DISI's catacomb report

Chelli M., Monti G., Sanità R., Tampieri E.

24 aprile 2021

Indice

Capitolo 1

Analisi

1.1 Requisiti

1.1.1 Requisiti funzionali

Il team si pone l'obiettivo di realizzare un gioco 2D roguelike, ovvero caratterizzato dall'esplorazione di livelli generati proceduralmente di un dungeon, da un gameplay a turni, da grafica tile-based e la morte permanente del giocatore [3], simile ai giochi Enter the Gungeon e Nuclear Throne.

Il giocatore dovrà affrontare una serie di stanze contenenti nemici e oggetti fino ad arrivare a un boss finale che sarà visibile solamente una volta uccisi tutti i nemici base. La mappa di gioco sarà generata in modo casuale e sarà casuale anche la stanza in cui il personaggio comincerà la sua avventura.

Inizialmente il protagonista avrà il 100% di vita e un arma base che potrà cambiare con armi più avanzate che troverà nel corso della partità. Ad ogni colpo subito il personaggio perderà una percentuale della vita, se il personaggio muore la partita terminerà lasciando al possibilità all'utente di poter cominciare un altra partità in una nuova mappa.

Il gioco può terminare anche se viene sconfitto il boss della mappa ovvero un nemico di dimensione maggiore rispetto a quelli incontrati nelle stanze precedenti. Il boss possiederà anche abilità superiori.

Se il giocatore si trova in difficoltà può trovare sparse per la mappa di gioco alcune pozioni curative che verranno lasciate sporadicamente dopo l'uccisione dei nemici base, anche la quantità di vita rilasciata dalle pozioni è casuale.

Esistono 2 tipi di nemici base:

- Nemici che attaccano da lontano
- Nemici che attaccano da vicino

Il gioco si basa sull'abilità del giocatore ma anche su una percentuale di fortuna nel trovare vite, armature ed armi che lo aiuteranno a sconfiggere il boss finale.

1.1.2 Requisiti non funzionali

- Il gioco dovrà risultare fluido e reattivo anche su macchine con hardware non recenti.
- Il gioco dovrà avere una grafica e comandi chiari e intuitivi.

1.2 Analisi e modello del dominio

Il sistema gestisce la generazione delle mappe con le varie stanze e le interazioni tra il personaggio e i nemici.

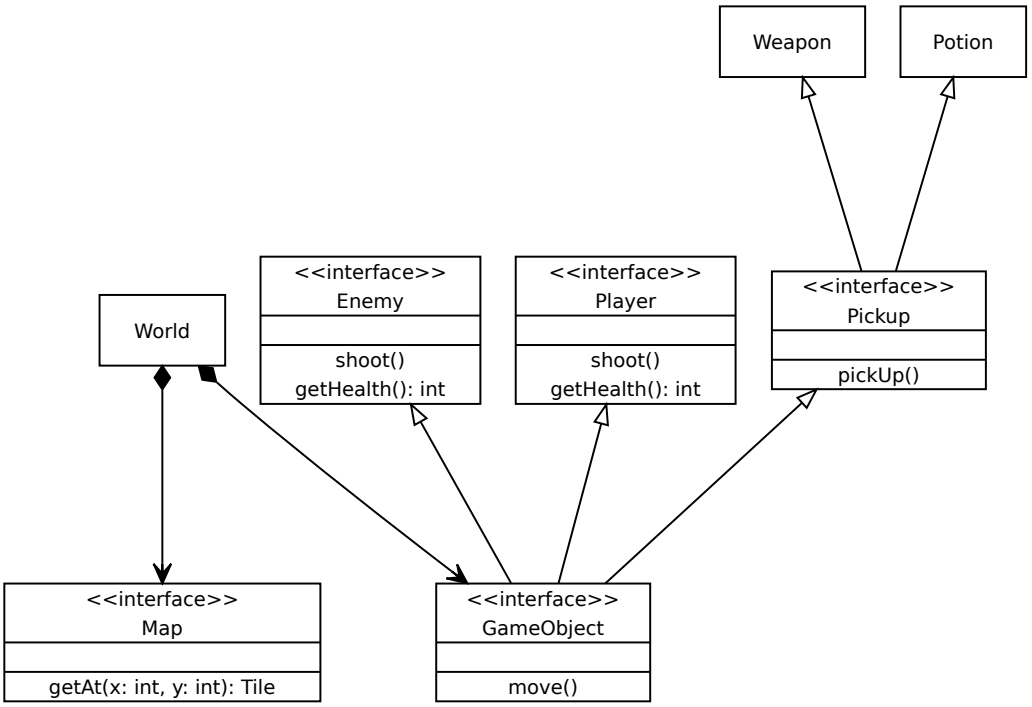
Oltre ai nemici il personaggio potrà interagire con oggetti trovati nelle varie stanze.

