Relazione Progetto OOP
"Dimension Holiday"

Lorenzo Prati, Elvis Perlika
Emanuele Dajko, Alessandra Versari

June 5, 2023

Contents

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 che è stato trasportato in un altro mondo dove dovrà esplorare un dungeon e affrontare un boss per tornare alla propria dimensione.

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 o lanciare proiettili energetici. 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 avrà ucciso tutti i nemici della stanza corrente compariranno dei reward casuali (cuori, monete ecc.) e il portale che ti trasporterà nella prossima stanza.

- 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:

- *meele*, ovvero ravvicinato, usando una spada e tramite il tasto sinistro del mouse
- dalla distanza, usando un proiettile energetico e tramite il tasto destro del mouse

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 e allo stesso tempo non entrarci in contatto, cosa che comportera' ulteriore danno. 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 (*dungeon*) 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 solo il nemico boss. Le stanze normali saranno intercambiate randomicamente tra un pool prescelto di mappe create a mano, mentre le stanze shop e boss saranno uniche.

La difficoltà sara' gestita in modo tale che proseguendo nel dungeon risulti piu' difficile il gioco, ad esempio facendo comparire piu' nemici nelle stanze

oppure nemici piu' forti. Questo aumento della difficoltà sara' compensato dai potenziamenti che il giocatore potra' acquistare nello shop, che comparira' dopo un numero costante di stanze normali superate.

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 *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.

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'.

- `System`: 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.

- `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.

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

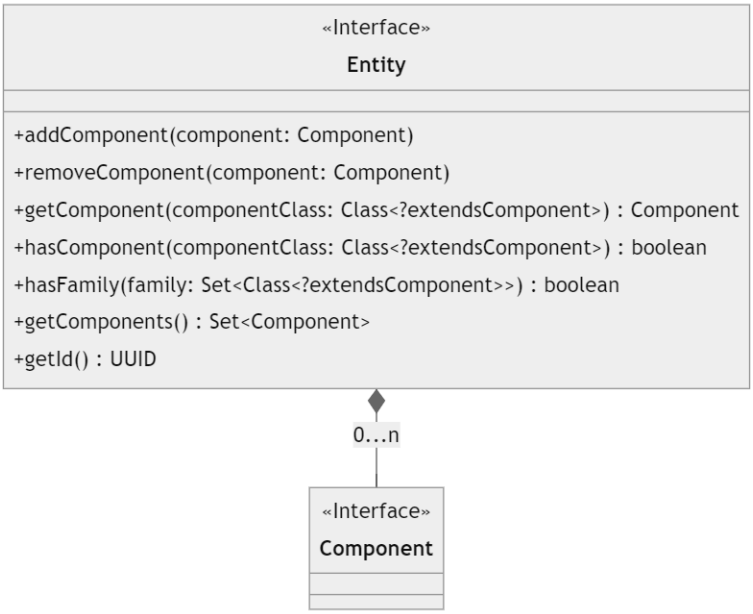

Figure 2.1: Diagramma UML della relazione tra Entity e Component

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. (vedi factories)

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 riconoscere su quali entita' e' in grado di operare. 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 ecc.

