ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

LOST IN DUNGEONS

Elaborato in
Paradigmi di Programmazione e Sviluppo

Presentato da
Luca Bazzocchi (0001002600 - luca.bazzocchi2@unibo.it)
Giacomo Casadei (0001005016 - giacomo.casadei12@unibo.it)
Fabio Muratori (0000984550 - fabio.muratori2@unibo.it)
Luca Bracchi (0000974076 - luca.bracchi3@unibo.it)

Anno Accademico 2020 – 2021

Introduzione

Tutti i membri del team condividono una grande passione per i videogiochi. Con questo progetto ci si è voluti addentrare, anche se solo parzialmente, in questo mondo per sperimentarne le problematiche e le soluzioni di realizzazione. Tutto questo ha portato alla realizzazione di **Lost in Dungeons** con l'obiettivo di sviluppare un gioco di genere Rougue-lite[1], caratterizzato da una mappa generata proceduralmente con il concetto di *"morte permanente"*, il reset completo dell'avventura in seguito al termine di una partita.

La maggiore fonte di ispirazione nella realizzazione di questo progetto, è il famoso *Dead Cells*[2], esso ha portato a farci scegliere un ambiente 2D invece che 3D, in stile pixel, in cui il player si trova all'interno di dungeon da esplorare pieni di mostri da affrontare. Il titolo inoltre vuole rimandare al fatto che il giocatore si troverà perso all'interno di dungeon, una sequenza di livelli sempre differenti e popolati da oggetti e nemici, senza una fine e con l'obiettivo di sopravvivere e di esplorarlo il più possibile.

[1] sottogenere di giochi di ruolo (RPG)

[2] `https://dead-cells.com/`

Indice

Elenco delle figure

Capitolo 1

Processo di sviluppo

Il progetto è stato realizzato adottando una metodologia di sviluppo *agile* che ha permesso una realizzazione per fasi, ognuna caratterizzata da obiettivi diversi e dalla consegna di un sottoprodotto sempre più vicino al prodotto finale in termini di feature e di qualità di quest'ultime.

Per aver modo di correggere i requisiti e di apportare modifiche in maniera rapida il prodotto viene mostrato al cliente in maniera frequente, raccogliendo e valutando i feedback di quest'ultimo. Per rendere più verosimile lo scenario, il cliente è stato impersonato dai membri del team stesso.

Si è scelto di usare *Scrum* come framework per la gestione del ciclo di sviluppo del software, adottando cicli settimanali, più comunemente chiamati *sprint*, e eleggendo tra i membri del team Luca Bazzocchi come Scrum Master e Fabio Muratori come Product Owner.

Il primo sprint è stato preceduto da una fase di raccolta dei requisiti, divisione dei compiti tra i membri del team, modellazione del sistema, prima progettazione di quest'ultimo attraverso diagrammi UML e creazione del **Product Backlog**, contenente la lista degli item con le relative priorità.

1.1 Divisione dei task

La divisione dei task ad alto livello è avvenuta nella fase iniziale del progetto: i requisiti sono stati suddivisi in macro-feature e ad ognuna di queste è stato assegnato un membro del team, cercando di bilanciare la suddivisione di massima dei lavori in base all'effort stimato per ogni blocco di feature e prevedendo un certo margine di flessibilità nel caso di eventuali imprevisti.

Gli item settimanali sono stati assegnati durante la fase di Sprint Planning, considerando un numero di ore lavorative standard per ogni membro del team e assegnando ad ognuno un carico di lavoro che non eccedesse il monte ore

della settimana. Per far ciò è stato necessario stimare la durata di ogni item in termini di numero di ore-lavoro.

1.2 Meeting

Sono stati adottati i principali eventi utilizzati in Scrum per ridurre al minimo la necessità di riunioni non definite dal framework stesso. I meeting hanno fatto uso di due piattaforme principali: Discord e WhatsApp.

All'inizio di ogni sprint è stata tenuta una riunione di **Sprint Planning**, dove venivano discussi gli obiettivi dello sprint entrante, selezionati e assegnati i task in base alla priorità e fatta una previsione sul numero di task completabili in quella settimana. Ogni Sprint Planning ha portato alla produzione di uno Sprint Backlog, prodotto della raffinazione della porzione di Product Backlog riguardante lo sprint attuale.

Al termine di ogni sprint è stata tenuta una riunione di Sprint Review, seguita da una di Sprint Retrospective. Negli **Sprint Retrospective** si è valutato l'andamento del progetto in termini di processo, definendo gli eventuali cambiamenti da apportare in quest'ultimo. Questi due eventi sono stati pianificati all'inizio della settimana, prima dello Sprint Planning relativo al nuovo sprint.

Al termine di ogni giornata lavorativa è stato effettuato un **Daily Scrum** su WhatsApp per aggiornare e coordinare i membri del team, soprattutto per quel che riguardava task bloccanti o che richiedessero la cooperazione di due o più membri per risolvere un singolo task.

1.3 Scelta degli strumenti

Sono stati adottati diversi strumenti per supportare le figure coinvolte nello sviluppo del progetto, con lo scopo di migliorare l'efficienza del processo automatizzandolo, così da permettere ai membri del gruppo di rimanere concentrati sugli sviluppi necessari per completare i task assegnati.

- **Git** come DVCS. Gli sviluppi sono stati effettuati su branch diversi, ognuno relativo a una macro-feature differente. Al termine di ogni sprint è stato eseguito un merge dei vari branch sul branch di sviluppo *develop*, seguito da una fase di testing del prodotto ottenuto e concludendo riportando tutte le modifiche sul branch *master*

- **GitHub** come servizio di hosting del codice sorgente

- **GitHub Action** come strumento di Continuous Integration per testare il progetto ospitato su GitHub
- **Google Drive** per la condivisione dei file usati dal team durante il processo di sviluppo, come i Product Backlog, gli Sprint Backlog, e i diagrammi UML
- **SBT** come strumento di build automation
- **ScalaTest** per quel che riguarda la scrittura e l'esecuzione dei test
- **Trello** come Spring Task Board

Capitolo 2

Requisiti

In questo capitolo verranno descritti i requisiti del sistema implementato. La scelta degli elementi nelle liste sottoelencate è stata orientata alla verificabilità dei singoli requisiti. I requisiti sono stati raccolti nel Product Backlog e sono rimasti generalmente invariati durante l'evoluzione del progetto, fatta eccezione per situazioni particolari che hanno richiesto la revisione degli item a minor priorità.

2.1 Requisiti di business

Gli obiettivi ad alto livello dell'applicazione **Lost in Dungeons** sono i seguenti:

- Il gioco è un RPG Rogue-lite dove l'eroe, impersonato dal giocatore, ha come obiettivo quello di sopravvivere all'interno di una sequenza di dungeon;
- La mappa di gioco e le entità al suo interno saranno generate in maniera pseudo-casuale;
- L'eroe che completa un dungeon ha la possibilità di passare al dungeon successivo.
- Eroe e nemici hanno statistiche che vengono modificate nel corso della partita:
  - L'eroe può raccogliere potenziamenti che migliorano le proprie statistiche
  - I nemici diventano più forti quando si passa da un dungeon a quello successivo

2.2 Requisiti utente

L'utente può:

- Iniziare una nuova partita
- Muoversi all'interno del dungeon:
  - La visuale di gioco deve seguire i movimenti dell'eroe
- Eseguire mosse speciali:
  - Saltare
  - Eseguire un doppio salto
  - Accovacciarsi
  - Scivolare sul pavimento
  - Attaccare da fermo
  - Attaccare in aria
  - Attaccare a distanza
- Interagire con le entità sparse nel dungeon:
  - Colpire i nemici
  - Aprire i forzieri
  - Raccogliere gli oggetti
  - Arrampicarsi su scale e liane
  - Scendere dalle piattaforme
  - Aprire le porte
  - Entrare nei portali
  - Venire rallentato dall'acqua
  - Subire danni dalla lava
- Ottenere oggetti:
  - Sconfiggendo i nemici
  - Aprendo i forzieri
  - Accedendo a stanze speciali situate nel dungeon
- Potenziare l'eroe raccogliendo oggetti che conferiscono bonus

- Attaccare con la spada o con l'arco (una volta ottenuto)
- Raccogliere una chiave che gli permetta di accedere alla stanza del boss
- Completare il dungeon attuale sconfiggendo un boss
- Accedere al livello successivo tramite un portale

2.3 Requisiti funzionali

Il sistema prodotto deve:

- Permettere al giocatore di iniziare una nuova partita o di uscire
- Creare il dungeon come composizione di stanze, alcune fisse e altre che variano da dungeon a dungeon
- Posizionare l'eroe nell'apposita stanza di spawn
- Supportare diverse tipologie di nemici, caratterizzati da differenti strategie di attacco e movimento In particolare sarà possibile incontrare:
  - *Skeleton*: un nemico discretamente proficiente nei combattimenti corpo a corpo e dotato di attacchi ampi e difficili da schivare;
  - *Slime*: un nemico molto lento nel movimento ed attacco ma particolarmente robusto. Saranno necessari molteplici attacchi per poterlo sconfiggere;
  - *Worm*: un nemico non molto resiliente ma dotato di attacchi a distanza in grado di colpire il giocatore da molto lontano;
  - *Bat*: un nemico in grado di raggiungere il giocatore ignorando la conformazione della mappa;
  - *Wizard*: un nemico decisamente più forte e resistente di altri, abile sia nei combattimenti ravvicinati ma anche in battaglie a distanza e statistiche che lo rendono difficile da sconfiggere.
- Creare il boss nell'apposita stanza
  - Il boss ha un comportamento più complesso rispetto a quello dei nemici comuni
- Creare le entità nelle varie stanze del dungeon:
  - *Suolo*: blocchi statici attraverso i quali le entità non possono passare

  - *Piattaforme*: l'eroe può camminarci sopra o interagirci per scendere di sotto
  - *Scale e liane*: interagendovi l'eroe può salire o scendere
  - *Bauli*: possono essere aperti interagendovi per ottenere l'oggetto che contengono
  - *Porte*: possono essere aperte interagendovi, tranne la porta della sala del boss che richiede una chiave speciale
  - *Nemici*: gli attacchi dei nemici danneggiano l'eroe e questo può sconfiggere i nemici usando la spada o l'arco
  - *Pozze e cascate di acqua*: l'eroe vede la sua velocità decrementata quando è a contatto con l'acqua
  - *Pozze di lava*: l'eroe viene danneggiato dal contatto con la lava
  - *Portale*: attivabile sconfiggendo il boss, permette di accedere al dungeon successivo
- Permettere la crezione di mappe, nemici, oggetti generati pseudocasualmente

2.4 Requisiti non funzionali

Il sistema prodotto deve:

- Presentare un grado di fluidità di gioco accettabile
- Comprendere animazioni particolari
- Avere un interfaccia semplice e intuitiva

2.5 Requisiti di implementazione

Per quel che riguarda i requisiti di implementazione, l'applicazione:

- Sarà sviluppata in Scala, usando ScalaTest per il testing
- Farà uso della libreria LibGDX per gestire le interazioni audio/grafiche e le proprietà fisiche del mondo di gioco
- Userà mappe disegnate con lo strumento LDTK e popolate con le entità necessarie con Tiled

Capitolo 3

Analisi del dominio

Lost in Dungeons vuole immergere il giocatore nell'ambiente di gioco, facendolo perdere all'interno del suo vasto ed impredicibile mondo.

All'inizio di ogni partita, il player sotto le vesti dell'hero, cercherà di addentrarsi all'interno di dungeon per proseguire sempre di più la sua avventura. Lo scopo del gioco è cercare di resistere il più possibile nei livelli senza morire mentre si sconfiggono nemici e raccolgono strumenti, aumentando così lo score della partita; ma non esiste una via d'uscita, il giocatore rimarrà per sempre perso all'interno di dungeon.

3.1 Mondo di gioco

In ogni livello l'eroe verrà posizionato all'interno di un dungeon bidimensionale, in un punto specifico comune a tutti i dungeon.