Il personaggio possiede principalmente una percentuale vite che aumenteranno con le pozioni o diminuiranno se colpiti da un nemico.

I nemici potranno essere di due tipi principali, melee (che attaccano da vicino) e ranged (che attaccano da lontano) e si muoveranno all'interno delle stanze e avranno una vita massima. Nemici più difficili da sconfiggere, come i Boss avranno una maggiore vita massima e mosse speciali che verrano usate durante il combattimento.

La mappa è divisa in più stanze di dimensione variabile, unite da corridoi, dove compariranno nemici e/o oggetti utilizzabili dal giocatore. Una volta sconfitti tutti i nemici presenti nella mappa il personaggio dovrà affrontare il Boss e dopo il combattimento il gioco terminerà.

Gli elementi considerati nel modello sono sintetizzati sotto

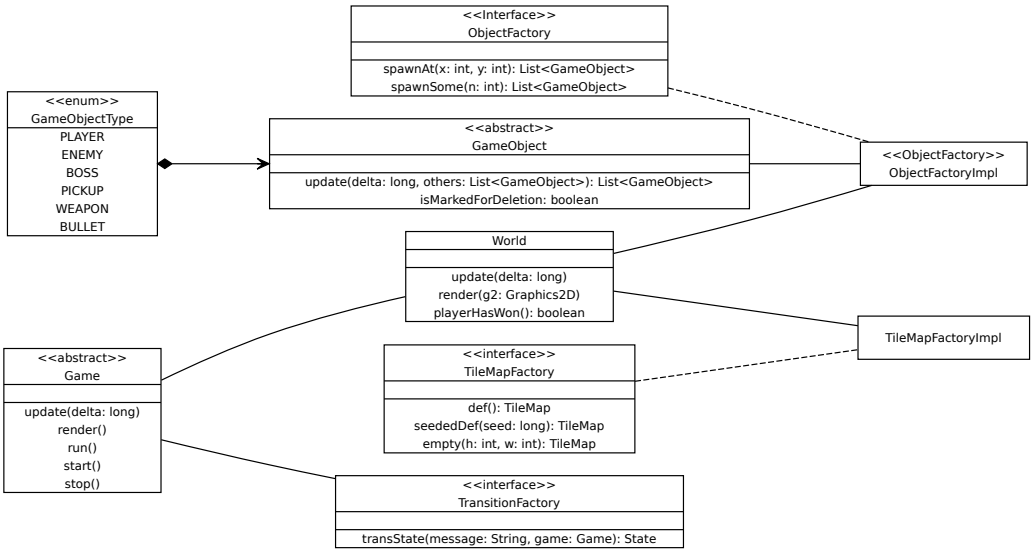

Weapon
Potion
<<interface>>
Enemy
shoot()
getHealth(): int
<<interface>>
Player
shoot()
getHealth(): int
<<interface>>
Pickup
pickUp()
World
<<interface>>
Map
getAt(x: int, y: int): Tile
<<interface>>
GameObject
move()