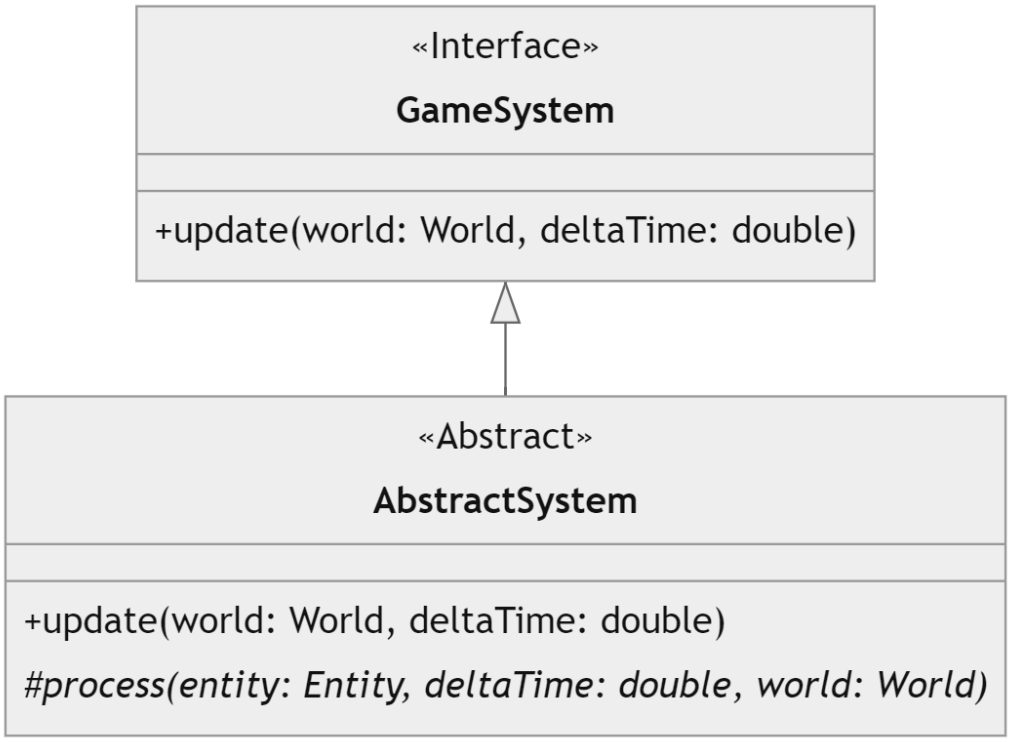

Figure 2.2: Diagramma UML dei *system*

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 sulle 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' possibile 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

