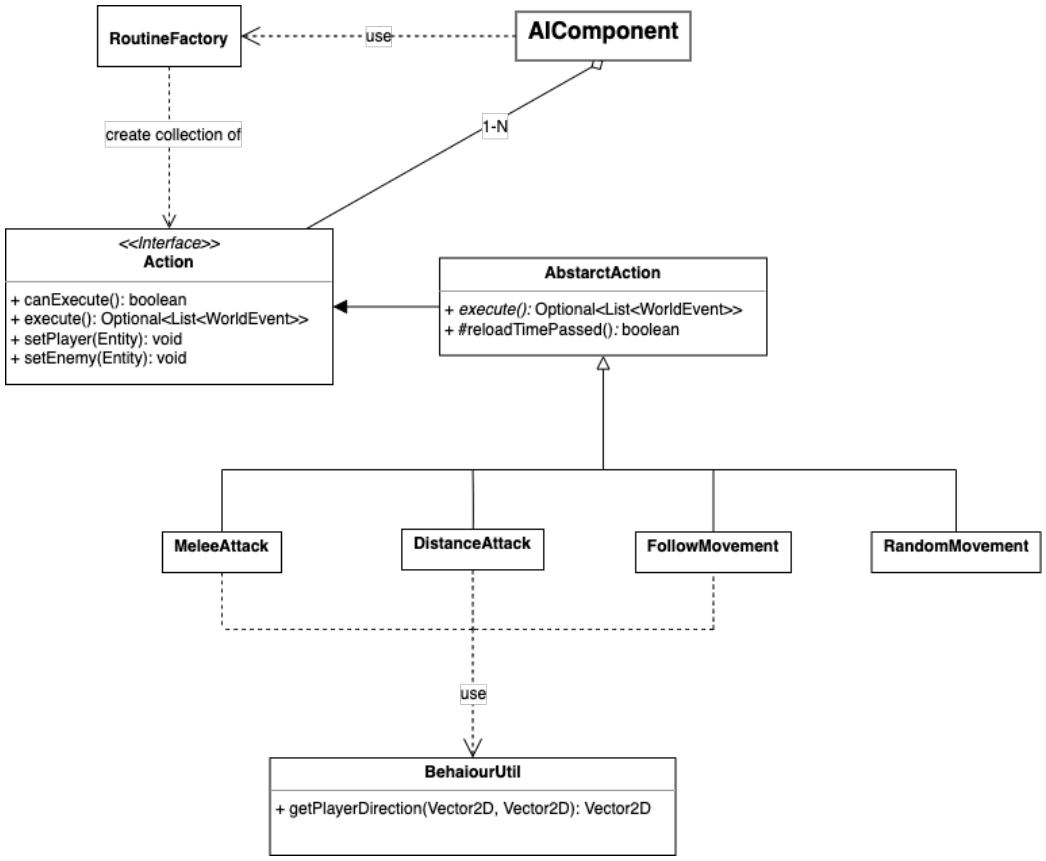

Figure 2.10: Diagramma UML delle Action.

Problema I nemici devono avere comportamenti diversi. Per *Comportamento* si intende "Insieme di azioni, di atteggiamenti con cui l'individuo esterna la propria personalità, rapportandosi agli altri e all'ambiente".

Soluzione Prendendo spunto dal *Pattern Strategy* ho creato l'interfaccia `Action`. Nel mio caso però le Action, a differenza del classico *Pattern Stratgy*, oltre ad eseguire una certa azione valutano se è il caso di eseguirla. Data la definizione di *Comportamento* citata prima, per creare comportamenti

predefiniti ho deciso di implementare la classe `RoutineFactory` che crea collezioni di azioni seguendo il *Pattern Factory Method*.

Esempio: *createShooterRoutine()* restituisce una classica personalità da zombie: segue il player se lo rileva nella sua aggro zone, lo attacca se è abbastanza vicino oppure si muove casualmente nel caso non sia il caso di eseguire le precedenti azioni.

Problema Tutte le Action in questa prima versione del gioco presentano una caratteristca comune: si attivano tenendo conto soltanto della distanza dal player.

Soluzione Ho deciso di creare la classe astratta `AbstractAction` che implementa il metodo *canExecute()* il quale valuta la distanza tra AI e giocatore. Oltre a quel metodo ho sfruttto la classe astratta per fattorizzare altri metodi comuni. E' nota la limitazione che questo sistema provoca ma si è deciso che per un gameplay semplice come il nostro potesse comunque andare bene.

Problema Permettere alle Action di creare nuove entità, ad esempio: attacchi.

Soluzione Per creare nuovi attacchi performati dalle AI nemiche ho creato la classe `EnemyAttackFactory` seguendo il *Pattern Factory Method*.

Nota: Per distinguere gli attachi da altre entità ho creato l'`AttackComponent`. L'`AttackComponent` mantiene anche i dati relativi al danno del attacco.

Ho quindi deciso di far restituire dal metodo *execute()*, presente nelle `Action`, una lista opzionale di `WorldEvent` che l'`AISystem` si occuperà di notificare al `World`. In una Action utile per attaccare sarà logico restituire una lista con eventi di tipo *AddEntityEvent* che prendono come parametro entità create dalla `EnemyAttackFactory`.

Problema La classi utili alla creazione di attacchi come `EnemyAttackFactory` oppure `PlayerAttackFactory` hanno metodi comuni utili al corretto posizionamento e direzionamento degli attacchi.

Soluzione Ho fattorizzato i metodi comuni in una classe astratta `AbstractAttackFactory`. Non ho fatto lo stesso per il metodo *getPlayerDirection()* che permette alle

AI di rintracciare il player perché ha la sua utilità sia in Action di attacco che non; quindi ho inserito il metodo nella classe `BehaviourUtil`.

Conclusione Ho ideato quindi un sistema che permette di creare nuove `Action` basandosi sullo spazio intorno alla AI e sullo scorrere del tempo. Considerato il dominio di gioco semplice, nonostante le chiare litazione che questo sistema presenta, riesce, nel suo piccolo, nella creazione di nuove personalità. Nelle sezione precedenti ho trattato soltanto di AI nemiche ma questo sistema permette anche la creazione di AI assegnabili ad NPC.

Combat System

Problema Evitare che le AI nemiche causino danno a loro AI alleate.

Soluzione Ho sfruttato lo stesso sistema ideato dalla collega Alessandra Versari per la creazione degli Item.
Per dettagli referenzio alla sezione di Design Dettagliato della collega.

HUD

Problema Creare un HUD semplice che mostri vita e monete del giocatore.

Soluzione Avrei potuto costruire un HUD più modulare ma dato il nostro dominio di gioco semplice ho valutato che bastassero pochi semplici metodi per la visualizzazione del HUD.