Capitolo 2

Design

2.1 Architettura

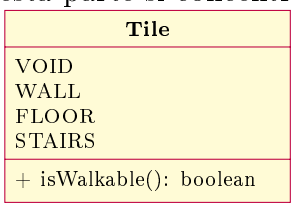
L'architettura di 'D.I.S.I's Catacombs' è stata pensata e realizzata seguendo una forma modificata e naive del pattern ECS (Entity–component–system) [2]. Questo pattern architetturale permette la definizione della entità di gioco, identificata solamente da un id. Queste enità hanno variabili che ne definiscono l'aspetto e l'interazione con il mondo di gioco. Il sistema itera tra le entità periodicamente durante il game loop, aggiornando le loro componenti fisiche (come posizione della entità, dimensioni e movimento) e grafiche (la sprite o l'animazione). In un pattern ECS standard più sistemi adempierebbero a compiti diversi, ma per diminuire la complessità dell'implementazione e favorire un codice più chiaro al gruppo si è deciso di implementare un solo sistema che esegue tutte le azioni sulle componenti delle entità e gli eventi di gioco. Questa genere di architettura garantisce una maggiore flessibilità nelle componenti delle entità e garantisce modifiche a runtime di quest'ultime. Il sistema infatti durante le iterazioni tra le entità ha la possibilità di gestirne e di eseguire la logica dei loro componenti utilizzando filtri di vario tipo. Questo permette l'integrazione di metodi di alto livello come stream, lambda, ecc.... Gli input servono a gestire unicamente il movimento e lo sparo del Player. Il lato negativo dell'architettura da noi usata è che il gioco può diventare molto pesante e lento dopo un elevato numero di entità generate e visualizzate durante l'esecuzione.



2.2 Design dettagliato

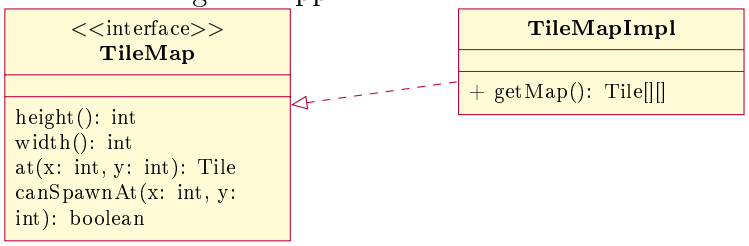
2.2.1 Chelli

Questa parte si concentrerà sugli aspetti relativi alla mappa di gioco.



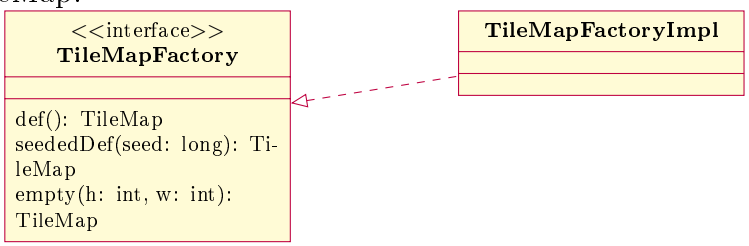
La mappa e' concettualmente una matrice di caselle, che possono essere pavimento o muro.

Le caselle vengono rappresentate da un enum chiamato Tile.



Per rappresentare la mappa, ho deciso di usare un interfaccia chiamata TileMap. Quest'interfaccia ha le poche funzioni necessarie per descrivere la rappresentazione concettuale descritta in precedenza. Anche se questa interfaccia ha una sola classe che la implementa, serve per astrarre l'implementazione che potrebbe cambiare col tempo e per lasciare spazio a nuove

implementazioni possibili. L'implementazione attuale utilizza una matrice di Tile siccome mi sembrava la rappresentazione piu vicina al concetto di TileMap.