I livelli sono costruiti come composizione di stanze fisse e variabili: i bordi esterni e le stanze speciali sono fisse, mentre le stanze interne e i bordi tra queste variano da dungeon a dungeon.In particolare, ogni dungeon prevede:

- Stanze fisse:
  - quattro bordi, uno per ogni punto cardinale;
  - una stanza dove posizionare l'eroe (hero-room);
  - una stanza dove posizionare il boss (boss-room);
  - due stanze speciali dove posizionare gli oggetti speciali, uno dei quali è la chiave per accedere alla stanza del boss;
- Stanze variabili
  - sei stanze interne scelte e ordinate in maniera casuale da un insieme di possibili mappe;

– un bordo interno, che separa le stanze interne, scelto in maniera casuale da un set di possibili mappe e che definisce quali sono i passaggi tra le stanze permessi e quali no.

3.2 Player

Il giocatore potrà far eseguire azioni all'hero attraverso la premuta di pulsanti sulla tastiera e click sul mouse, in particolare, l'hero avrà, per quanto riguarda i movimenti:

- una scivolata che gli permetta di passare in spazi più stretti del normale e di schivare gli attacchi diretti verso di lui;
- la possibilità di correre se non è in aria;
- due salti, uno consecutivo all'altro per raggiungere le piattaforme più elevate.

L'Hero

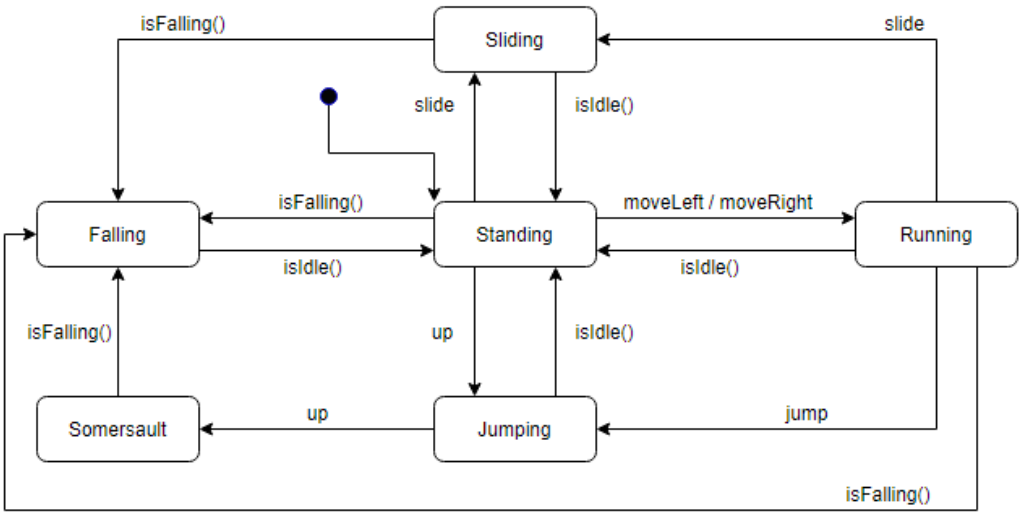

Figura 3.1: Diagramma degli stati rappresentante i movimenti del player.

- tre attacchi con la spada da fermo, essi sono consecutivi, avvengono uno dopo l'altro ed interropono eventuali azioni che stava eseguendo l'hero;
- un attacco mentre si è in aria; esso porta ad eseguire uno schianto a terra con la spada;

- un attacco a distanza, eseguibile una volta ottenuto l'arco; viene sparata una freccia verso la direzione in cui sta guardando il giocatore, essa colpirà l'entità a cui va incontro e, se è un nemico, gli causerà del danno;
- una scivolata che gli permetta di passare in spazi più stretti del normale e di schivare gli attacchi diretti verso di lui;
- due salti, uno consecutivo all'altro per raggiungere le piattaforme più elevate.

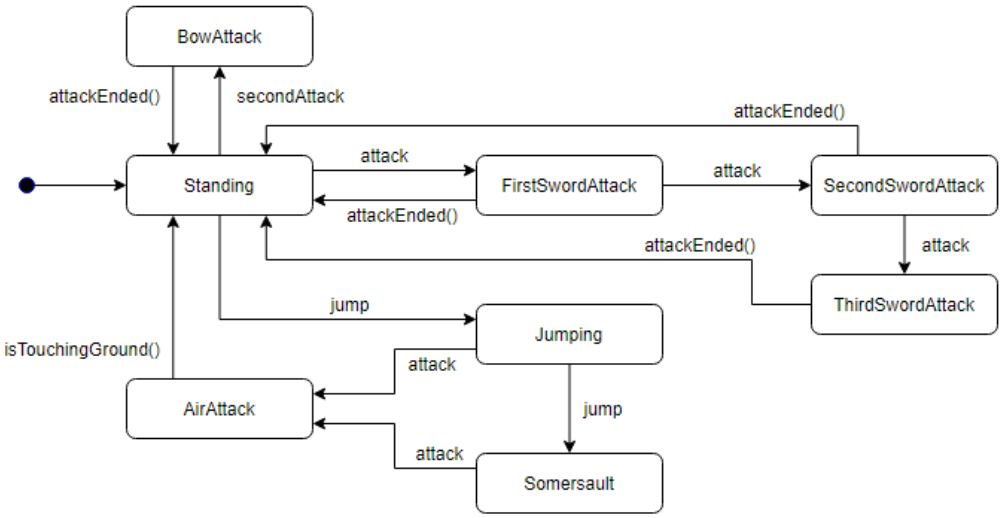

Figura 3.2: Diagramma degli stati rappresentante gli attacchi del player.

L'hero, come ogni entità mobile, presenterà delle statistiche di default, quali vita, forza, velocità, difesa; che potranno aumentare o diminuire raccogliendo oggetti, subendo danno e superando i livelli. Attraverso tali statistiche la difficoltà e velocità di avanzamento dei livelli sarà variabile.

3.3 Nemici

La mappa di gioco sarà popolata da diverse tipologie di nemici caratterizzati da statistiche, attacchi e metodi di movimento personalizzati. Generalmente, i nemici presentano sia statistiche personalizzate che comportamenti (movimenti e metodi di attacco) altamente variabili e variegati.

Si identificano 3 macro-categorie di metodi di movimento:

- per nemici a terra - un movimento caratterizzato da diversi stati quali "sorvegliare" una piattaforma orizzontale, "affrontare" il giocatore non appena si trova nelle prossimità del nemico, "inseguire" il giocatore che si allontana. Questo tipo di movimento è tipico dei nemici *Skeleton*, *Slime* e *Worm*

- per nemici volanti - un movimento adottabile da entità nemiche volanti (*Bat*). Questo metodo ignora la conformazione della mappa e permette al nemico di attraversare ostacoli e moversi verso il giocatore ma solamente se sufficientemente vicino.

- per il boss di fine livello - l'entità nemica incontrata alla fine del livello deve essere sempre in grado di percepire il giocatore, avvicinarsi quando possibile e affrontarlo senza mai dargli le spalle.

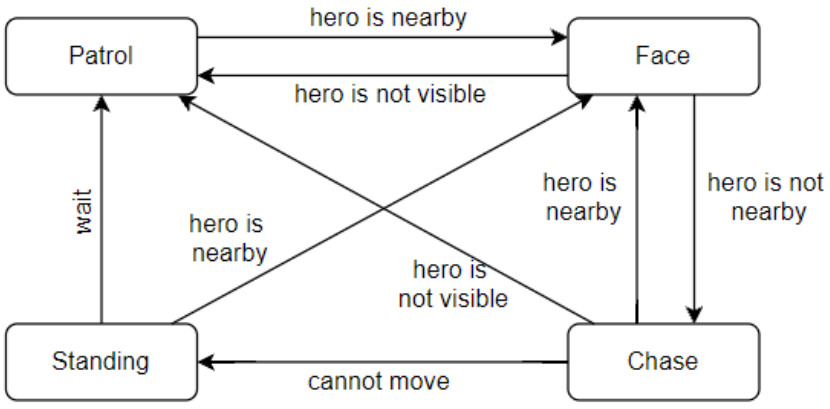

Figura 3.3: Macchina a stati della tipologia di movimento *a terra* utilizzato da molteplici entità nemiche.

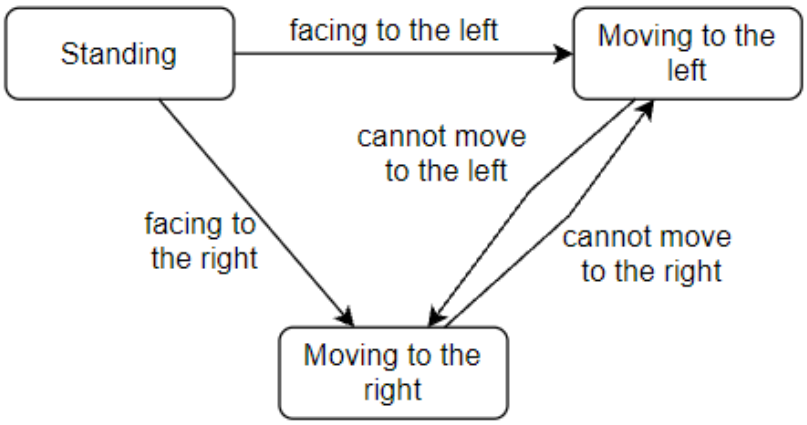

Figura 3.4: Macchina a stati della tipologia di movimento Patrol (pattuglia).

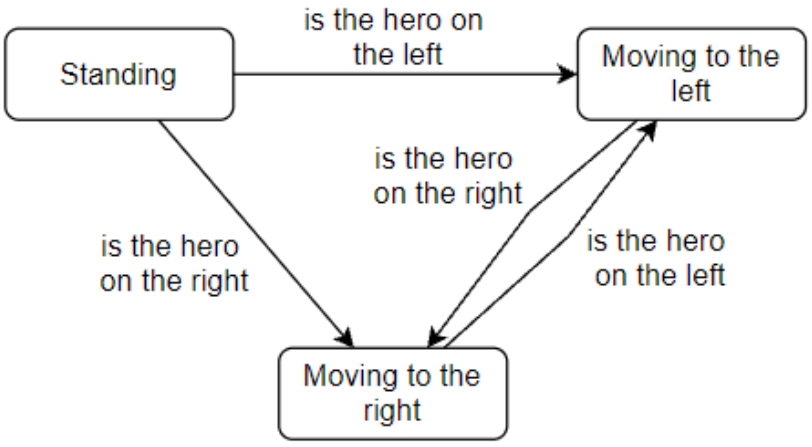

Figura 3.5: Macchina a stati della tipologia di movimento Chase (insegui).

Nell'immagine 3.3 è esplorato il movimento di un nemico *Skeleton*. Attraverso una macchina a stati è possibile definire chiaramente come un entità si comporta ed anche secondo quali condizioni è possibile modificare comportamento. Si nota come determinate porzioni della macchina siano esse stesse dei potenziali metodi di movimento più rudimentali e riusabili da altre politiche (ad esempio la porzione *Facing* è comune a tutte le entità nemiche che devono sempre rivolte verso il giocatore).

Il comportamento adottato in 3.4 ed in 3.5 mostrano un insieme di stati identico con l'unica differenza identificabile nelle transizioni che comportano una modifica del comportamento.

Inoltre, ogni nemico può essere dotato di uno o più metodi di attacco:

- attacco corpo a corpo - eseguito dai nemici *Skeleton*, *Bat*, *Slime* e *Wizard* quando il bersaglio si trova a breve distanza;

- attacco a distanza - eseguito dai nemici *Worm* e *Wizard* quando il bersaglio è visibile o generalmente entro una distanza dal nemico più ampia rispetto ad attacchi corpo a corpo.