2.2.3 Emanuele Dajko

Introduzione al concetto di TileMap e il suo ruolo nel gioco

In questa sezione, fornirò una panoramica del concetto di TileMap e del suo ruolo fondamentale nel nostro gioco. La TileMap rappresenta una matrice 2D composta da diverse Tile che, unite insieme, costituiscono la struttura di gioco. Si può pensare alla TileMap come un vero e proprio "puzzle" in cui le Tile sono i pezzi che compongono l'intero quadro del gioco. Ora condividerò le sfide affrontate durante lo sviluppo della TileMap e le soluzioni adottate per superarle.

Parsing delle configurazioni della TileMap

Problema Durante lo sviluppo del nostro gioco, una delle sfide principali che ho affrontato riguardava il parsing delle diverse configurazioni della TileMap per i vari livelli. Era necessario leggere le informazioni specifiche relative a ciascuna TileMap e utilizzarle per creare la corrispondente mappa di gioco.

Soluzione Per ottenere maggiore flessibilità nel design del gioco, ho deciso di adottare una TileMap dedicata per ogni livello. Questa scelta mi ha permesso, anche grazie all'aiuto del collega Elvis Perlika, di definire configurazioni personalizzate per ciascun livello e gestire in modo più efficace le diverse proprietà delle Tile. Ad esempio, ho utilizzato la proprietà "TileMapIdInt" associata a un booleano "WalkableBool", per distinguere le aree camminabili (true) da quelle bloccate (false).
Per affrontare questa sfida, ho sviluppato un parser utilizzando la libreria JDOM (Java DOM parser) per leggere i file XML generati dal programma "Tiled" e gestire il caricamento delle TileMap. Il parser ci ha consentito di estrarre le informazioni necessarie da ciascun file XML e creare la corrispondente TileMap nel nostro gioco. Durante lo sviluppo del parser, ho preso decisioni di design specifiche per garantire un'adeguata gestione delle configurazioni delle TileMap e un corretto caricamento delle informazioni nel nostro gioco. Inoltre, grazie al fatto che rispettando le proprietà definite si può garantire il parsing di diversi tipi di TileMap, sono riuscito a ottenere un riuso parziale del parser stesso.

Note: Durante lo sviluppo del parser per le TileMap, ho affrontato alcune sfide legate alle configurazioni dei file XML attualmente utilizzati. È importante sottolineare che il nostro sistema è stato progettato per gestire le configurazioni delle TileMap conformi alle regole definite. Tuttavia, sono consapevole che il parsing abbia dei limiti derivanti dalla natura dei file XML. Pertanto, per migliorare la flessibilità nella gestione delle configurazioni, prevediamo di estendere il supporto anche ai file JSON in futuro. Durante il processo di sviluppo del parser, ho preso alcune decisioni specifiche per garantire un'adeguata gestione delle configurazioni delle TileMap e un corretto caricamento delle informazioni nel nostro gioco. Ad esempio, ho affrontato il problema della distinzione tra aree camminabili e bloccate utilizzando la proprietà "TileMapIdInt" associata a un booleano "WalkableBool". Questo approccio ci ha consentito di definire chiaramente le diverse proprietà delle TileMap e di gestirle efficacemente durante il parsing.

L'approccio di parsing attuale ha dimostrato di essere vantaggioso nel nostro progetto grazie al riuso del codice esistente per l'analisi e il caricamento delle informazioni. Questo ha comportato un notevole risparmio di tempo nello sviluppo e nella manutenzione del codice. Tuttavia, ho prestato attenzione a mantenere il parser sufficientemente modulare e flessibile per consentire futuri miglioramenti e l'integrazione di nuove tipologie di file come i JSON. Riconosciamo l'importanza di mantenere un equilibrio tra il riuso del codice e la complessità. Ho lavorato diligentemente per evitare di rendere il parser eccessivamente complesso, garantendo al contempo un'adeguata gestione delle configurazioni delle TileMap. Questo equilibrio è fondamentale per garantire l'efficienza e la manutenibilità del sistema nel lungo termine.

Gestione dei layer della TileMap

In questa sezione, introdurrò il concetto di layer nella gestione della TileMap e le migliorie apportate per garantire mappe più complesse ed esteticamente accattivanti.
Introduzione al concetto di layer: Nel contesto della nostra TileMap, per layer si intende uno strato della mappa che rappresenta diversi elementi sovrapposti. Possiamo immaginare la mappa come un sandwich, dove ogni strato corrisponde a una fetta. Ad esempio, il primo layer potrebbe rappresentare la zona percorribile (pavimento), il secondo layer potrebbe essere associato a un blocco di prato (ancora camminabile o meno), e così via, fino all'ultimo strato che costituisce la parte superiore del sandwich.

Problema Tuttavia, l'utilizzo di un solo layer limita le possibilità di creare un design più ricercato e dettagliato nella parte grafica della mappa. Questa limitazione può influire negativamente sull'estetica complessiva del gioco e sulla varietà delle strutture presenti nella mappa.

Soluzione Per superare questa limitazione, ho apportato delle migliorie nella gestione dei layer della TileMap. Ora il sistema è in grado di leggere e gestire ogni singolo layer presente nella mappa corrente, consentendo la creazione di mappe maggiormente estetiche. Uno dei vantaggi dell'approccio è quello di poter cambiare la struttura della mappa senza troppi compromessi. Le migliorie implementate permettono di sovrapporre diversi strati nella mappa, consentendo di aggiungere dettagli e varietà nella struttura complessiva. Ad esempio, posso creare layer separati per rappresentare il terreno, gli elementi di arredamento, gli ostacoli, l'acqua e molti altri. Ogni strato può essere progettato e personalizzato in modo indipendente, offrendo una

maggiore flessibilità e libertà creativa nella creazione delle mappe. Questo nuovo approccio consente di ottenere mappe più dettagliate e realistiche, migliorando significativamente l'aspetto visivo del gioco. Inoltre, grazie alla gestione avanzata dei layer, è possibile modificare la struttura della mappa in modo più flessibile senza compromettere la funzionalità del gioco.

Gestione delle transizioni tra i livelli

Problema La seconda sfida che ho affrontato riguardava la transizione fluida tra i diversi livelli all'interno del gioco. Era fondamentale garantire che il giocatore potesse passare da una stanza all'altra in modo corretto e coerente, mantenendo eventuali statische acquisite durante il corso della partita (come vita, monete e power-up).

Soluzione Per affrontare questa sfida, ho introdotto un meccasimo chiave chiamato "LevelManager". Il LevelManager ha assunto la responsabilità di gestire l'intercambio tra i diversi livelli e coordinare l'avanzamento tra le stanze di gioco. Nel nostro gioco, ho definito diverse tipologie di stanze, tra cui stanze normali, stanze del negozio e stanza del boss. Il LevelManager assicura che il giocatore si sposti correttamente attraverso le diverse stanze, gestendo la transizione e il posizionamento corretto da una stanza all'altra.

Introduzione al concetto di gestione delle stanze all'interno del gioco

In questa sezione, introdurrò il concetto di gestione delle stanze all'interno del gioco, utilizzando il pattern di progettazione chiamato "Strategy".