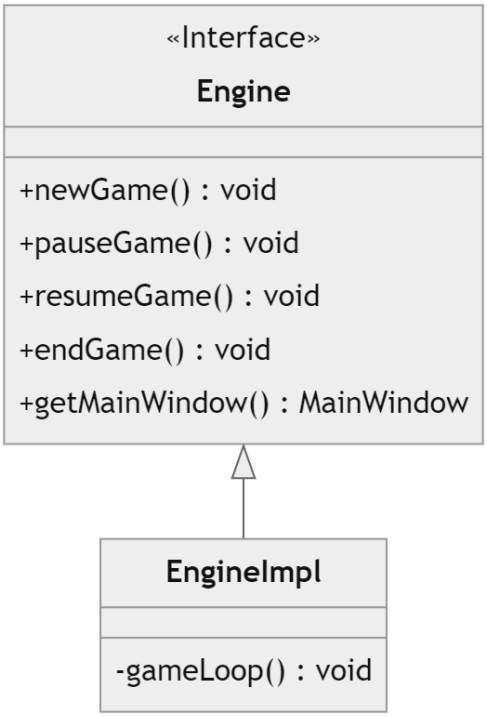

Figure 2.3: Diagramma UML dell'Engine

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

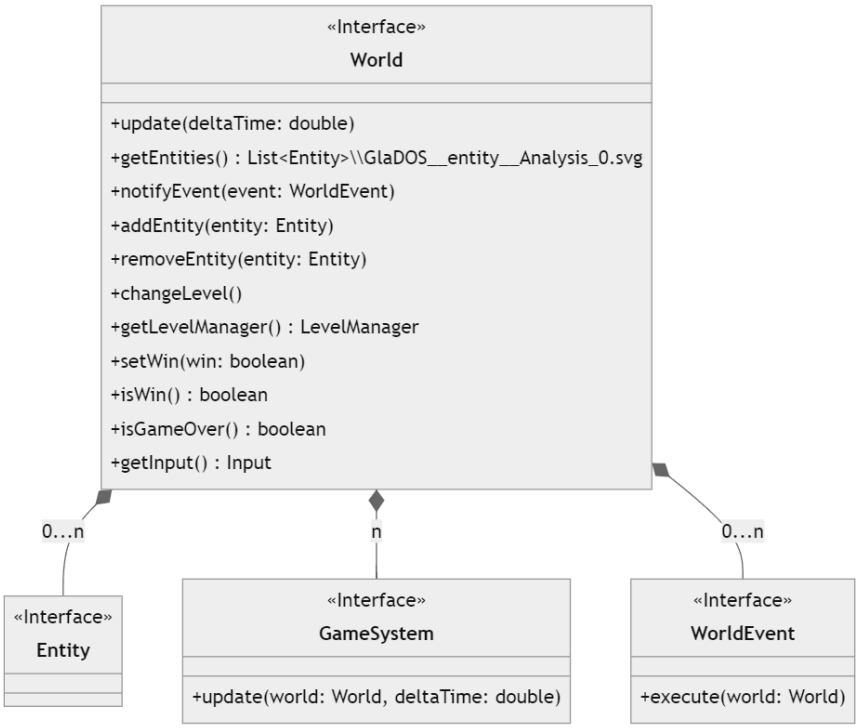

Figure 2.4: Diagramma UML del World

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, come ad esempio il metodo `update`, che viene chiamato ad ogni ciclo del *game loop* in esecuzione sull'`Engine`. 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 quest'ultimo motivo, 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