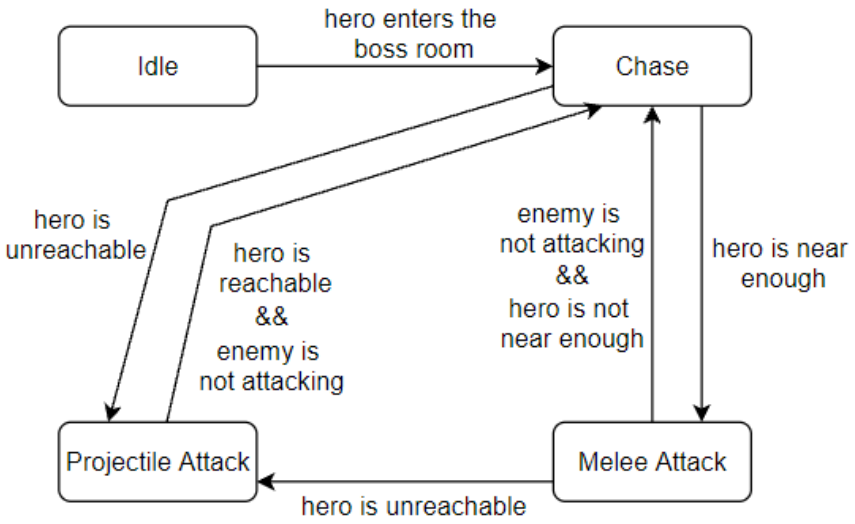

Figura 3.6: Comportamento adottato dal nemico *Wizard*, il boss incontrato alla fine di un livello del dungeon.

Nell'immagine 3.6 invece viene ancora una volta utilizzata una macchina a stati per rappresentare il comportamento generale dell'entità *Wizard*, il boss di fine livello. In questo caso si nota come lo stato non governi più solamente il movimento dell'entità ma anche la politica di attacco che egli adotta.

3.4 Interazioni ambientali

Nell'esplorazione della mappa di gioco, il giocatore potrà interagire con l'ambiente, anche per raggiungere posti altrimenti irraggiungibili. Ad esempio saranno presenti:

- *scale e liane* per potersi muovere verticalmente, il cui percorrimento comporterà una modifica del comportamento del giocatore il quale sarà solo in grado di muoversi verso l'alto o verso il basso;
- *porte* sparse per la mappa che impediscono il normale attraversamento delle entità mobili (proiettili, nemici e giocatore) ma apribili dal giocatore attraverso un apposito comando o attacco con la spada
- *porta* che impedisce l'accesso all'arena in cui dovrà essere affrontato il boss ed apribile solamente dopo aver ottenuto la chiave trovabile all'interno della mappa
- *forzieri* posizionati in aree di gioco difficilmente raggiungibili e contenenti pozioni in grado di curare il giocatore
- *piattaforme* attraverso le quali il giocatore può salire o scendere
- *stagni d'acqua* che rallentano il giocatore
- *pozze di lava* in grado di danneggiare il giocatore, oltre che rallentarlo
- *portale* che trasporta il giocatore in un nuovo dungeon ed attivabile alla morte del boss

3.5 Oggetti

Il giocatore sarà in grado di raccogliere oggetti sparsi nella mappa:

- *pozioni* raccoglibili da forzieri o alla morte dei nemici incontrati nell'avventura ed in grado di curare le ferite dell'hero
- *chiave* ottenuta raggiungendo specifiche stanze all'interno della mappa di gioco che permette l'accesso all'arena dove sarà possibile affrontare il boss
- un *arco* ottenuto in seguito alla morte del boss di un livello che permette al giocatore di attaccare a distanza
- *equipaggiamenti* in grado di potenziare il giocatore

Capitolo 4

Design

In questo capitolo viene descritto in un primo momento il design di massima prodotto dall'analisi dei dominio applicativo in tutte le sue macro-parti ed in seguito viene nel dettaglio esplorato ciascuno componente degno di nota, analizzandone scelte architetturali.

4.1 Design architetturale

Nell'analizzare i requisiti ed il domioni applicativo vengono definiti i macro-componenti del sistema, i compiti che essi avranno in un ottica il più possibile modulare e generalizzata. La definizione della architettura di massima del sistema è suddivisa principalmente in 2 aspetti: modello MVC e studio delle entità.

4.1.1 Struttura MVC

Data la natura del progetto si è scelto di utilizzare il pattern strutturale Model-View-Controller nell'identificazione dei principali componenti e delle relazioni che tra di essi intercorrono. In particolare, dovendo permettere la visualizzazione a schermo della sessione di gioco e al contempo permettere l'esecuzione di task relativi al livello, si è deciso di impiegare tale pattern per mantenere l'architettura il più possibile modulare e flessibile.

VIEW La view è in grado di rappresentare il mondo di gioco in maniera grafica all'utente. I compiti individuabili si possono riassumere in: gestire opportunamente le varie schermate (menu iniziale, schermata di fine partita e di gioco), catturare gli input del giocatore e propagarli al Controller, visualizzare

lo stato di avanzamento del gioco con un elevata frequenza di aggiornamento, gestire vari *spritesheets*[1] e definire animazioni associate alle varie entità.

CONTROLLER Il Controller ha il compito di gestire l'esecuzione del Model, ricevere e processare eventi generati dalla View ed anche propagare lo stato del Model stesso alla View. Sia la View che il Controller hanno un proprio flusso di controllo indipendente ed è qui che aspetti di esecuzione asincrona sono gestiti.

MODEL Il Model racchiude il mondo di gioco nella sua interezza, comprendedo sia la fisica che aspetti prettamente ludici. I compiti principali del Model sono: creare e mantenere lo stato di gioco ed in particolare il livello currente, mantenere e aggiornare le varie entità presenti all'interno del livello, simulare la fisica ed identificare collisioni.

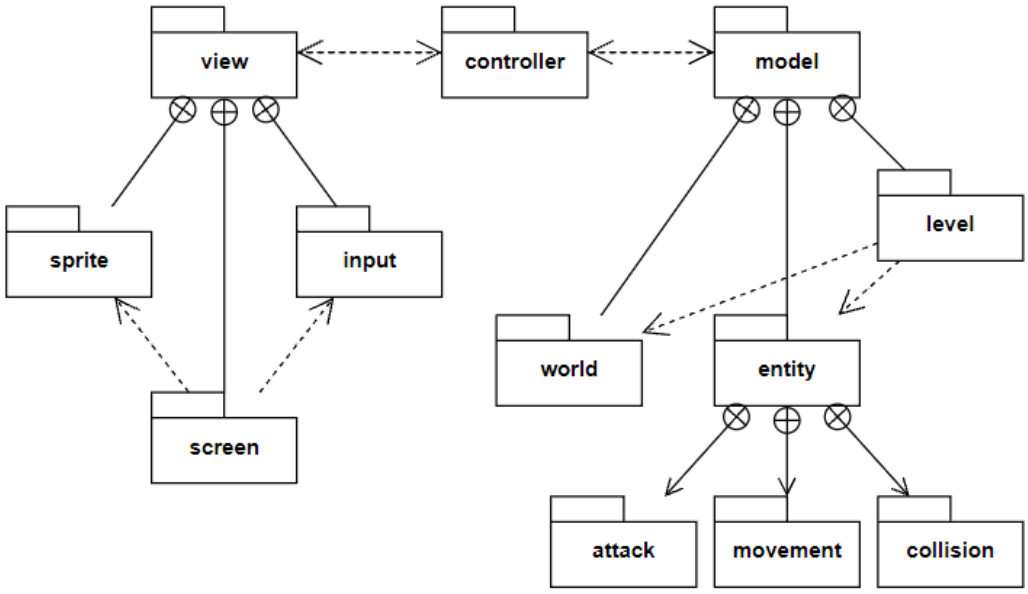

Figura 4.1: Diagramma di massima dell'architettura del progetto.

4.1.2 Entità

Dall'analisi del dominio applicativo si evidenzia una particolare importanza nel concetto di **entità**: essa può essere sia una piattaforma sopra la quale il giocatore può muoversi, un muro invalicabile, il giocatore stesso, un nemico, una pozza di lava, una scala, un proiettile od un attacco creato da giocatore e nemici. Ciascuna di esse è caratterizzata da una forma ed una posizione nel

[1] un'immagine composta a sua volta da immagini più piccole dette *sprite* che vanno a comporre un'animazione

mondo. Tutte le possibili interazioni che possono portare ad una modifica dello stato del gioco sono derivate da collisioni (o sovrapposizioni, intersezioni) tra entità:

- il giocatore subisce danno se tocca la lava;
- il giocatore è rallentato se entra all'interno di una pozza di acqua;
- il giocatore può interagire con oggetti, porte, portali, scale, etc. nel momento in cui viene a contatto con le relative entità;
- il giocatore, così come alcuni tipi di nemici, non può attraversare muri o piattaforme.

Un ulteriore caratteristica identificabile è la distinzione tra entità **mobili**, come giocatore, nemici, attacchi, ed entità **immobili** come muri, porte, piattaforme, portali, distinzione definita quindi sulla capacità dell'entità di muoversi e cambiare posizione. Di seguito sono forniti alcuni esempi:

- il giocatore può muoversi a destra, sinistra, saltare, salize e scendere le scale a seconda dei comandi inviati ed anche considerando lo stato dell'entità;
- ogni tipologia di nemico individua differenti comportamenti di movimento ed anche la possibilità di variare nel comportamento adottato in seguito a interazioni con altre entità (si prenda in considerazione un nemico che cambia direzione se collide con un muro) o condizioni (il giocatore è sufficientemente vicino, la vita del nemico è inferiore alla metà, etc.);
- ogni attacco generato da giocatore o nemici può a sua volta avere un movimento particolare (in proiettile che "insegue" il giocatore, un attacco con la spada).

Un'ultima distinzione degna di nota è la definizione di entità **viventi** ed entità **non viventi**, ossia caratterizzate o meno da una *vita* e che possono morire; si identificano in questo caso entità vivienti quali giocatore e nemici mentre tutte le altre entità citate sono non vivienti. Una caratteristica aggiuntiva in questo dominio è la capacità di attaccare e provocare danno che comporta la diminuzione della vita in altre entità viventi.

4.2 Design di dettaglio

In questa sezione vengono discusse le scelte principali che hanno guidato lo sviluppo dei componenti e l'organizzazione del software. Si porrà particolare attenzione alle interazioni con la librerira LibGDX utilizzata in vari punti del sistema.

4.2.1 Model View Controller

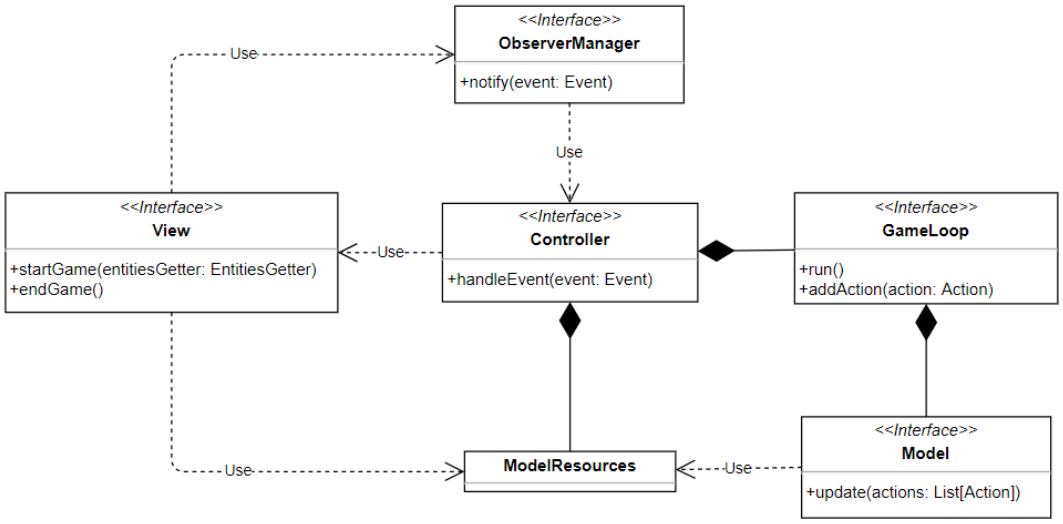

Figura 4.2: Diagramma delle interazioni tra Model, View e Controller.