Problema Durante lo sviluppo del gioco, mi sono trovato di fronte alla sfida di gestire in modo flessibile le diverse strategie delle stanze, al fine di garantire un'esperienza di gioco diversificata e coinvolgente.

Soluzione Per affrontare questa sfida, ho adottato il pattern "Strategy", che mi ha permesso di definire una strategia personalizzata per la generazione e la gestione delle entità all'interno di ciascuna stanza. Questo approccio ha favorito un design modulare e manutenibile, consentendomi di separare le logiche specifiche delle stanze e riutilizzare il codice in modo efficiente. Ho iniziato creando un'interfaccia chiamata "RoomStrategy" per rappresentare la strategia generale delle stanze. Successivamente, ho implementato diverse strategie specifiche, tra cui "NormalRoomStrategy", "ShopRoomStrategy" e "BossRoomStrategy". Ogni strategia ha una logica unica per la generazione

e la gestione delle entità in base alle caratteristiche specifiche della stanza. Per ridurre la duplicazione di codice e favorire il riutilizzo, ho creato una classe astratta chiamata "AbstractRoomStrategy". Questa classe contiene i metodi comuni a tutte le strategie di stanza, ad esempio, il posizionamento del giocatore. Le strategie specifiche, come "NormalRoomStrategy", "ShopRoomStrategy" e "BossRoomStrategy", estendono questa classe astratta e implementano i metodi unici alle loro caratteristiche specifiche. L'utilizzo del pattern "Strategy" mi ha consentito di modellare in modo flessibile le diverse strategie per le stanze del gioco. Ad esempio, la stanza del boss ospita un'entità boss, a differenza delle altre stanze, mentre la stanza del negozio contiene power-up. Ogni strategia di stanza può personalizzare la generazione e la gestione delle entità in base alle sue caratteristiche specifiche. È importante notare che, grazie al pattern "Strategy", ho potuto gestire in modo efficace le diverse strategie delle stanze, semplificando l'aggiornamento delle logiche di generazione e gestione delle entità in base alle caratteristiche specifiche di ciascuna stanza. Questa soluzione ha favorito un alto livello di modularità e facilitato la gestione e l'espansione delle diverse stanze nel mio gioco.

Note: È importante notare che ho cercato di bilanciare la complessità delle strategie delle stanze nel mio gioco. Tuttavia, sono consapevole che potrebbe esserci spazio per miglioramenti futuri, come l'aggiunta di interazioni più complesse tra le entità o meccanismi avanzati. Si tenga presente che la classe astratta "AbstractRoomStrategy" sarà sicuramente soggetta a miglioramenti futuri per ridurre la complessità e migliorare l'organizzazione del codice. In quanto ad ora non è ben organizzata e contiene troppe righe di codice, infatti sarà mio obbiettivo in futuro gestire meglio il tutto. Infatti punto a lavorare sulle mie implementazioni e adattarle man mano all'evoluzione del gioco.

Design Pattern Per gestire le stanze in modo flessibile, ho adottato il pattern di progettazione chiamato "Strategy". Questo pattern mi ha permesso di definire una strategia personalizzata per la generazione e la gestione delle entità all'interno di ciascuna stanza, compreso il posizionamento corretto. Ho creato un'interfaccia chiamata "RoomStrategy" per rappresentare la strategia generale delle stanze e ho implementato diverse strategie specifiche, come "NormalRoomStrategy", "ShopRoomStrategy" e "BossRoomStrategy". In seguito, ho cercato di prendere quanto comune delle tre strategie realizzate, creando una classe "AbstractRoomStrategy", un esempio ne è la generazione del player che è uguale per ogni stanza. Questo ha ridotto di molto la duplicazione di codice all'interno delle tre strategie di stanza, riutilizzando i metodi

senza doverli implementare direttamente. Ogni implementazione delle strategie dettaglia le logiche di generazione e gestione delle entità in base alle caratteristiche specifiche di ciascuna stanza. Ad esempio, la stanza del boss ospita un'entità boss, a differenza delle altre stanze, mentre la stanza del negozio contiene power-up. Utilizzando il pattern "Strategy", sono stato in grado di modellare in modo flessibile le diverse strategie per le stanze, semplificando l'aggiornamento delle logiche di generazione e gestione delle entità in base alle caratteristiche specifiche di ciascuna stanza. Questo ci consente di mantenere un alto livello di modularità e facilita la gestione e l'espansione delle diverse stanze nel nostro gioco.

Note: Ho cercato di bilanciare la complessità delle strategie delle stanze nel mio gioco, cercando di trovare un punto di equilibrio tra la semplicità e la complessità. Sebbene strategie più complesse potrebbero offrire un'esperienza di gioco più sofisticata, ho considerato le caratteristiche specifiche del mio dominio di gioco e ho optato per strategie che soddisfano le esigenze senza aggiungere un'eccessiva complessità. Tuttavia, comprendo che potrebbe esserci spazio per miglioramenti futuri. Sono consapevole che potrei aggiungere ulteriori funzionalità alle stanze, ad esempio, per gestire interazioni più complesse tra le entità o altri meccanismi più avanzati. Sono consapevole inoltre che la classe astratta "AbstractRoomStrategy", abbia troppe righe di codice, ma in fase di realizzazione ho pensato che fosse la soluzione per me più pratica, e semplice da realizzare. Nel complesso, ho scelto di mantenere le strategie delle stanze in una fase adeguata al mio attuale stato di sviluppo del gioco, ma sono aperto a esplorare e implementare strategie più sofisticate in futuro.

Introduzione allo generazione delle entità all'interno delle stanze

Le stanze menzionate in precedenza, ognuna con la propria strategia, posizionano in modo randomico all'interno delle Tile camminabili tutte le entità presenti. La prima delle entità generate, è sempre il player.

Problema Tuttavia, ho incontrato una sfida riguardante il posizionamento corretto delle entità all'interno delle stanze.

Soluzione Per affrontare questa questione, ho introdotto un processo di spawn delle entità basato su un approccio di posizionamento randomico controllato. Ogni entità viene creata prendendo in considerazione le zone camminabili all'interno della stanza, ottenute tramite il metodo automatizzato

"getRandomTile()". Per illustrare meglio il processo, prenderemo come esempio il posizionamento del player. Per garantirene un corretto posizionamento, sono stati sviluppati diversi metodi:

- Il metodo "createPlayer()" crea un'entità player, non tenendo conto delle dimensioni del player stesso. Utilizza un "Set" (insieme) di Tile camminabili e seleziona due coordinate libere per il posizionamento del player.
- Il metodo "createAndPlacePlayer()" considera anche le dimensioni del player e utilizza il metodo "getRandomTile()" per ottenere una zona camminabile casuale. Questo metodo viene utilizzato per creare e posizionare il player all'interno della stanza.
- Il metodo "generatePlayer()" è responsabile di controllare se il player esiste già nella lista delle entità del World prima di crearne uno nuovo. Se il player esiste già, viene utilizzato il player esistente, garantendo così la coerenza delle informazioni, come monete e cuori accumulati durante il gameplay. Se il player non esiste, viene creato un nuovo player utilizzando il metodo sopra descritto.