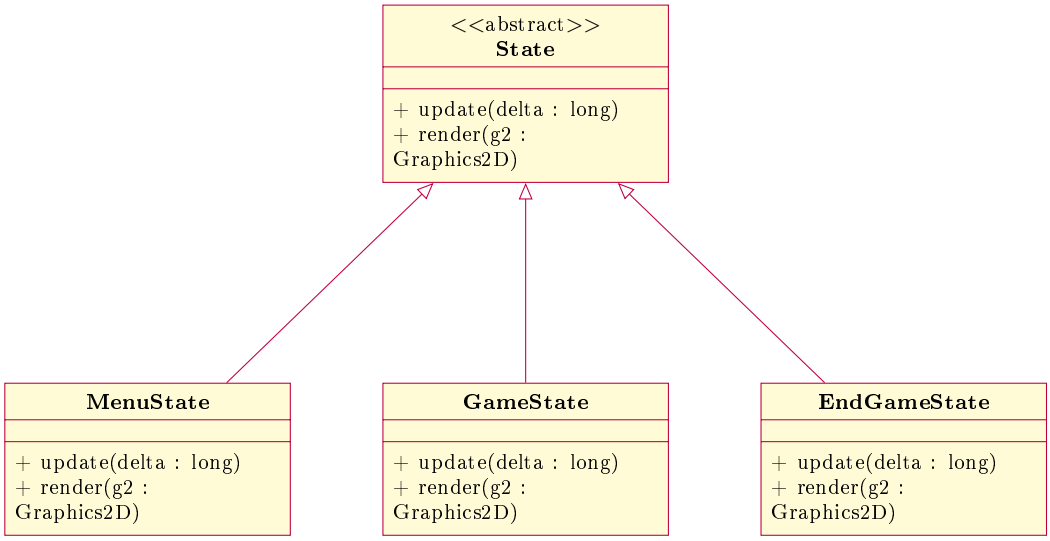


Per la creazione delle TileMap ho deciso di usare una Factory, anche questa divisa in interfaccia e implementazione. La Factory fornisce metodi per avere una TileMap con parametri default, un metodo simile per fare la stessa cosa ma con un seed dato, e un metodo per una TileMap vuota, usato per testing.

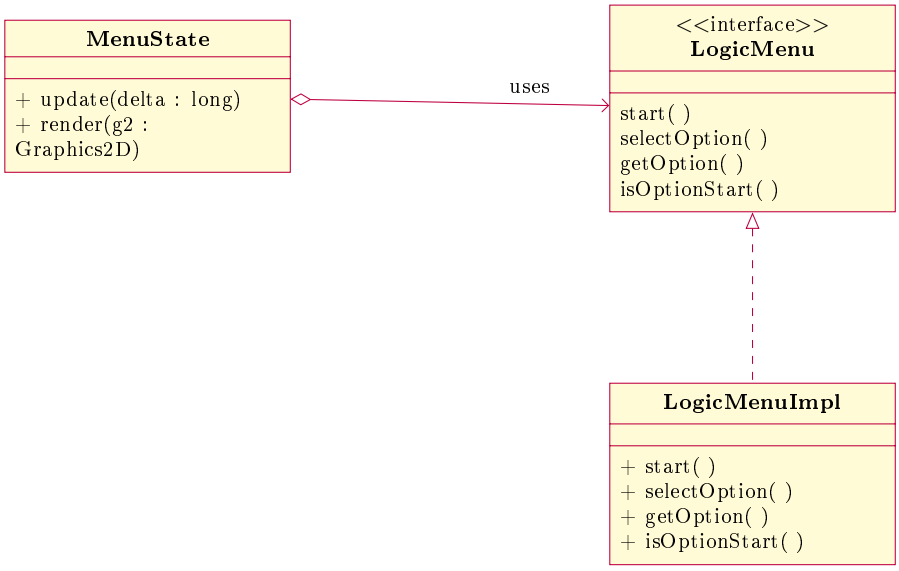
L'implementazione della Factory, e' divisa in diverse funzioni private in quanto una singola funzione risultava troppo lunga e difficile da leggere. Tutte le TileMap richieste dall'interfaccia della Factory, possono essere costruite con la stessa funzione internamente, a parte quella vuota usata per il testing.

2.2.2 Sanità

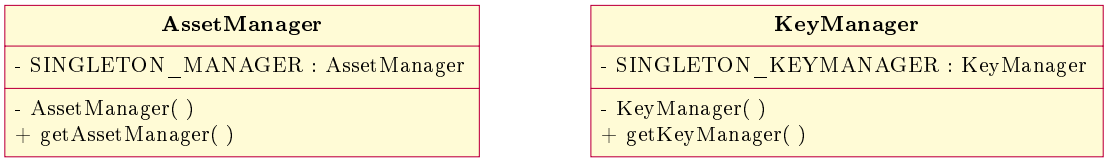
Questa parte si concentrerà sugli aspetti relativi ai vari stati che costituiscono il gioco (menu, stato di gioco, stato di fine). Il sistema per la creazione di questi stati è la creazione di una classe astratta State dalla quale poi verranno estese tutte le classi che costituiscono i diversi stati. Questa classe rappresenta lo scheletro di un qualunque stato ed essendo una classe astratta l'implementazione di due metodi astratti quali update e render sarà diversa da stato a stato in modo che la logica e la grafica di ogni stato risulti diversa.