In Figura 4.2 è rappresentata una versione esemplificata della struttura MVC e le interazioni tra Model, View e Controller. Data la natura del sistema di gioco, è stato necessario separare l'esecuzione del processo in due flussi di controllo distinti, uno per la visualizzazione della sessione di gioco ed uno per l'aggiornamento e la simulazione del mondo stesso. Questo ha portato alla realizzazione di un GameLoop il cui compito è permettere l'aggiornamento del Model ed un Monitor per l'accesso ad informazioni di gioco da visualizzare nella View.

Ogni operazione richiesta dal giocatore è propagata al GameLoop che racchiude e gestisce il flusso di esecuzione del Model: in questo modo si separa la richiesta dell'utente dal momento in cui essa verrà effettivamente eseguita, provocando una modifica del mondo di gioco.

Per permettere la visualizzazione del mondo di gioco si è fatto uso di un *monitor* come contenitore di risorse del livello accedute in maniera concorrente dai 2 flussi di controllo: il Model popola questo contenitore con le informazioni del livello come ad esempio le entità di gioco, mentre la View legge esclusivamente i valori per poi poterli visualizzare a schermo. Questo monitor è inizializzato dal Controller ed identifica un canale preferenziale tra View e Model, che non richiede l'intervento dello stesso. Tutte le tematiche di gestione della concorrenza e dei 2 flussi principali sono state gestite all'interno del Controller, rendendo quindi più agevole la realizzazione del sistema e la formalizzazione delle interazioni tra View e Model.

LibGDX

LibGDX è un framework di sviluppo di giochi sulla JVM multipiattaforma basato su OpenGL. Questa libreria fornisce una vasta gamma di funzionalità ad elevate performance. Durante la progettazione è stato necessario identificare i soli moduli di interesse allo sviluppo ed integrarli all'interno della struttura MVC del progetto.

È da precisare il fatto che pur essendo stato usato LibGDX trasversalmente all'interno del progetto, è stato possibile separare nettamente le tematiche inerenti alla visualizzazione delle entità nella View ed alla simulazione delle stesse nel Model; in questo modo una eventuale modifica del framework di visualizzazione del gioco non pregiudica l'utilizzo della libreria all'interno del Model, garantendo l'indipendeza tra i due ambiti.

In generale sono identificati di seguito le varie funzionalità demandate al framework:

- **VIEW** L'utilizzo del framework di sviluppo LibGDX ha permesso di agevolare lo sviluppo e la strutturazione della View; infatti esso definisce la struttura e le modalità di inizializzazione e gestione delle varie schermate, la gestione degli input da tastiera e mouse ed anche le metodologie per l'organizzazione di animazioni e suoni.

- **MODEL** All'interno del Model, la libreria ha trovato utilizzo nella simulazione della fisica del gioco, l'identificazione di collisioni tra entità e tra oggetti denominati *sensori* i quali non hanno rappresentazione visiva ma permettono ad entità autonome di avere consapevolezza di ciò che le circonda e potersi comportare di conseguenza.

LDtk e Tiled

Per disegnare le mappe è stato usato il software open-source *LDtk*[2] (Level Designer toolkit), prodotto dal creatore di Dead Cells con l'obiettivo di offrire un sistema user-friendly per il design di mappe 2D.

Una volta definita la dimensione della mappa e il tileset da utilizzare, LDtk permette di definire una lista di livelli e, per ognuno di questi, un insieme di regole di visualizzazione che definiscono come vengono auto-renderizzati i tile disegnati. In questo modo si automatizza un processo solitamente noioso e ripetitivo: ad esempio è stato possibile indicare i blocchi di vegetazione da posizionare sopra il suolo e la percentuale di occorrenza, così il toolkit li ha aggiunti in automatico, mentre il progettista ha dovuto solo tracciare i blocchi di base che componevano il suolo.

[2] `https://www.ldtk.io/`

Una volta terminata la mappa è stato possibile esportarla in un formato leggibile da *Tiled*[3], level editor 2D che è stato usato per inserire nella mappa le entità che la popolavano, quindi i blocchi che compongono il suolo, i nemici, acqua, lava e le altre entità già citate in precedenza. Per aggiungere le entità è sufficente aggiungere un object-layer sopra la mappa, definirlo con un nome fissato in precedenza (ad esempio "ground" per il suolo) e disegnare i rettangoli che lo compongono.

Tiled offre la possibilità di definire i bordi delle entità a mano libera, ma visto che questo avrebbe complicato la renderizzazione con LibGDX, si è optato per utilizzare unicamente entità rettangolari, lineari e puntiformi.

Il software Tiled è stato usato anche per aggiungere le animazioni nelle varie mappe (acqua, lava e torce): per fare ciò è bastato selezionare il tile nel tileset e definire quali altri tile compongono l'animazione e quanti millisecondi deve durare ogni tile presente nell'animazione.

Questi due strumenti hanno permesso di realizzaare mappe dall'aspetto soddisfacente in una quantità di tempo decisamente limitata.

4.2.2 Entity

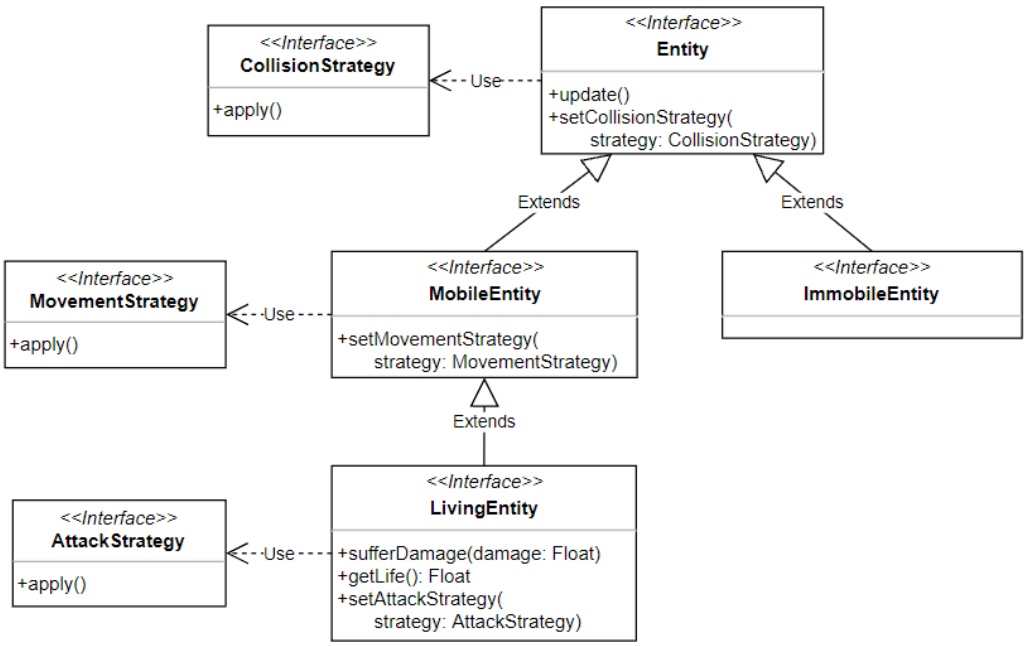

Figura 4.3: Diagramma della strutturazione di entità.

[3] `https://www.mapeditor.org/`

Per permette una corretta rappresentazione delle entità è stato necessario formalizzarne la tassonomia come segue:

- **Entity** ogni elemento che possiede una posizione nel mondo di gioco, un corpo e potenzialmente un comportamento derivato dalla collisione con altre entità
- **MobileEntity** un sottoinsieme di Entity caratteruzzati dalla capacità di muoversi o generalmente cambiare posizione secondo uno specifico metodo di movimento
- **LivingEntity** una specializzazione di MobileEntity in corrispondenza di entità capaci di attaccare e subire danno

Inoltre, in corrispondenza di ogni classificazione è possibile indivudiare una collezione di possibili comportamenti, sintetizzati come segue:

- **CollisionStrategy** individua il comportamento da adottare durante il *ciclo di vita* (inizio e fine del contatto) di una collisione individuata tra 2 entità
- **MovementStrategy** incapsula la strategia di movimento associata ad un'entità mobile
- **AttackStrategy** incapsula le condizioni necessarie affinchè un'entità vivente possa attaccare ed anche cosa comporta la creazione di un entità di attacco (definito come estensione di un entità mobile)

Dall'analisi dei requisiti si nota come tutte le entità viventi siano in realtà un sottoinsieme delle entità mobili così come queste ultime siano una specializzazione di entità generiche. Alternativamente, si sarebbe potuto optare per una specializzazione della classe Entity in MobileEntity e LivingEntity, aspetto che però non avrebbe reso evidente la relazione tra queste ultime. Infatti, dall'analisi dei requisiti non è emerso il caso di entità viventi (quindi dotate di statistiche relative) ma non in grado di muoversi, così come la presenza di entità capaci di attaccare escluse quelle viventi.

Un'altra osservazione degna di nota è la capacità di poter cambiare dinamicamente il comportamento adottato da un entità. Tramite gli strategy quindi è da una parte facile implementare comportamenti variegati e riusabili e dall'altra diventa semplice poter modificare un comportamento dinamicamente all'occorrenza. All'interno delle classi Entity e delle sue specializzazioni è presente solamente il comportamento basilare che gestisce quindi le interazioni con le varie strategie.

CollisionStrategy

Si vuole di seguito esplorare le scelte archietturali che hanno definito la struttura implementativa dei metodi di gestione delle collisioni. Come accennato in precedenza, in questa struttura trova spazio una funzionalità fornita dal framework LibGDX per l'identificazione di collisioni tra i corpi delle entità.

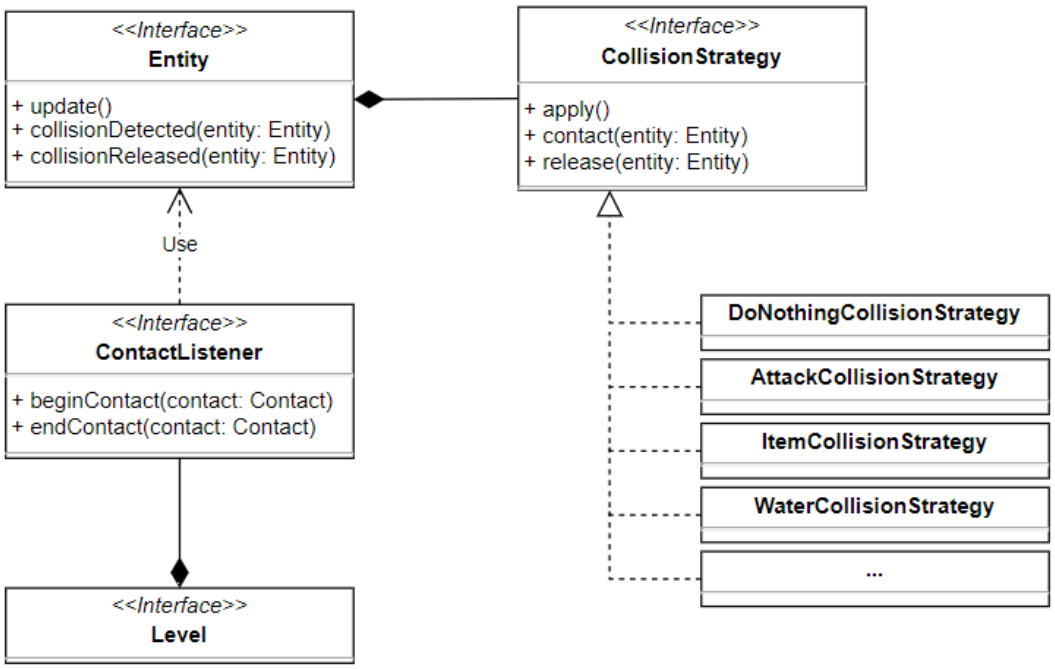

Figura 4.4: Diagramma della gestione di collisioni.