Questo processo di creazione e posizionamento delle entità viene applicato a ciascuna stanza, garantendo imprevedibilità nel posizionamento delle entità all'interno del gioco. In conclusione, ho migliorato il sistema di spawn delle entità all'interno delle stanze, introducendo un metodo controllato di posizionamento randomico basato sulle zone camminabili. Questo assicura un'esperienza di gioco più interessante e coerente, evitando la perdita di progressi del player e mantenendo la sfida e l'equilibrio del gameplay.

Introduzione alla generazione delle entità con dimensioni maggiori

Nella sotto-sezione precedente ho affrontato la questione della gestione della generazione delle entità di dimensioni "grandi", un nuovo aspetto che ha richiesto *particolare* attenzione durante la fase di progettazione e implementazione.

Problema Uno dei principali ostacoli affrontati è stata la gestione della generazione di entità che occupano più di una singola tile. Fino a questo punto, tutte le entità nel gioco erano di dimensione 1x1 tile, rappresentando un'area quadrata corrispondente a una singola tile. Tuttavia, mi sono trovato nella situazione di dover gestire l'introduzione di entità come "Gate" e "Boss" che possono occupare una superficie maggiore.

Soluzione Per risolvere questo problema, ho introdotto un processo di controllo che verificasse se l'entità in questione, data la sua dimensione, potesse essere posizionata all'interno dell'area designata senza sovrapporsi alle tile occupate da altre entità. Il processo di generazione e posizionamento di entità di dimensioni maggiori ha richiesto lo sviluppo di diversi metodi specifici. Il primo metodo, chiamato "isTileOccupied()", controlla se la tile specificata è occupata da un'altra entità presente nella mappa di gioco. Questo metodo è utilizzato all'interno di un altro metodo chiamato "canAccommodate()", che verifica se l'entità può essere posizionata senza sovrapporsi ad altre entità già presenti.
Per affrontare il posizionamento specifico dell'entità boss, sono stati sviluppati ulteriori metodi:

- Il metodo "canAccommodateTileForBoss()" utilizza il metodo "canAccommodate()" per controllare se l'entità boss può essere posizionata correttamente all'interno dell'area di tile dedicata alla sua generazione.
- Il metodo "getRandomTileForBoss()" viene utilizzato per selezionare casualmente una tile libera che possa ospitare l'entità boss, tenendo conto delle sue dimensioni specifiche
- Successivamente, il metodo "createBoss()" viene chiamato per creare l'entità boss, assegnandole le coordinate di posizione ottenute dal metodo precedente.
- Ora vi è il meotodo "createBoss()", che appunto si occupa di crare l'entità boss, assegnandogli ora una posizione con rispettive coordinate x e y.
- Infine, il metodo "generateAndPlaceBoss()" utilizza tutti i metodi di supporto descritti in precedenza per generare e posizionare l'entità boss all'interno della mappa di gioco.

È importante sottolineare che, per garantire la corretta generazione e posizionamento dell'entità boss, è stato necessario adottare un approccio specifico per selezionare le tile libere e assicurarsi che l'entità non si sovrapponesse ad altre entità esistenti.

Introduzione alle collisioni interne alla TileMap

In questa sezione, fornirò una panoramica del concetto di collisione interna alla TileMap e del ruolo che assume nel nostro gioco. Le collisioni interne alla mappa sono gestite tramite un apposito sistema. Come anticipato nella

sezione precedente, la mappa è costituita da zone "camminabili" e zone "bloccate", grazie alle quali è stato possibile costruire il sistema "MapCollisionSystem". Questo sistema si occupa di rilevare se una zona della mappa è accessibile o attraversabile dalle entità di gioco, come ad esempio il player o i nemici.

Problema La terza sfida affrontata era relativa alle collisioni interne alla mappa, in quanto le entità non avevano limiti e potevano uscire dalla mappa di gioco senza restrizioni.

Soluzione Ho creato un sistema in grado di rilevare le interazioni tra le zone bloccate e le entità di gioco. Ciò ha permesso di definire una sorta di "barriera" che impedisce alle entità di uscire dalle zone predefinite. Quando viene rilevata una posizione al di fuori della zona camminabile, il MapCollisionSystem esegue l'azione appropriata. Ad esempio, se il player cerca di uscire dalla mappa, viene resettato alla sua ultima posizione valida prima di superare i limiti. Allo stesso modo, le entità che rapprestano i proiettili, i quali sono sparati dai nemici o player vengono rimossi quando raggiungono una zona non camminabile.

Introduzione alla vita delle entità di gioco

In questa sezione, forniremo una panoramica del concetto di salute delle entità e il ruolo che essa assume nel nostro gioco. La gestione della salute delle entità è un aspetto fondamentale per determinare la loro vitalità e partecipazione attiva nel gameplay. La salute di un'entità nel nostro gioco è rappresentata dal componente HealthComponent. Un'entità è considerata "viva" se il valore del suo HealthComponent è superiore a zero, mentre è considerata "morta" se il valore è uguale o inferiore a zero.

Problema All'inizio, non era presente un controllo sulla salute delle entità sconfitte, e di conseguenza rimanevano in gioco nonostante fossero state eliminate.

Soluzione Per affrontare questo problema e garantire un'esperienza di gioco più realistica, ho introdotto un componente dedicato chiamato `HealthComponent`. Questo componente viene utilizzato da tutte le entità di gioco, inclusi il player, i nemici e il boss, per gestire la loro salute. Ho sviluppato anche un sistema specifico chiamato `"CheckHealthSystem"` per gestire il controllo della salute delle entità e rimuovere quelle che muoiono durante i combattimenti.

Il CheckHealthSystem viene eseguito in sincronia con l'aggiornamento del gioco, controllando costantemente lo stato di salute di ciascuna entità. Grazie all'implementazione del CheckHealthSystem e del RemoveEntityEvent, le entità sconfitte vengono correttamente rimosse dal gameplay. I sistemi del gioco che gestiscono le entità catturano l'evento di rimozione e intraprendono le azioni necessarie per eliminare l'entità dallo schermo, interrompendo le sue interazioni con gli altri elementi di gioco. Questa soluzione ha migliorato notevolmente l'esperienza di gioco, rendendola più realistica. Ora i giocatori possono concentrarsi sulle entità ancora vive e non vengono più disturbati dalla presenza di entità sconfitte che rimangono in gioco.

Introduzione alle nuove entità create

Una volta completato quando sopra riportato, mi sono dedicato alla creazione di nuove entità, che ho poi usato per scopi specifici. Ne è un esempio il boss per utilizzato per la realizzazione della boss fight.

Problema Uno dei nostri obiettivi era ampliare il gioco introducendo nuove entità.