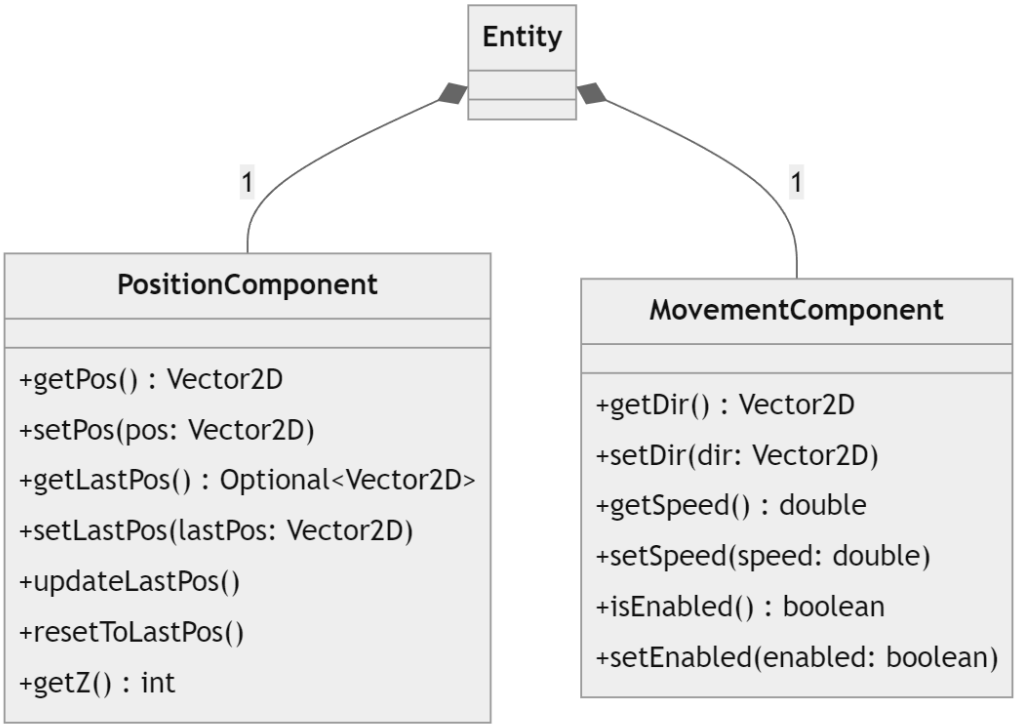

Figure 2.5: Diagramma UML dei componenti della posizone e del movimento

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

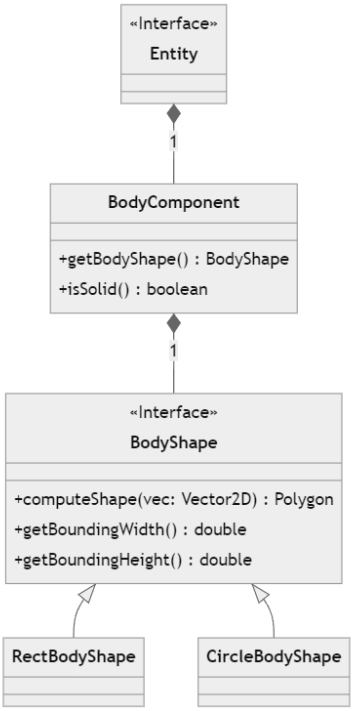

Figure 2.6: Diagramma UML del componente del corpo e delle shape

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

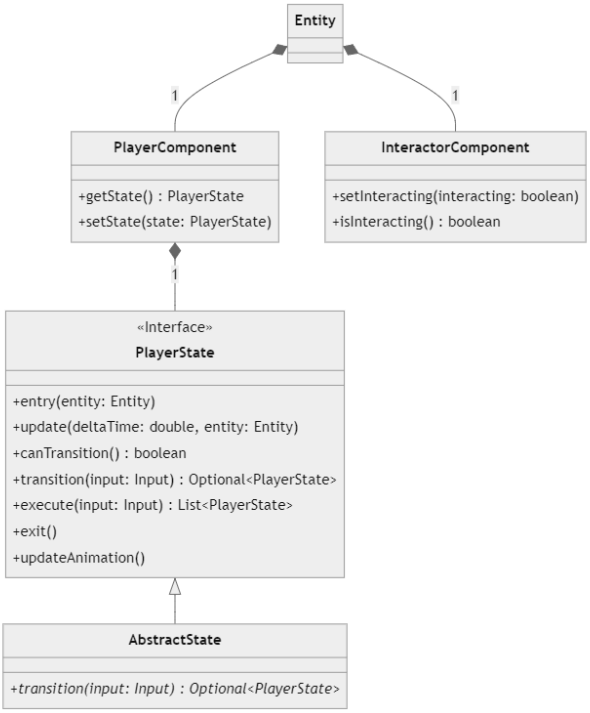

Figure 2.7: Diagramma UML dell'entita' che rappresenta il giocatore e alcuni dei suoi componenti

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-

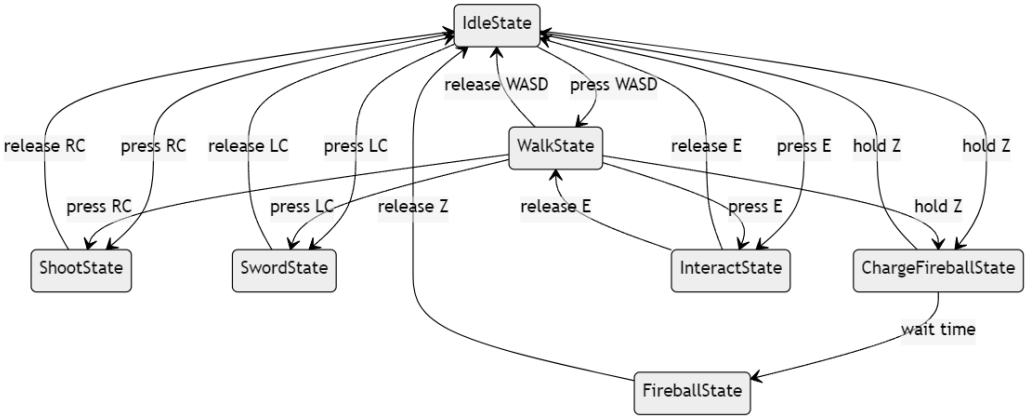

Figure 2.8: Diagramma degli stati del player. LC ed RC indicano rispettivamente il tasto sinistro e destro del mouse. Il player puo' trovarsi solo in uno di questi stati alla volta

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 di quello del mio collega Lorenzo Prati, cioè Top-Bottom)

AI

Problema Si vuole creare nemici con comportamenti differenti; il loro scopo principale deve comunque essere quello di eliminare il player.