In Figura 4.4 si identifica la presenza del pattern di progettazione Strategy tra la classe Entity e CollisionStrategy. La ricezione di eventi relativi al *life cycle* di una collisione sono recepiti dall'interfaccia **ContactListener** fornita da LibGDX, da cui è possibile risalire alle entità coinvolte sia nella fase di *primo contatto* che di *rilascio della collisione*. Ogni entità rappresentata dalla classe Entity fornisce delle funzionalità per la ricezione di tali eventi. La gestione reale delle collisioni è in seguito demandata ad un particolare CollisionStrategy assegnato durante la creazione dell'entità.

MovementStrategy

Data la necessità di mantenere e gestire un insieme potenzialmente grande di metodi di movimento, anche in questo caso, è stato scelto di usare il pattern di programmazione Strategy per separare le politiche di movimento esternamente alla entità mobili. Inoltre, considerando la potenziale complessità elevata nel

gestire ed implementare strategie di movimento e dall'altra la necessità di permettere una definizioe dinamica delle strategie utilizzate, è stato necessario adottare un pattern di programmazione State.

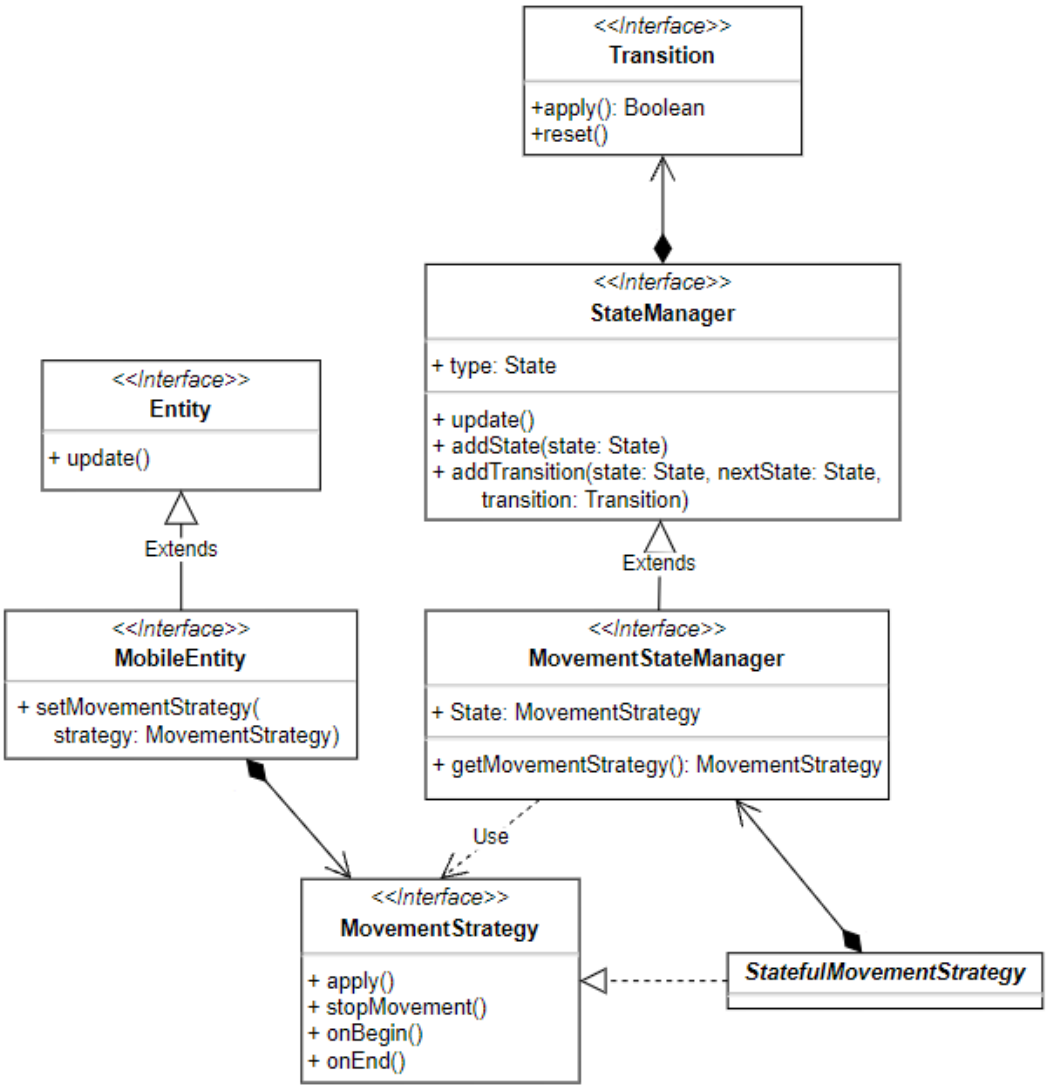

Figura 4.5: Diagramma della gestione di strategie di movimento.

In Figura 4.5 è possibile osservare la struttura adottata: una MobileEntity si compone di un MovementStrategy; una delle possibili istanziazioni dell'interfaccia è rappresentata dalla classe astratta StatefulMovementStrategy che racchiude al suo interno uno StateManager, una struttura utilizzata per consentire la modifica del metodo di movimento secondo delle specifiche tran-

sitioni, ossia condizioni verificabili durante l'esecuzione del gioco che devono comportare una mutazione nello stato corrente.

In questo modo è possibile fornire ad una MobileEntity dei MovementStrategy più o meno complessi in maniera trasparente ma anche realizzare una strategia di movimento come composizione di altre strategie in maniera agevole e facilmente mantenibile e testabile.

AttackStrategy

Il meccanismo utilizzato per la specificazione di attacchi segue, similmente alla gestione delle collisioni, il pattern Strategy. Ogni strategia definisce quando è possibile effettuare un attacco ed anche cosa comporta la creazione dell'attacco stesso. Ogni attacco è una particolare istanza di MobileEntity, caratterizzata da un CollisionStrategy ben definito (visibile in Figura 4.4, Classe AttackCollisionStrategy) e da un particolare metodo di movimento necessario principalmente per attacchi di tipo proiettile.

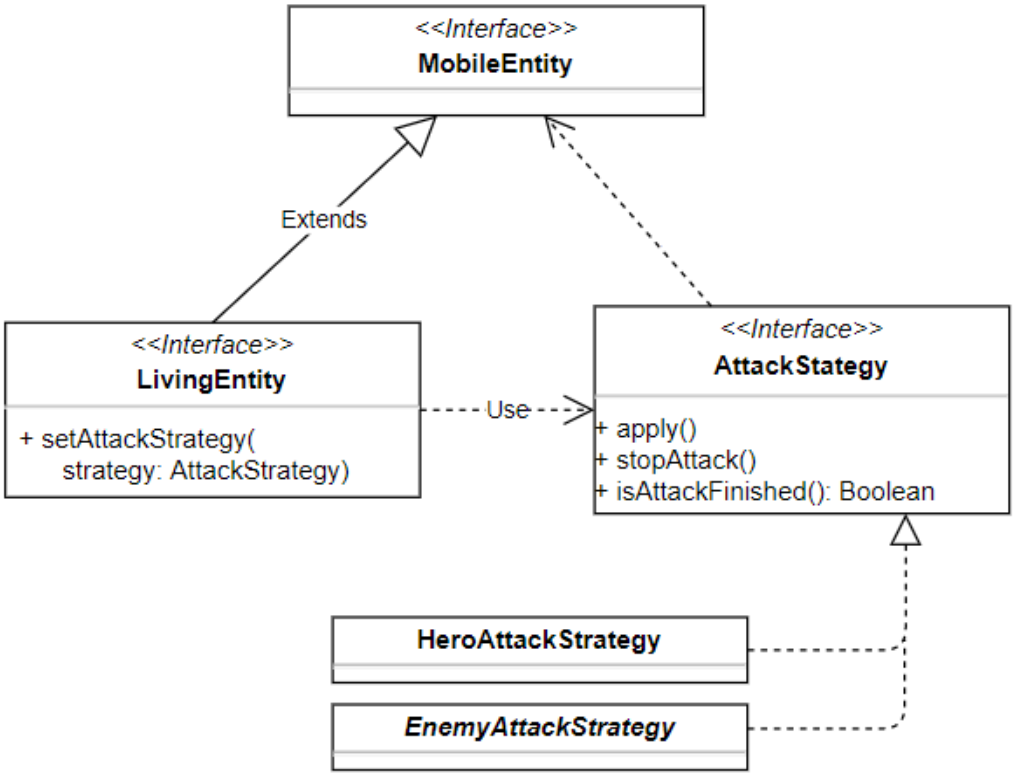

Figura 4.6: Diagramma della gestione di strategie di attacco.

4.2.3 Map

Ogni mondo è strutturato come una griglia 2x3 dove ogni cella corrisponde a una stanza 40x40[4]. Le stanze interne (rispetto all'asse orizzontale) consentono di accedere alle sale speciali, dove è possibile trovaere la chiave per la stanza del boss. La stanza dell'eroe e quella del boss sono agli antipodi, la prima a ovest e la seconda a est, ognuna centrata rispetto all'asse verticale e confinante con entrambe le stanze laterali della griglia 3x2.

La figura 4.7 mostra la disposizione delle stanze che compongono il dungeon.

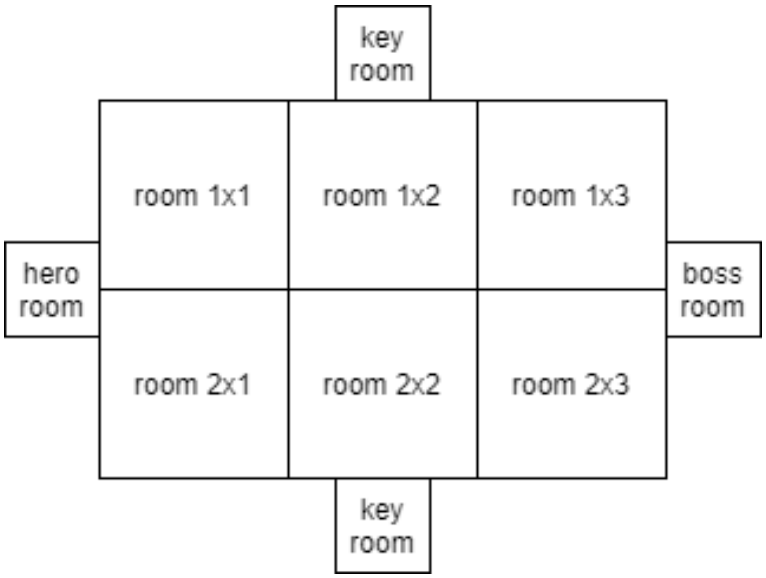

Figura 4.7: Disposizione delle stanze nei vari dungeon.

Per implementare la mappa è stato necessario individuare un modo per rendere le stanze interne intercambiabili rispettando alcuni vincoli:

- Deve essere possibile entrare e uscire liberamente da tutte le stanze
- Deve esistere un percorso che porta dalla stanza dell'eroe a quella del boss
- In ogni dungeon non possono esserci due stanze interne uguali tra loro

Le stanze sono state modellate con un numero fisso di uscite(8, una a nord e a sud e due a est e a ovest) e per quel che riguarda il collegamento tra le stanze è stato creato un insieme di bordi interni che chiudono alcune di queste uscite, lasciandone altre aperte: in questo modo il bordo definisce quali stanze

[4]Dimensione in numero di tile, ognuno composto da 8x8bit

sono collegate e attraverso quale uscita è possibile passare da una a un'altra. I dungeon sono quindi differenti tra loro per quel che riguarda le mappe: gli elementi variabili sono le 6 stanze interne, il loro ordine e il bordo interno che le collega.