Soluzione Per affrontare questa sfida, ho utilizzato il design pattern Factory Method. In particolare, ho realizzato la classe "BossFactory" che implementa il metodo "createBoss()" per la creazione dell'entità boss. Questo approccio mi ha fornito diversi vantaggi. Il pattern Factory Method permette di incapsulare la logica di creazione dell'entità all'interno dei metodi stessi, noti come factory methods. In questo modo, possiamo creare diversi tipi di entità senza dover esporre i dettagli della loro costruzione all'esterno. Inoltre, il pattern facilita l'estensione futura aggiungendo nuovi metodi di factory per creare tipi aggiuntivi di entità (ad esempio ci sono anche i minions). La classe BossFactory, insieme alla classe "GenericFactory", che adotta lo stesso pattern, mi hanno portato alla creazione delle entità Boss e Minion (nella prima), così come dello shop-keeper (nella seconda), in modo flessibile ed estensibile.

Introduzione alla boss fight

Alla proposta di progetto, avevamo messo tra le scelte opzionali, quella della realizzazione della boss fight, ho dunque deciso di occuparmene, per rendere il gioco più completo e giocabile. Introuduciamo ora la Boss entity, la quale è presente nell'ultima delle stanze, con stanza dedicata, essa è l'entità nemica più potente (da qui si ha anche l'introduzione degli item e power-up).

Problema Il problema da affrontare era la realizzazione di un combattimento coinvolgente tra l'entità player e il boss.

Soluzione Per consentire il combattimento nel nostro gioco, abbiamo bisogno di definire delle azioni specifiche per le entità coinvolte. Nel caso del boss, ho adottato nuovamente il design pattern Factory Method per gestire la creazione delle sue azioni.
Le azioni che il boss può compiere sono le seguenti:

- Attacco corpo a corpo speciale: un attacco con un'area abbastanza grande, attivabile solo quando il boss raggiunge una soglia di vita specificata.
- Incremento di velocità: l'aumento della velocità del boss, attivato solo quando il boss si avvicina a una soglia di vita specifica. Questo bilancia lo scontro nel caso in cui il giocatore abbia acquisito numerosi power-up.
- Incremento di vita: l'aumento della vita del boss, attivato quando il boss raggiunge una determinata soglia di vita. Questo permette al boss di recuperare energia quando il giocatore si concentra principalmente sull'attacco a distanza.

La creazione delle azioni speciali del boss avviene tramite l'utilizzo del design pattern Factory Method. I metodi, come ad esempio "createBossMeleeAttack()", sono responsabili della creazione delle entità di attacco specifiche utilizzate solo dal boss. Questo approccio incapsula la logica di creazione e restituisce un'istanza dell'oggetto Entità che rappresenta l'attacco stesso. L'utilizzo del Factory Method centralizza la creazione delle entità di attacco, consentendo flessibilità e disaccoppiando il codice "client" dai dettagli specifici dell'implementazione dell'attacco. Ciò promuove il principio di incapsulamento e rispetta il principio "Open/Closed", consentendo l'aggiunta di nuovi tipi di attacco senza modificare il codice esistente.
Le azioni del boss sono gestite dalla classe "EnemyRoutineFactory", che implementa il design pattern Factory Method. I metodi, come "createBossRoutine()", sono responsabili della creazione delle routine logiche che guidano il comportamento del boss durante la boss fight. Questo approccio consente l'incapsulamento della logica di creazione delle azioni comportamentali e fornisce flessibilità ed estensibilità senza modificare il codice esistente.
Ritengo che l'implementazione sopra descritta sia sufficiente per garantire uno scontro divertente ma non troppo impegnativo tra il giocatore e il boss, offrendo un'esperienza di gioco più completa e coinvolgente.

Note: Sono consapevole che avrei potuto realizzare delle action più complesse, ma ho ritenuto che delle azioni semplici fossero sufficienti per soddisfare le esigenze del nostro gioco. Date le caratteristiche del dominio di gioco, l'implementazione di azioni complesse avrebbe potuto aggiungere una complessità eccessiva senza apportare un valore significativo all'esperienza di gioco complessiva.

Introduzione ai nuovi component realizzati

Nella mia implementazione del gioco, ho sviluppato due component specifici: il Boss e lo Shop-Keeper component. Per rendere queste entità interattive e funzionali agli eventi, ne ho creato dei component dedicati. Il componente del Boss ha una funzione importante nel riconoscere lo stato della partita, questo evento non è però stato realizzato da me. Quando il Boss viene sconfitto, viene lanciato l'evento "Win", seguito dalla visualizzazione di una schermata di vittoria e dalla conclusione del gioco. Questo componente, che funge da "tag", permette al gioco di rilevare la fine della sfida con il Boss e di fornire una conclusione ai giocatori.
Il componente dello Shop-Keeper, invece, ha una funzionalità differente rispetto alle altre entità normali, come i nemici comuni (escludendo il Boss e il giocatore). Nelle stanze di gioco, di solito è necessario eliminare tutti i nemici per poter accedere al gate e passare al livello successivo. Tuttavia, nello specifico caso dello Shop-Keeper, che appare solo nella stanza del negozio, il gate può essere utilizzato fin da subito, in quanto lo Shop-Keeper è escluso dalle entità che devono essere eliminate. Questa eccezione è possibile grazie all'introduzione del componente specifico dello Shop-Keeper (tramite uso del suo "tag"). I componenti del Boss e dello Shop-Keeper non implementano una logica interna complessa, ma svolgono il ruolo di "tag" o riconoscimento delle entità.

Problema Generare l'evento di fine partita.

Soluzione Per gestire questo problema, ho creato il componente necessario ad un sistema che, al momento della sconfitta del Boss, genera l'evento "Win". Questo evento viene rilevato dal sistema del gioco, che avvia una sequenza di azioni, tra cui la visualizzazione di una schermata di vittoria e la conclusione del gioco. Dettagli specifici sulla gestione di questo evento sono stati forniti precedentemente.

Problema Gestire il cambio di livello anche con lo Shop-Keeper che precedentemente non era costituito da un componente.

Soluzione Per permettere il cambio di livello con la presenza dello Shop-Keeper, ho introdotto un componente dedicato a questa entità. Questo componente viene utilizzato per all'interno del predicate BiFunction "useGate", la presenza del suo component, è usata come filtro, quando tutti i nemici oppure lo shop-keeper è presente, allora avviene l'evento "ChangeLevelEvent" non realizzato da me. I dettagli specifici sono stati realizzati all'interno della classe `"InteractableObjectFactory"`. Con queste migliorie, i componenti del Boss e dello Shop-Keeper diventano parte integrante del gameplay, offrendo funzionalità specifiche e consentendo una gestione più sofisticata degli eventi e delle transizioni di livello nel gioco.

Menu tutorial

Problema Il mio obiettivo era quello di fornire al giocatore una schermata di tutorial iniziale, pre-game, che spiegasse i tasti e le regole di gioco in modo chiaro e comprensibile. Volevo garantire che il giocatore fosse in grado di godersi appieno il gameplay fin dal primo momento.