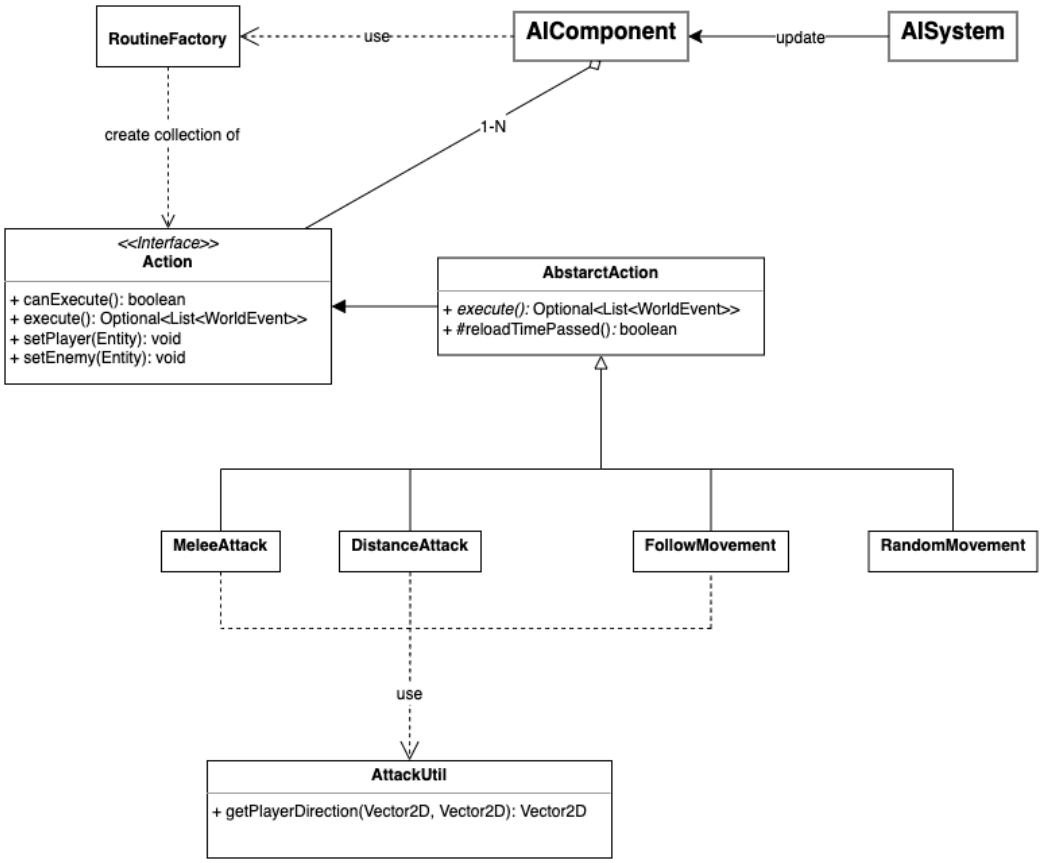

Figure 2.9: Diagramma UML delle Action.

Soluzione Ho ideato un `AISystem` che "estende l'abstract system" e si occupa di aggiornare l'`AIComponent` e notificare al `World` nuovi eventi che i nemici possono aver creato.

Ma come possono questi nemici creare eventi?
Ci riescono grazie alle `Action` che l'`AIComponent` contiene. Tendenzialmente un nemico avrà più Action quindi ho deciso di creare una `RoutineFactory` che mi resitituisse un *Comportamento* (inteso come: Insieme di azioni, di atteggiamenti con cui l'individuo esterna la propria personalità, rapportandosi agli altri e all'ambiente [Dizionario Italiano, Corriere della Sera]); ho

costruito questa classe usando il *Pattern Factory*.

Esempio: *createShooterRoutine()* restituisce una classica personalità zombie che segue il player se lo rileava 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.