4.2.4 Hero

L'Hero rappresenta l'entità vivente controllata dal giocatore. I comandi inviati dalla tastiera e dal mouse passano dalla View fino al Model per mezzo del Controller; una volta che il comando raggiunge l'entità, sulla base della tipologia del comando esso può essere inviato all'AttackStrategy o al MovementStrategy per essere processato normalmente oppure scartato. Non tutti i comandi che poi arrivano ai due principali strategy di questa entità, corrispondono ad una azione; ogni strategy decide, sulla base dello stato dell'Hero, come applicare una particolare forza o alterazione fisica.

Questa particolare entità vivente ha anche la possibilità di aggiornare il proprio stato sulla base della fisica e lo stato stesso che essa presenta, ad esempio se sta eseguendo una scivolata e il terreno sotto i suoi piedi improvvisamente finisce, egli stesso cambiarà il proprio stato interno da *Sliding* a *Falling*.

Inoltre, ogni attacco dell'Hero crea una nuova entità mobile capace di rotarsi e spostarsi in modo da colpire i nemici, per poi essere eliminata ogni volta che termina un attacco.

4.2.5 Enemy e gestione degli stati

I nemici presenti all'interno di un livello sono una particolare classe di entità viventi. Essi sono caratterizzati da un insieme di comportamenti, talvolta anche per una singola entità, complessi e variegati. Anche il boss di fine livello rientra in questa classificazione.

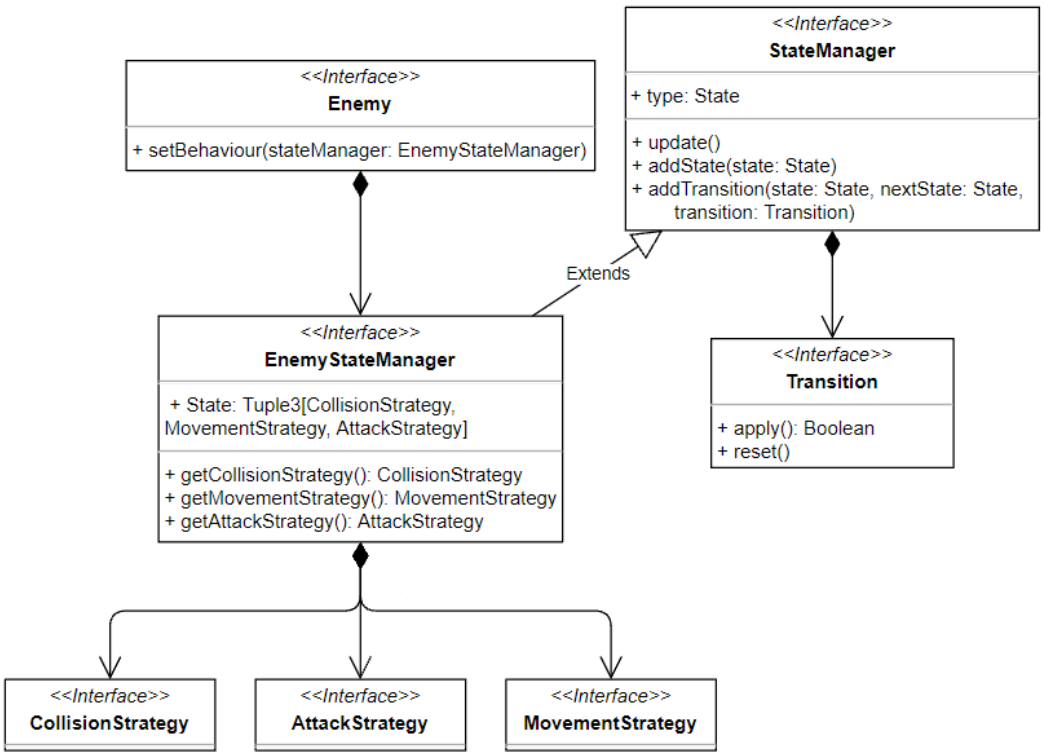

Figura 4.8: Diagramma della gestione dello stato delle entità nemiche.

Data la mutabilità delle strategie di un nemico è stato necessario sviluppare il concetto di *comportamento*: esso è caratterizzato da un CollisionStrategy, un MovementStrategy ed un AttackStrategy. Inoltre, questa tripla di strategie definite su una singola entità può mutare autonomamente durante la partita: in questo caso è stata utilizzatauna struttura già introdotta con i MovementStrategy, ossia il pattern State. In questo caso, la classe Enemy si compone di uno StateManager; è interessante notare come la definizione di un MovementStrategy di un Enemy possa essere definita a livello di StateManager oppure a livello di MovementStrategy per poi essere semplicemente utilizzato da uno EnemyStateManager meno complesso.

4.2.6 Environment Interaction

Le interazioni ambientali sono rappresentate da una coppia *Comando - Interazione*, che vengono assegnate all'Hero quando egli entra in contatto con una entità che può fornire tale interazione e rimosse al termine del contatto. Quando all'entità giocante, verrà notificata la premuta del comando corrispondente, sarà eseguita l'interazione ambientale assegnata che, come nel caso della

Ladder Interaction, può comportare un cambiamento nel MovementStrategy e AttackStrategy dell'Hero. La maggior parte delle interazioni vengono abilitate dalla collisione dell'Hero con sensori posti ai lati di ogni elemento ambientale (porte, piattaforme, ecc..) e disabilitate se si interrompe la collisione o si esegue l'interazione.

4.2.7 Item

Gli oggetti sono considerati a tutti gli effetti entità immobili che offrono un punteggio. E' stato utilizzato il pattern Subclass Sandbox, che consiste in una classe astratta (sandbox class) da cui estendono varie classi (sandboxed subclasses) che implementano i metodi sandbox utilizzando i metodi non astratti già definiti. Ci sono varie categorie di oggetti già definite in 3.5, ognuna di esse corrisponde una ItemPool (un insieme di oggetti dello stesso tipo). Ogni ItemPool è associata ad un modo diverso di ottenimento, per esempio, dai nemici è possibile (con una probabilità del 10 percento) che vengano droppate delle pozioni curative alla loro morte, mentre trovando le stanze del tesoro è assicurata la presenza di un equipaggiamento o la chiave al loro interno. Gli oggetti vengono scelti casuale da ogni Pool e non sono previsti duplicati.

Capitolo 5

Implementazione

5.1 Scala

In questa sezione veogno esplorati i principali utilizzi delle funzionalità più avanzate offerte dal linguaggio Scala:

- **Pimp my library** Tramite questo meccanismo è stato possibile ampliare funzinalità fornite dal framework LibGDX come l'interazione con la rappresentazione fisica del mondo di gioco ma anche aggiungere funzinalità a classi quali Int, Float e Tuple2.

- **Abstract types and family polymorfism** La realizzazione del comportamento di un nemico o di un metodo di movimento, mostrata nelle figure 4.5 e 4.8, ha comportato la definizione di un interfaccia StateManager la quale generalizzasse il meccanismo di gestine degli stati, implementando una rudimentale macchina a stati finiti.

Più nel dettaglio, la realizzazione dell'interfaccia StateManager non considera volutamente il contenuto reale dello stato da gestire: infatti, questa interfaccia definisce solamente un tipo astratto e le meccaniche per la sua gestione (cambiamento di stato, analisi delle transizioni disponibili per uno stato) instanziato realmente da altre interfacce che la estendono, rispettivamente da `MovementStateManager` che definisce come stato un particolare MovementStrategy ed anche `EnemyStateManager` che invece definisce come tale la tripla [CollisionStrategy, MovementStrategy, AttackStrategy].

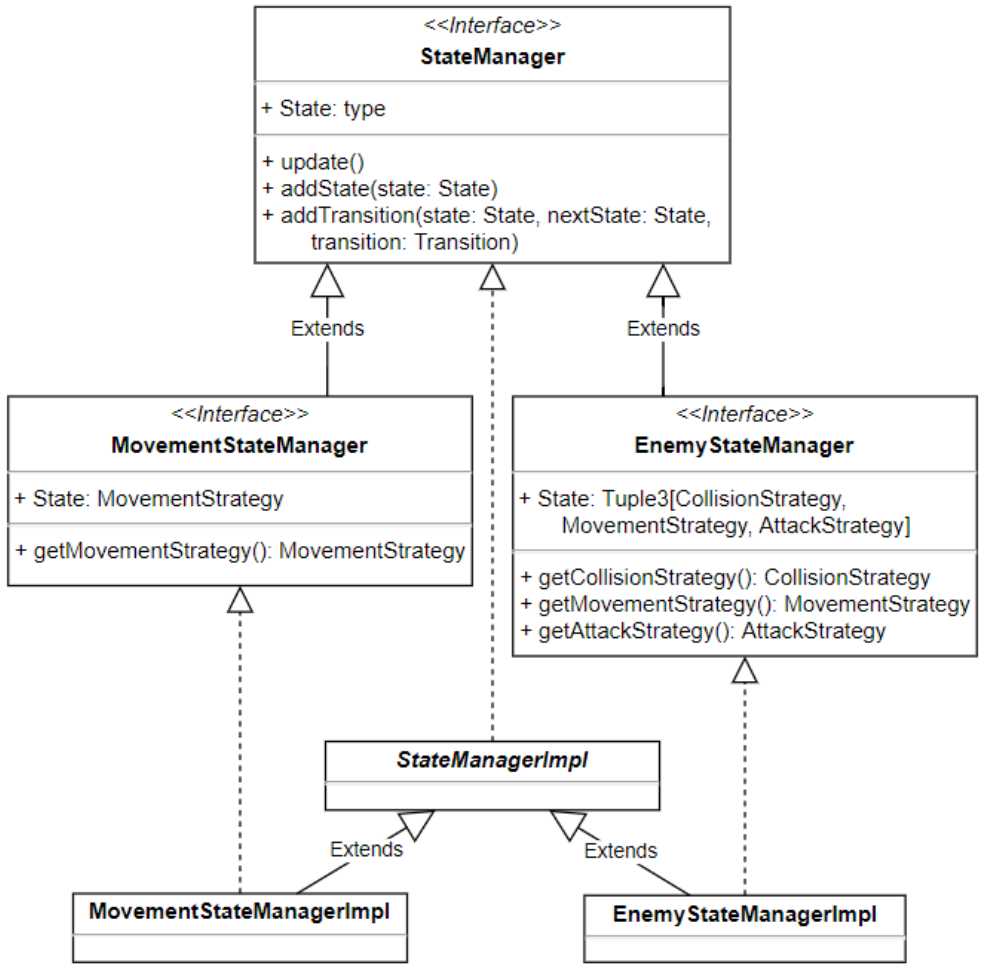

Figura 5.1: Polimorfismo ad-hoc per la gestione di stati.

5.2 Suddivisione dei lavori

5.2.1 Luca Bazzocchi

I principali ruoli che ho avuto in questo progetto sono stati: la realizzazione del personaggio principale, l'Hero, e la creazione della componente grafica riguardante Hud e menu di gioco.

Ho cercato di immedesimarmi nel ruolo di giocatore, distaccandomi da quello del programmatore, per cercare di avere un personaggio divertente ed entusiasmante da giocare; anche prendendo ispirazione da altri giochi, tra cui Dead Cells per la decisione delle mosse da usare.

Sin dalle fasi di design, insieme agli altri membri del gruppo, ho cercato di rendere l'Hero il più mutevole possibile, in modo che, in qualunque momento, si fosse in grado di cambiargli le strategie di moviemento, di attacco, di collisione, la sua grandezza e hitbox, le sue interazioni con l'ambiente e la possibilità di ottenere oggetti che gli sbloccano nuove abilità. Per i suoi movimenti sono stati utilizzati tre diversi movement strategy: uno per il normale comportamento, un altro per l'attesa di un determinato evento e che non gli permettesse di eseguire alcuna azione ed infine un ultimo per salita e discesa di scale e liane.