Soluzione Per affrontare questa sfida, ho creato una schermata di tutorial che si attiva al primo click del pulsante "play" nel menu principale. Ho voluto utilizzare delle immagini catturate dal gameplay stesso, poiché ciò rende le istruzioni più chiare e immediate. Ho abbinato a ogni immagine un testo esplicativo che illustra il modo corretto di eseguire le azioni nel gioco. Ad esempio, mostriamo i tasti utilizzati per il movimento del giocatore e spieghiamo le regole per cambiare stanza o utilizzare il gate (portale). La disposizione delle immagini e del testo è stata organizzata in modo semplice e intuitivo per facilitare la comprensione del giocatore. Ho cercato di mantenere il tutorial conciso e diretto al punto, evitando informazioni superflue che potrebbero confondere o annoiare il giocatore. In conclusione, la schermata di tutorial che ho implementato nel menu principale fornisce al giocatore un'introduzione semplice ma completa ai tasti e alle regole di gioco. Ciò permette al giocatore di immergersi nel gameplay fin dal primo momento, senza troppe difficoltà, sfruttando appieno le meccaniche di gioco e godendo dell'esperienza offerta dal nostro gioco.

Note: Sono consapevole che avrei potuto costruire un Menu tutuorial più funzionale, ad esempio un tutorial interattivo, ma dato il nostro dominio di gioco piuttosto semplice, ho valutato che bastassero pochi e semplici metodi per la visualizzazione a schermo dei comandi e delle regole di gioco.

2.2.4 Alessandra Versari

Items

Problema Con item si intende un oggetto di gioco che verrà raccolto a seguito del passaggio del player sopra ad esso. Gli item presenti nel nostro gioco sono cuori e monete. Ciascun item una volta entrato in contatto con un'entità deve prima verificare che si tratti del player ed effettuare i controlli necessari prima di applicare il proprio "effetto" (cioè la conseguenza/reazione che l'item avrà sull'entità che ha colliso con esso). Di seguito le principali caratteristiche degli items presenti nel nostro gioco:

- Gli items "cuore" permetteranno di aumentare la vita corrente del player, ma ciò verrà fatto solo a seguito del controllo sulla vita corrente: l'item verrà raccolto se e solo se la vita corrente è minore della vita massima che il player può avere. Questo tipo di item è disponibile in ogni stanza, escludendo lo Shop.

- Gli items "moneta", anch'essi presenti in ciascun livello (shop escluso), permetteranno di aumentare l'ammontare delle monete raccolte dal player. Questo tipo di item, oltre a verificare che l'entità che ha colliso con esso sia il player, non farà ulteriori controlli. Lo scopo degli items "moneta" è quello di consentire al player, l'acquisto di power up (che tratterò in seguito) all'interno della stanza Shop.

Soluzione Per realizzare quanto descritto ho deciso di creare due componenti principali: HealthComponent e CoinPocketComponent, appartenenti alla lista di componenti del player, che permettono di consultare e modificare vita corrente, vita massima e monete raccolte. Tramite i getter e i setter, in questo modo, posso creare gli "effetti" degli item, all'interno dell'ItemFactory. Ogni item è dotato di un componente detto ItemComponent che oltre a rendere riconoscibili gli item, memorizza una Bifunction che corrisponde a quello che fin'ora ho definito "effetto". La Bifunction in questo caso prende come argomento l'entità che ha colliso con l'item e una lista di component. Nel nostro gioco, al momento, solo il player ha la possibilità di raccogliere items, ma nel caso in cui volessimo rendere questi items raccoglibili anche ai nemici, sarebbe possibile farlo, passando come argomento alla Bifuntion una lista contenente i componenti identificativi di player e nemici (ossia PlayerComponent e AIComponent). Inizialmente avevo scelto di creare gli effetti utilizzando il Factory Method perché pensavo potesse rendere più riutilizzabili gli effetti e velocizzarne la creazione, ma il risultato era un insieme di classi Factory, molto simili tra loro e la cui unica differenza era operare

su componenti diversi (una factory per gli effetti che modificavano la vita, un'altra per quelli che modificavano l'ammontare delle monete e così via). Inoltre siccome gli effetti alla fine sono praticamente solo incrementi e decrementi, mi è sembrato più sensato utilizzare interfacce funzionali e lamba per crearli all'interno della factory. Per quanto riguarda le interfacce funzionali, ho deciso di utilizzare le Bifuction così da poter restituire un boolean che permette di capire se l'effetto è stato applicato o meno e se quindi è necessario rimuovere l'item.

Pattern utilizzati Nella classe ItemFactory è stato utilizzato il Factory Method.

InteractableObjects

Problema Con interactable objects si intende tutti gli oggetti di gioco che per essere utilizzati necessitano dell'interazione del giocatore. A differenza degli items infatti, la collisione con l'oggetto in questo caso non è sufficiente, è necessario premere in tasto E una volta posizionato il player sull'oggetto. Gli oggetti interactable presenti nel nostro gioco sono i power-up e il gate. I power-up sono potenziamenti che il player può acquistare nella stanza shop pagando il loro prezzo. Essi si suddividono in:

- Power-up vita: permette di aumentare la vita massima del giocare.
- Power-up velocità: permette di aumentare la velocità del giocatore.

Il gate invece è semplicemente il portale che permette di accedere al livello successivo.

Soluzione Per realizzare tutto ciò ho creato i vari InteractableObjects all'interno dell'interactableObjectsFactory assegnando a ciascuno di essi i propri componenti. Ognuno di questi oggetti è distinguibile grazie all'InteractableComponent che memorizza il loro "effetto" mediante un Bifunction che prende come argomenti un'entità e il world (necessario affinchè alcuni di essi possano lanciare WorldEvent, per esempio il gate come effetto lancia l'evento ChangeRoomEvent()) e restituisce un booleano che permette di capire se l'effetto è stato applicato o meno. Sempre all'interno della factory ho implementato anche gli "effetti" tramite Bifunction che poi utilizzano altre interfacce funzionali (BiPredicate,Predicate e BiConsumer) per effettuare i controlli necessari in modo da poterli riutilizzare all'interno di "effetti" diversi se necessario. Ho deciso di implementare tutto questo all'interno della factory (come nel caso degli items) perché anche in questo caso si trattava di qualche incremento o

decremento di interi contenuti in Component diversi e qualche controllo in più. Anche in questo caso inizialmente avevo tentato di utilizzare il Factory method per poi rendermi conto che non era la strada migliore perché il risultato erano una serie di classi molto simili e quindi codice ripetitivo, inoltre le factory non erano molto riutilizzabili siccome questo tipo di effetti devono essere applicati sempre e solo sul player. Ho creato infine l'InteractableSystem che si occupa di tutti gli oggetti che hanno l'InteractableComponent e controlla se alcune entità hanno colliso con essi. Nel caso affermativo, se tutti i controlli vengono superati (es. l'effetto del gate viene applicato solo se il player sta interagendo con esso, se tutti i nemici sono stati eliminati e quindi la Bifunction ritorna true) l'oggetto viene rimosso tramite il WorldEvent RemoveEntityEvent().