Per la creazione delle `Action` mi sono ispirato allo Strategy Pattern. Nel mio caso però le Action oltre ad eseguire una certa azione valutano se è il caso di eseguirla. In questa prima versione del gioco il comportamento delle AI è determinato soltanto dalla distanza tra nemico e player (in altri termini: se il player entra nella aggro zone citata precedentemente) ed è comune a tutte le Action, per questo motivo ho deciso di implementare anche un `AbstractAction`. E' nota la limitazione che questo sistema provoca ma si è deciso che per un gameplay semplice come il nostro potesse comunque andare bene. In alcune Action viene fatta un ulteriore valutazione interna al esecuzione della stessa action, una valutazione di tipo temporale che rende i nemici soggetti al passare del tempo anche per la creazione degli eventi.

Per le Action che hanno bisogno di conoscere la posizione del player in termini di "direzione" più che "coordinate" ho pensato di creare una clase `AttackUtil` che espone un metodo utile ad ottenere, apputno, la direzione in cui is trova il player.

In conclusione ho ritenuto questo sistema abbastanza semplice per la creazione di nuove azioni. Un sistema che permette di creare nuove `Action` basandosi sullo spazio intorno alla AI e sullo scorrere del tempo. Considerando che il nostro dominio di gioco fa riferimento soltanto a nemici che hanno come unico scopo quello di eliminare il player, nonostante questo sistema sia limitante, è abbastanza completo; infatti senza considerare il dominio si può sfruttare questo sistema per la creazione di NPC.

2.2.3 Emanuele Dajko

2.2.4 Alessandra Versari

- Items: 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.

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.

Nella classe ItemFactory è stato utilizzato il Factory Method.

- InteractableObjects: 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. Con gli items, anche gli interactable hanno degli "effetti" sul player e sul gioco stesso. Il power-up vita infatti controlla che il player abbia monete sufficienti per eseguire l'acquisto, in caso affermativo viene modificato l'intero che indica la vita massima all'interno dell'HealthComponent. Il power-up velocità richiede anch'esso un controllo sulle monete raccolte dal giocatore e nel caso fossero sufficienti va ad aumentare il campo "speed" del MovementComponent. Il gate invece per poter essere attivato/utilizzato richiede che tutti i nemici siano stati eliminati e solo dopo aver fatto questo controllo lancerà l'evento ChangeRoomEvent() che permetterà di cambiare stanza. Una volta utilizzati, gli interactable objects vengono rimossi. Per implementare tutto ciò ho utilizzato il Factory Method (vedi InteractableObjectFactory) per creare i vari oggetti interactable e ho implementato gli "effetti" di tali oggetti sempre all'interno della classe Interactablefactory (per gli stessi motivi degli items visti precedentemente). Per la creazione degli effetti ho utilizzato anche in questo caso interfacce funzionali (Bifunction, BiPredicate, Predicate e BiConsumer). Oltre a necessitare l'interazione, questi oggetti si distinguono dagli items perché hanno la possibilità di lanciare eventi sul world all'interno del loro effetto.

  È stato utilizzato il Factory Method nella classe InteractableObjectFactory.

- Animazioni:

- Menù di gioco:

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.

3.2 Metodologia di lavoro

Lorenzo Prati

- Base del *pattern ecs*
  - 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`

-

3.3 Note di sviluppo

Lorenzo Prati

- uso della libreria jts (link alla pagina github) 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/587a03e4db001028dbcf5649ed6af5807a3ef15src/main/java/dimhol/logic/collision/RectBodyShape.java#L51-L62`

- reflection, stream e lambda nella classe `EntityImpl`: `https://github.com/LorenzoPrati/OOP22-dim-hol/blob/5d2b44fc5bf5fa50b6f0215695d2a5bac89cab8src/main/java/dimhol/entity/EntityImpl.java#L61-L89`

- stream e lambda utilizzati anche in `RemoveEntityEvent` e in `WorldImpl` e nel metodo template di `AbstractSystem`, e in minima parte 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

Chapter 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.2 Difficoltà incontrate e commenti per i docenti

Appendix A

Guida utente

Appendix B

Esercitazioni di laboratorio

Bibliography