Per quanto riguarda la parte grafica, nelle animazioni, ho utilizzato degli sprite gratuiti che poi ho composto e animato attraverso l'utilizzo di LibGdx. La creazione di menu e Hud ha invece richiesto la realizzazione di tutte le sue texture e di una GUIFactory per font, pulsanti, scritte e gli altri componenti grafici da visualizzare a schermo.

Prolog

Nel comportamento dell'Hero ho integrato una parte in prolog, specificatamente nel controllo comando-stato all'interno del normale e ladder movement strategy. Attraverso prolog viene verificato se il comando arrivato dalla View al player, possa essere o meno eseguito sulla base dello stato corrente dell'entità. Per ogni tipologia di comando viene chiamata una specifica funzione per controllare se soddisfa o meno quel goal.

Scrum Master

Nella mia esperienza come Scrum Master ho cercato sin dall'inizio del progetto di stimolare gli altri membri nel cercare di ottenere un buon prodotto e di rispettare le scandenze. Ho anche cercato proporre diverse tecnologie da utilizzare nello sviluppo del gioco avendo già avuto qualche esperienza con il progetto *ManPac* alla triennale.

Il lavoro come Scrum Master è durato per tutta la durata del progetto con una maggiore partecipazione durante i meeting settimanali e, alla fine di esso mi ritengo soddisfatto di ciò che ne è venuto fuori.

5.2.2 Luca Bracchi

Mi sono occupato delle mappe: dal design globale del dungeon come insieme di stanze comunicanti a quello di dettaglio delle singole stanze. Come tileset ho usato uno di quelli offerti come esempio dal toolkit LDtk, espandendolo con alcuni tile per poter cambiare i temi delle singole stanze.

Ho creato gli object layer tramite Tiled e li ho gestiti lato codice in maniera tale da chiamare, in base al nome del layer, l'API giusta necessaria per istanziare l'entità giusta.

Nel codice ho istanziato le mappe usando le API di LibGDX e impostando gli offset in maniera tale da ottenere la disposizione desiderata.

Ho iniziato disegnando i bordi e in seguito mi sono ritagliato un po' di tempo ad ogni sprint per disegnare una o due mappe nuove da aggiungere e facendo in modo da differenziarle il più possibile, sia visivamente che in termini di difficoltà.

Vista la difficoltà riscontrata durante i tentativi di automazione dei test relativi alle mappe, ho deciso di effettuare i test del caso a mano, verificando quindi le feature in maniera diretta, giocando.

Personalmente sono soddisfatto del risultato ottenuto, ma se avessi avuto più tempo mi sarebbe piaciuto rendere variabile anche la forma della griglia delle mappe, evitando di creare sempre la solita 2x3 in favore di qualche disposizione più interessante.

5.2.3 Giacomo Casadei

Io mi sono dedicato al sistema degli oggetti e agli elementi ambientali con cui è possibile interagire. Per gli oggetti mi sono ispirato al sistema presente in The Binding of Isaac, ossia basato su diverse item pool a esaurimento (ossia, senza la possibilità di ottenere dei duplicati). Le descrizioni degli oggetti visualizzate al momento della raccolta sono (per la maggior parte) citazioni a iconiche frasi della storia videoludica.

Per gli elementi ambientali ho cercato di ricrearli e di interagirvi nel modo in cui un giocatore medio si aspetterebbe, per esempio il passare attraverso le piattaforme sottili dal basso oppure dall'alto alla pressione del pulsante di azione (come in buona parte dei giochi platform o Metroid-Vania), oppure il subire danno se si finisce nella lava o ancora essere rallentati all'interno dell'acqua.

Per la parte grafica ho utilizzato sprite gratuiti messi a disposizione su OpenGameArt.com e composti (a volte anche ritoccati con Photoshop) in animazioni tramite LibGdx.

Sinceramente, avrei potuto fare di più se non ci fosse stato il lavoro di mezzo (più oggetti per esempio), ma mi ritengo abbastanza soddisfatto del risultato e soprattutto soddisfatto nell'utilizzo di questa metodologia di suddivisione e svolgimento dell'intero processo.

5.2.4 Fabio Muratori

I miei compiti hanno compreso principalmente la strutturazione delle entità nemiche e la definizione dei vari metodi strategy trattati rispettivamente in 4.2.5 e 4.2.2.

Inizialmente mi sono concentrato sulla realizzazione di semplici entità nemiche senza preoccuparmi di aspetti strutturali come la gestione degli stati. Ad ogni entità è stato associato in un primo momento un attacco e movimento decisamente banali ma che servissero a consolidare l'esplorazione di questo dominio. Ho tentano più volte di adottare una metodologia TDD ma mi è risultato difficile ragionare in maniera opposta a quanto fatto fino ad ora, complice anche il fatto che fino a fasi inoltrate della realizzazione ero indeciso su come definire efficacemente la struttura dei comportamenti dei nemici. In seguito, con la definizione della struttura mostrata in 4.2.5 mi è stato sia più agevole configurare interazioni interessanti tra nemici e giocatore ma anche testare in maniera efficace i singoli componenti che compongono una strategia di movimento.

Avendo necessità di interagire costantemente con funzionalità del framework LibGDX (specialmente con l'entità World e oggetti Body) ho dovuto definire una serie di strumenti di utilità per l'arricchimento di quanto offerto dal framework utilizzato.

Nella realizzazione delle varie funzionalità ho avuto modo di interagire con molti aspetti trattati dagli altri membri del team, fornendo supporto per l'implementazione di alcuni componenti. Parallelamente allo sviluppo delle entità nemiche ho avuto modo anche di apportare migliorie alla gestione delle *sprite* relative a tutte le entità di gioco, ristrutturando il codice già implementato.

Product Owner

L'attività di Product Owner ha richiesto anche la necessità di dirigere e monitorare l'avanzamento dello stato di sviluppo e l'interazione con un ipotetico cliente (i membri stessi del team) coerentemente con le metodologie di sviluppo adottate ed esplorate nel Capitolo 1. Una prima bozza dei task definiti nel **Product Backlog** è stata definita, successivamente vagliata dai membri del team e modificata secondo le esigenze. Inizialmente è stato deciso di non affrontare tematiche troppo avanzate ma invece di dettagliare le attività ancora da svolgere con una finestra di 1-2 sprint. Talvolta la gestione dei task e la conduzione dei singoli sprint ha trovato ostacoli derivati da cause esterne al team.

Capitolo 6

Retrospettiva

In questo capitolo viene fornita una sintesi del percorso seguito dal progetto in termini di sprint effettuati e di stato del progetto al termine della settimana lavorativa.

Anche se non riportato esplicitamente, ogni sprint ha richiesto una certa quantità di ore-lavoro per il processo di rifattorizzazione del codice, necessario per mantenere il codice leggibile e semplificare il processo di integrazione dei vari branch sul branch di sviluppo.

6.1 Avviamento

La prima fase è stata quella di avviamento: ci si è concentrati sulla stesura dei requisiti, sull'analisi del problema, sulla definizione dei principali componenti del sistema e sulla costruzione di un'intelaiatura di progetto da dove partire con i primi sviluppi, previsti per il primo sprint.

Sono stati progettati alcuni diagrammi UML che descrivessero il sistema a diversi livelli di astrazione e dettaglio, è stato redatto il Product Backlog ed è stato impostato il repository GitHub contenente una prima impostazione del sistema, in particolare sono stati preparati gli strumenti SBT e LibGdx.

In questa fase i membri del gruppo già formati su alcune delle tecnologie usate, come LibGdx, LDtk e Tiled, hanno istruito i membri restanti sulle funzionalità offerte da questi strumenti e su come utilizzarle, in modo da partire fin dall'inizio con un'idea più nitida delle funzioni che sarebbero tornate utili durante i prossimi sviluppi.

Come fase è risultata essere la più costosa dell'intero processo in termini di tempo, ma ha permesso al team di sviluppo di avere le idee chiare sul cosa fare e come farlo, dalle prime fasi fino all'ultimo sprint, minimizzando il numero di ripensamenti.

6.2 Sprint 1

Periodo: 9 agosto - 14 agosto.

Nello sprint iniziale è stata impostata la struttura di massima del progetto, impostando i componenti del pattern Model View Controller e realizzando una schermata di gioco rudimentale con una prima bozza di generazione di entità, animazioni e fisica di gioco.

Lo sprint ha prodotto un sistema primordiale che ha sollevato alcune questioni marginali che non erano emerse dall'analisi iniziale e sono state espanse e tradotte in items durante la Sprint Retrospective.

Un aspetto che non è stato possibile risovere è stato la configurazione dell'ambiete di Countinuous Integration, il cui task è stato propagato nello sprint successivo.

6.3 Sprint 2

Periodo: 16 agosto - 22 agosto.

Nel prodotto del secondo sprint sono stati introdotti il supporto completo agli input da tastiera, un'HUD rudimentale, l'assegnazione degli sprite alle varie entità e la creazione con conseguente posizionamento in mappa di nemici e oggetti.

Sono state introdotti insiemi di strategy diversi: le attack strategy relative agli attacchi dei nemici, le movement strategy relative ai movimenti delle entità mobili e le collision strategy relative alla gestione delle collisioni tra due entità.

6.4 Sprint 3

Periodo: 23 agosto - 29 agosto.

Nel terzo sprint è stata sviluppata una prima mappa di gioco, composta da un livello grafico da renderizzare e un livello contenente le varie entità che popolano la mappa, da creare e posizionare sulla mappa renderizzata.

L'HUD è stato migliorato aggiungendovi la barra della vita dell'eroe, il punteggio attuale e gli oggetti speciali raccolti; sono state aggiunte svariate feature riguardanti il punteggio di gioco e le statistiche di eroe e nemici.

Sono state inoltre aggiunte le porte, la possibilità di interagirvi, l'arco come arma ottenibile e utilizzabile dall'eroe per attaccare i nemici e l'impostazione di un primo sistema di riproduzione suoni relativi a eroe, nemici e musiche di sottofondo.

6.5 Sprint 4

Periodo: 30 agosto - 5 settembre

Nel quarto sprint è stato prodotto un sistema che si avvicinava molto a quello desiderato: è stato introdotto il boss del dungeon, le entità scale, piattaforme, lava, acqua, porta della stanza del boss e relativa chiave, con le varie interazioni previste dai requisiti.

Sono state definite entità puntiformi che corrispondono a punti dove posizionare i nemici alla creazione del dungeon ed è stato implementato il sistema che permette all'eroe, una volta sconfitto il boss, di accedere al portale che lo trasporterà al livello successivo, dove troverà nemici più forti rispetto a quelli nel livello attuale.

6.6 Sprint 5

Periodo: 6 settembre - 12 settembre.

Nello sprint finale il team si è occupato di implementare le schermate del menù principale e di Game Over, di aggiungere un nuovo attacco all'eroe per rendere il combattimento più vario, di rendere più complesso il comportamento del boss e di aggiungere effetti sonori quali musica di sottofondo e suoni relativi alle azioni di eroe e nemici.

La generazione della mappa è stata migliorata, aumentando il numero di stanze per dungeon da 1 a 6, prese in maniera casuale da un insieme di possibili stanze.

In questo sprint si è inoltre sviluppato il sistema di generazione fissa del dungeon in base a un seed casuale: in questo modo il dungeon attuale è replicabile in tutto e per tutto, usando il seed, che se omesso viene generato casualmente.

6.7 Raffinamento

Al termine del 5° sprint è stato possibile produrre una versione definitiva del sistema ideato. Il codice prodotto è stato in seguito vagliato da tutti i membri del team, sono stati individuati potenziali migliorie (un esempio sono l'utilizzo di utilità implicite utili a tutti i membri) ed in generale normalizzate alcune nomenclature. Infatti, avendo ogni membro del team sviluppato una sottoparte del sistema generalmente distinta è stato possibile solo in questa fase individuare delle funzionalità duplicate e fattorizzabili.

Il codice è stato poi analizzato attraverso il plugin automatico `scalatest` per identificare ulteriori anomalie precedentemente sfuggite.