Pattern utilizzati È stato utilizzato il Factory Method nella classe InteractableObjectFactory.

Animazioni

Problema Si vuole realizzare un gioco le cui entità sono animate. Ogni entità può assumere stati diversi: può stare semplicemente ferma (idle), può attaccare in modi differenti (con la spadata, sparando...), può subire un danno etc.

Soluzione Siccome ciascuna entità ha un numero di sprite diverso, con dimensioni differenti e vari stati ho deciso di realizzare un componente chiamato AnimationComponent con lo scopo di mantenere tutti i dati neccassari per l'animazione (solo interi e stringhe). Ogni AnimationComponent contiene una mappa riguardante il proprio tipo di entità (es. l'animation component del player avrà una mappa che conterrà solo i dati riguardanti il player) ricavata da una mappa generica creata in AbstractFactory grazie alla lettura di un file di configurazione yaml. L'AnimationSystem elabora poi i dati presenti nell'AnimationComponent, semplicemente incrementando o resettando degli interi, non contiene quindi informazioni di view. Questi dati vengono poi passati dal world alla view sotto forma di GraphicInfo, ossia un semplice oggetto contenente tutte le informazioni necessarie per il disegno (come posizione, numero di immagine da utilizzare, numero dello sprite da ritagliare etc.) e che viene aggiunto alla lista di entità da disegnare mantenuta nella view. Tutte le immagini necessarie per le animazioni e per il disegno della mappa di gioco vengono caricare grazie ad una classe dedicata a questo: il Resource loader. Per non limitare la scelta degli sprites utilizzando tutti sprite con le stesse dimensioni, ho deciso di realizzare un secondo file di

configurazione contenente altezza e larghezza del singolo sprite di ciascuna entità, che verranno poi utilizzate per eseguire il ritaglio dell'immagine da disegnare.

Menù di gioco

Problema L'obiettivo era creare varie schermate, più nello specifico una schermata iniziale contente il menù, una schermata Options che permette di modificare la risoluzione, una schermata che permette di interrompere la partita e riprendere dal punto in cui è stata interrotta e una schermata finale che semplicemente mostra il risultato della partita.

Soluzione Per fare ciò ho deciso di creare una classe AbstractScreen così da ridurre il più possibile la ripetizione di codice. Questo è stato possibile perché tutte queste schermate utilizzano il GridBagLayout come layout più esterno. In questo modo nelle classi che implementano i vari menù e schermate compaiono solo gli elementi differenti e non utilizzati negli altri menù o schermate.

Chapter 3

Sviluppo

3.1 Testing automatizzato

Tutti i test sono stati realizzati con JUnit. Non sono stati testati i system.

Lorenzo Prati

Test delle entita' e dei componenti Ho realizzato una classe `EntityTest`, che contiene un metodo di test che si occupa di creare un'entita' di prova usando l'`EntityBuilder` e successivamente controlla il corretto funzionamento dei metodi dell'interfaccia `Entity`, soprattutto quelli che servono a manipolare i `Component`.

Test del player, input e stati Ho realizzato una classe `PlayerTest` che prima si occupa di inizializzare l'entita' che rappresenta il giocatore, tramite relativa factory, poi sono presenti due metodi di test: il primo controlla la corretta inizializzazione del player, il secondo testa il funzionamento degli stati e dell'input.

Alessandra Versari

Test degli effetti degli items Ho realizzato la classe `ItemTest`, che contiene due metodi, uno per testare l'effetto dell'item cuore e l'effetto dell'item moneta, confrontando i valori contenuti nei component dei player dopo che l'effetto è stato applicato

3.2 Metodologia di lavoro

Lorenzo Prati

- package entity
  - interfaccia `Entity`
  - classe `EntityImpl`
  - classe `EntityBuilder`
  - classe `GenericFactory` solo per quanto riguarda il metodo di creazione del player
- package component
  - interfaccia `Component`
  - classe `PositionComponent`
  - classe `MovementComponent`
  - classe `PlayerComponent`
  - classe `CollisionComponent`
  - classe `InteractorComponent`
- package systems
  - interfaccia `GameSystem`
  - classe `AbstractSystem`
  - classe `PlayerInputSystem`
  - classe `MovementSystem`
  - classe `CollisionSystem`
  - classe `PhysicsSystem`
  - classe `ClearCollisionSystem`
- package core
  - interfaccia `Engine`
  - classe `EngineImpl`
  - interfaccia `World`
  - classe `WorldImpl`
- package input

  - interfaccia `Input`
  - classe `InputImpl`
- package events
  - interfaccia `WorldEvent`
  - classe `AddEntityEvent`
  - classe `RemoveEntityEvent`
  - classe `ChangeLevelEvent`
- package logic
  - logic.player
    - interfaccia `PlayerState`
    - classe `PlayerState`
    - intero package player.states
  - logic.collision
    - interfaccia `BodyShape`
    - classe `RectBodyShape`
    - classe `CircleBodyShape`
  - logic.util
    - classe `DirectionUtil`
- package view
  - classe `InputListener`
  - classe `ResultScreen`

Elvis Perlika

- package entity
  - factories
    - classe `AbstractFactory`
    - classe `AbstractAttackFactory`
    - classe `EnemyAttackFactory`
    - classe `EnemyFactory`
    - classe `PlayerAttackFactory`

- package component
  - classe `AIComponent`
  - classe `AttackComponent`
  - classe `MeleeComponent`
  - classe `BulletComponent`
- package systems
  - classe `AISystem`
  - classe `CombatSystem`
- package logic
  - logic.AI
    - interfaccia `Action`
    - classe `AbstarctAction`
    - classe `BehaviourUtil`
    - classe `DistanceAttack`
    - classe `FollowMovement`
    - classe `MeleeAttack`
    - classe `RoutineFactory`
    - classe `FollowMovement`
- package view
  - interfaccia `HUD`
  - classe `HUDImpl`

Alessandra Versari

- Package components
  - classe `AnimationComponent`
  - classe `CoinPocketComponent`
  - classe `HealthComponent`
  - classe `InteractableComponent`
  - classe `ItemComponent`
- Package entity

  - classe `InteractableObjectFactory`
  - classe `ItemFactory`
- Package systems
  - classe `AnimationSystem`
  - classe `InteractableSystem`
  - classe `ItemSystem`
- Package view
  - classe astratta `AbstractScreen`
  - classe `GraphicInfo`
  - classe `HomeScreen`
  - classe `OptionScreen`
  - classe `PauseScreen`
  - classe `ResourceLoader`
  - interfaccia `Scene`
- Package resources
  - file di configurazione `animations.yaml`
  - file di configurazione `spritesDimensions.yaml`

Parti sviluppate in collaborazione

- classe `SceneImpl`
- interfaccia `MainWindow`
- classe `MainWindowImpl`

3.3 Note di sviluppo

Lorenzo Prati

- uso della libreria jts (Java Topology Suite) sia per la classe `Vector2D` che e' usata in tutto il progetto, sia per per creare facilmente forme geometriche di interfaccia `Polygon` che tra l'altro forniscono anche metodi per controllare le intersezioni tra poligoni.

Esempio: https://github.com/LorenzoPrati/OOP22-dim-hol/blob/587a03e4db001028dbcf5649ed6af5807a3ef155/src/main/java/dimhol/logic/collision/RectBodyShape.java#L51-L62

(link alla pagina github della libreria)

- reflection, stream e lambda nella gestione delle `Entity`

  Esempio: https://github.com/LorenzoPrati/OOP22-dim-hol/blob/5d2b44fc5bf5fa50b6f0215695d2a5bac89cab85/src/main/java/dimhol/entity/EntityImpl.java#L61-L89

- stream e lambda utilizzati anche in `RemoveEntityEvent`,in `WorldImpl`, nel metodo template di `AbstractSystem`, nel `CollisionSystem` e in altri punti del codice

- sebbene faccia parte di java.util, menziono l'utilizzo della classe UUID per generare id per le entita'

- Optional utilizzati sia in `PositionComponent` che nei vari stati del player

Alessandra Versari

- uso della libreria SnakeYAML sia per la classe AbstractFactory sia per la classe ResourceLoader. Esempio:

- reflection, stream e lambda utilizzate in varie classi. Seguono alcuni esempi: Esempio: utilizzo reflection in ItemSystem https://github.com/LorenzoPrati/OOP22-dim-hol/blob/78f510d43dec8c4b5855980ccefefa539f23f48
  src/main/java/dimhol/systems/ItemSystem.java#LL20C8-L20C8 Esempio: utilizzo lambda in InteractableObjectsFactory https://github.com/LorenzoPrati/OOP22-dim-hol/blob/cc4664659311592c5d0f16e6f7d332ba8f5b813
  src/main/java/dimhol/entity/factories/InteractableObjectFactory.java#L29 Esempio: utilizzo stream in AnimationComponent https://github.com/LorenzoPrati/OOP22-dim-hol/blob/cc4664659311592c5d0f16e6f7d332ba
  src/main/java/dimhol/components/AnimationComponent.java#LL64C4-L72C1

Chapter 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Lorenzo Prati

Mi ritengo complessivamente soddisfatto del lavoro svolto insieme ai miei compagni di gruppo, anche se ammetto che il percorso che ci ha portati ad arrivare a questo punto del progetto non e' stato facile. Inizialmente si era scelto di utilizzare il pattern architetturale MVC, ma a causa piu' che altro di una nostra mal comprensione delle dinamiche e caratteristiche del pattern, era risultato molto difficile pensare a una struttura adeguata tanto che si e' deciso di passare alla deadline successiva ricominciando quasi da capo. Infine siamo passati al *pattern ecs* per gestire la logica del gioco, di cui mi sono molto informato personalmente tramite ricerche online. Visto che mi sono occupato di realizzare la base dell'architettura dell'*ecs*, devo dire che il cercare di realizzare un'implementazione personale e sicuramente molto semplice di un pattern noto ma praticamente mai implementato a un livello *semplice* e adatto al progetto o addirittura non molto usato con il linguaggio Java (infatti spesso versioni dell'*ecs* che si trovano online sono realizzate in C++ o altri linguaggi), e' stata una bella sfida. La discussione con i docenti anche e' stata importante per chiarirci le idee su a cosa dovessimo effettivamente dare priorita'. Non escludo in futuro di tornare a lavorare a questo progetto, anche se per il momento ritengo essere state realizzate tutte le feature piu' importanti che ci eravamo prefissati.

Alessandra Versari

Sono abbastanza soddisfatta del risultato raggiunto insieme al gruppo soprattutto considerando le varie difficoltà iniziali. Credo ci abbia insegnato molto

soprattutto su come affrontare progetti di gruppo in futuro. Ammetto di aver avuto molte difficoltà all'inizio sia perchè era un progetto diverso dal solito (molto più lungo e complesso), sia perchè bisogna svilupparlo nel secondo periodo, mentre ci sono lezioni di altri corsi che portano via molto tempo e per quanto mi riguarda è stato parecchio complicato riuscire a incastrare tutto. Al momento non credo che continuerò a lavorarci per mancanza di tempo e per dare spazio alle altre materie, ma non escludo di farlo in futuro.

4.2 Difficoltà incontrate e commenti per i docenti

Lorenzo Prati

Le difficolta' maggiori che ho incontrato sono personali: il lavoro di gruppo, la gestione del tempo, la collaborazione e la discussione sul codice insieme ad altre persone, che erano tutte cose che non avevo mai affrontato prima in questo modo. Per quanto riguarda il corso, sarebbe stato utile vedere e discutere un esempio di un progetto di questa *portata* in modo tale da avere dritte e consigli dai docenti fin da subito; comunque, ritengo che le basi forniteci a lezione sul linguaggio Java e sui pattern siano piu' che buone e assolutamente adatte per permetterci di costruire, da soli, un progetto di questo tipo.

Elvis Perlika

Il lavoro di squadra è probabilmete lo scoglio più grande ma non insommortabile. Non saprei ben descrivere altre difficoltà incontrate se non quelle legate alle mie personali competenze.

Alessandra Versari

Essendo un progetto abbastanza grande rispetto a ciò a cui siamo abituati mi sono trovata un po' spaesata soprattutto all'inizio e la collaborazione con i miei compagni è stata fondamentale. Sicuramente ciò che ho ritenuto più difficile è la scelta di un'architettura adeguata a ciò che avevamo in mente di sviluppare. Inoltre il fatto di aver dovuto ricominciare il progetto quasi a ridosso della scadenza scelta (a causa di nostre scelte errate) e aver dovuto lavorarci per più mesi del previsto, facendo sempre fatica a trovare il tempo per le altre materie e altri impegni mi ha un po' buttato giù. Mi sarebbe piaciuto vedere qualche progetto degli anni precedenti a lezione, così

da sentirmi magari più pronta all'inizio del progetto, ma complessivamente ritengo che le competenze acquisite siano sufficienti.

Chapter 5

Guida utente

Tutti i comandi di gioco e le regole di base sono spiegate in una schermata tutorial che compare la prima volta che si gioca ad ogni apertura dell'applicazione.

Inoltre, e' possibile attivare una **Modalita' DEBUG**. Per farlo, andare in *HOME* $\rightarrow$ *OPTIONS* $\rightarrow$ spuntare il check *DEBUG MODE*. Questa modalita' puo' essere utile, ad esempio ai docenti per motivi di testing, per visitare facilmente tutti i tipi le stanze e arrivare molto velocemente al boss.

Chapter 6

Esercitazioni di laboratorio

Bibliography