Capitolo 7

Guida utente

All'avvio dell'applicazione, ci si troverà davanti al menu di gioco, in cui è possibile:

- iniziare una nuova partita, cliccando sul pulsante Start ;
- uscire, tramite Exit .

Iniziata una nuova partita, attraverso i pulsanti della tastiera e click sul mouse, sarà possibile far eseguire delle azioni all'*hero* per farlo proseguire all'interno della mappa di gioco.

7.1 Scopo del gioco

Ogni partita è differente e lo scopo del gioco è sopravvivere il più a lungo possibile, accumulando oggetti, potenziamenti e sconfiggendo nemici che incrementeranno lo score della partita. In ogni livello è presente un boss che, una volta sconfitto, attiva il portale per accedere al livello successivo. Il gioco non ha una fine, ad ogni nuovo livello la mappa e i nemici verranno generati casualmente e resi più potenti. Il game over avverrà alla morte del giocatore e, in tal caso, si dovrà rincominciare il gioco dall'inizio, il quale, ancora una volta, presenterà una nuova avventura.

Ogni livello presenta:

- una stanza iniziale, dove il giocatore parte per l'esplorazione;
- due stanze del tesoro contenenti potenziamenti per il player o oggetti speciali. In ogni livello almeno una di queste due stanze, conterrà la chiave per accedere al boss;
- una stanza del boss, bloccata da una porta apribile solamente con una chiave trovabile all'interno del livello;

- nemici che attaccheranno il giocatore e possono far cadere pozioni una volta uccisi;
- oggetti che conferiscono potenziamenti in delle statistiche del giocatore o utilizzabili per proseguire all'interno della mappa;
- un portale per accedere al livello successivo.

7.2 Informazioni e azioni del player

Di seguito sono riportate le informazioni principali sul gioco e sulle azioni attuabili dal player all'interno della mappa di gioco:

Hud Qui sono contenute le informazioni principali di una partita.

Figura 7.1: Hud.

1. Vita del giocatore;
2. Numero del livello;
3. Punteggio accumulanto nella partita corrente;
4. Item speciali, raccolti dal player.

Luoghi speciali Luoghi nascosti all'interno della mappa che contengono potenziamenti o oggetti per proseguire nel livello.

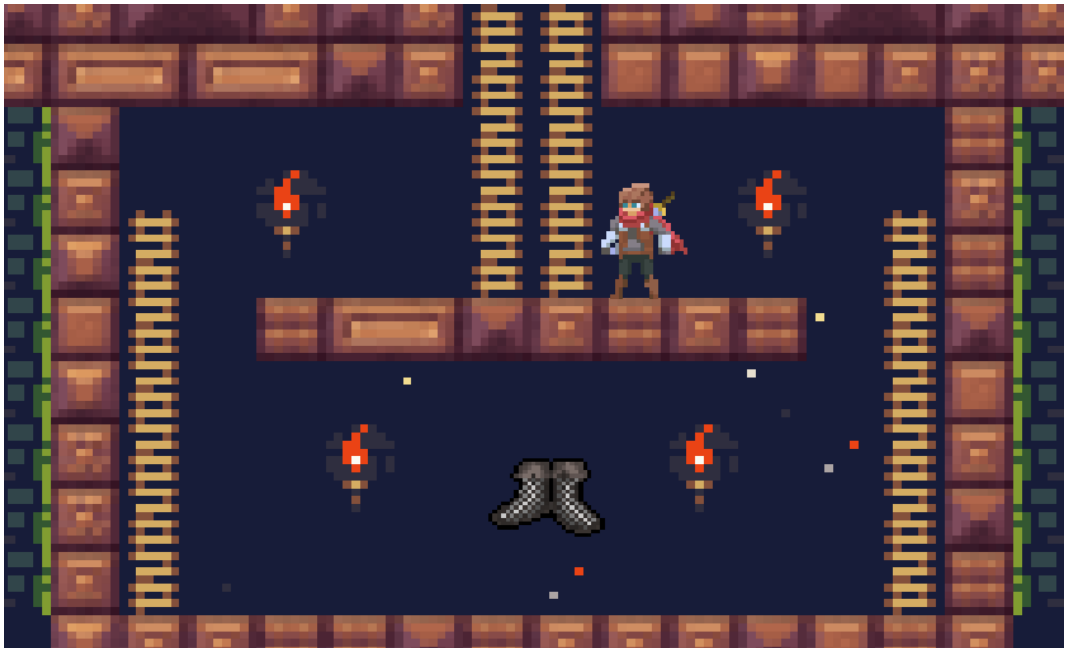

Figura 7.2: Stanza del tesoro con item speciali.

Movimenti del giocatore Azioni per spostarsi all'interno della mappa.

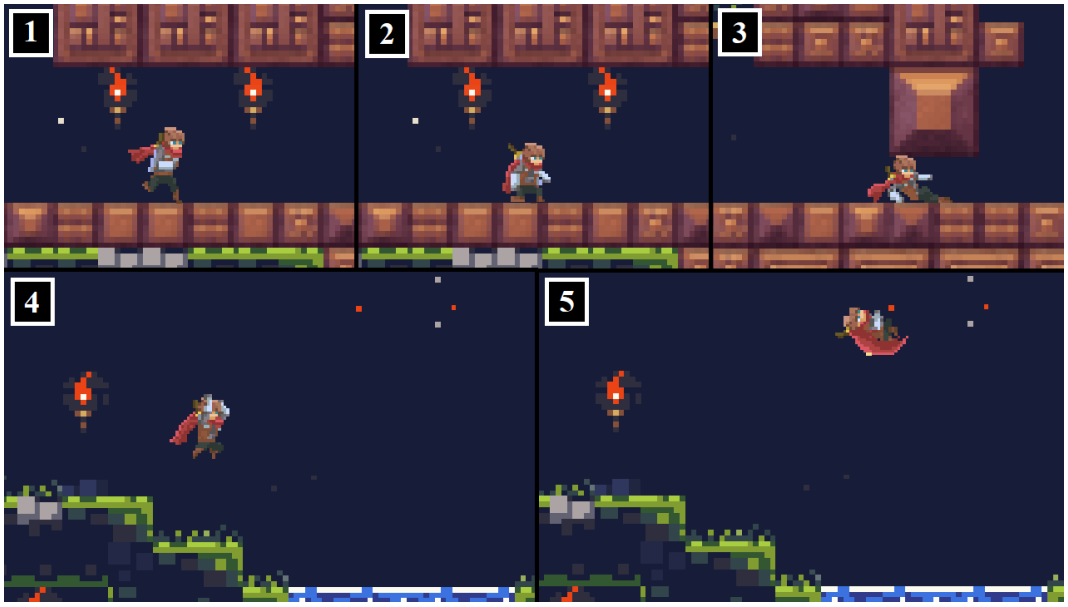

Figura 7.3: Movimenti dell'hero.

1. **Run**: premere A / ← o D / → per muoversi;
2. **Crouch**: premere S / ↓ per accovacciarsi;
3. **Slide**: premere E per la scivolata, essa rende il giocatore invulnerabile dagli attacchi nemici;
4. **Jump**: premere W / ↑ per saltare;
5. **Somersault**: premere nuovamente W / ↑ mentre si sta saltando, per effettuare una capriola in aria e raggiungere posti più elevati.

Attacchi del giocatore Azioni eseguibili contro altre entità viventi che causano danno.

Figura 7.4: Attacchi dell'hero con la spada.

1. **First Sword Attack**: mentre si sta toccando il terreno, premere il pulsante sinistro del mouse per eseguire un attacco con la spada;
2. **Second Sword Attack**: prima della fine del primo attacco, riliccare nuovamente il pulsante sinistro del mouse per eseguire un attacco consecutivo a quello precedente;
3. **Third Sword Attack**: terzo e ultimo attacco, eseguibile dopo il secondo e alla stessa maniera dei precedenti.

Figura 7.5: Attacco aereo e con l'arco.

1. **Air Attack**: mentre si saltando, cliccando con il pulsante sinistro del mouse, è possibile eseguire un attacco aereo dall'alto verso il basso;
2. **Bow Attack**: attacco con l'arco, eseguibile con il click destro del mouse e solo se si è entrati in possesso dell'oggetto *Arco*, trovabile all'interno della mappa. La freccia causa un contraccolpo ai nemici che gli stanno andando incontro.

Interazioni con l'ambiente Azioni che cambiano il comportamento tra hero ed entità interagibile.

Figura 7.6: Interazione con le piattaforme.

1. **Platform interaction**: premere SPACE per scendere dalle piattaforme di legno e raggiungere posti altrimenti irraggiungibili.

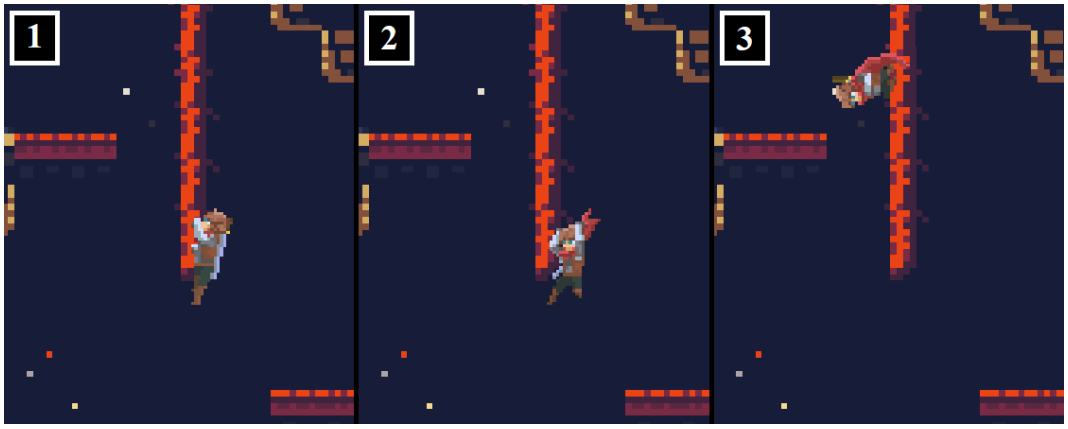

Figura 7.7: Interazione con scale e liane.

1. **Ladder interaction**: premere SPACE per interagire con scale e liane, sarà possibile salire o scendere su di esse;

2. **Ladder stop interaction**: premere SPACE mentre si è su liane o scale per terminare l'interazione;
3. **Ladder jump**: premere SPACE mentre si sta salendo per interrompere l'interazione ed eseguire automaticamente un salto in aria.

Conclusioni

Il progetto ha permesso ai membri del gruppo di affrontare in maniera diretta i principali aspetti di gestione, progettazione e sviluppo di un sistema software non banale e di addentrarsi in quello che è il mondo della programmazione di progetti videoludici, che offre svariate opportunità lavorative interessanti.

Si è inoltre avuto modo di operare adottando Scrum e la metodologia agile per l'orchestrazione dei lavori, così da vedere in pratica l'utilità delle riunioni effettuate e degli artefatti prodotti durante gli sprint.

Oltre alle metodologie utilizzate, il gruppo ha avuto modo di applicarsi utilizzando il linguaggio Scala e il paradigma di programmazione funzionale, oltre alle tecnologie già citate in precedenza.

Visti il comune interesse dei membri del team al mondo videoludico e la natura del sistema da realizzare, è stato semplice impiegare inventiva e fantasia in ogni parte del progetto, rendendo il lavoro più interessante, dalla progettazione all'implementazione delle varie funzionalità. Sarebbe stato interessante espandere alcune delle feature realizzate o aggiungerne di nuove, ma vista la quantità di tempo allocabile per il progetto è stato necessario selezionare attentamente cosa sviluppare e cosa no.

Per concludere, il gruppo ha trovato il bilanciamento ottimale nella suddivisione dei lavori: questo ha permesso di lavorare in un ambiente tranquillo senza far emergere problematiche significative.