Questa parte verterà sulla gestione della schermata principale e su quella di fine gioco la quale poi introdurrà nuovamente alla schermata di menu. Il sistema per la gestione di questi due stati è la medesima, entrambi utilizzano il pattern Strategy infatti è possibile modificare la grafica di queste schermate senza andare ad impattare direttamente la logica che le gestisce. Questa scelta è stata fatta principalmente per evitare di infrangere SRP (Single Responsibility Principle) e quindi di incaricare una classe sia della gestione grafica di una schermata sia della sua logica creando così una complicazione in un futuro cambiamento.

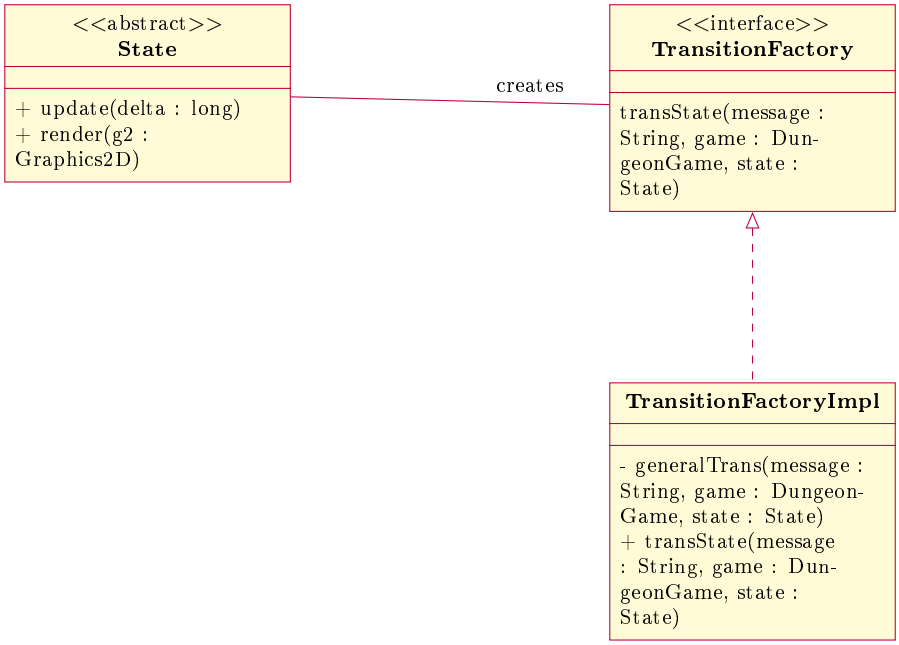


Questa parte riguarda la gestione degli asset più precisamente la creazione di strutture e il caricamento delle immagini all'interno di esse. Il pattern utilizzato in questo caso è Singleton permettendo che ci sia un'unica istanza di AssetManager accessibile globalmente senza doversi preoccupare di fornire il riferimento a chi lo richiede. Questa scelta è stata fatta perché il caricamento delle immagini in quelle strutture basta farla una volta sola per poi permetterne l'accesso a chiunque lo richieda. Lo stesso ragionamento è stato fatto per il KeyManager essendo anche esso di utilità a più classi evitando anche in questo caso la creazione di istanze superflue.



Questa parte di codice si occupa invece della cambiamento di stato ovvero delle transizioni che portano da uno stato a l'altro. In questo caso è stato utilizzata una Factory di transizione per via del fatto che la forma delle transizioni è la medesima le uniche differenze riguardano gli stati di partenza e di arrivo e il messaggio che il giocatore vedrà. In questo non utilizzando una Factory avrei avuto tre classi in cui quasi tutto il codice veniva ripetuto e solo poche parti sarebbero cambiate e così facendo avrei violato una regola base

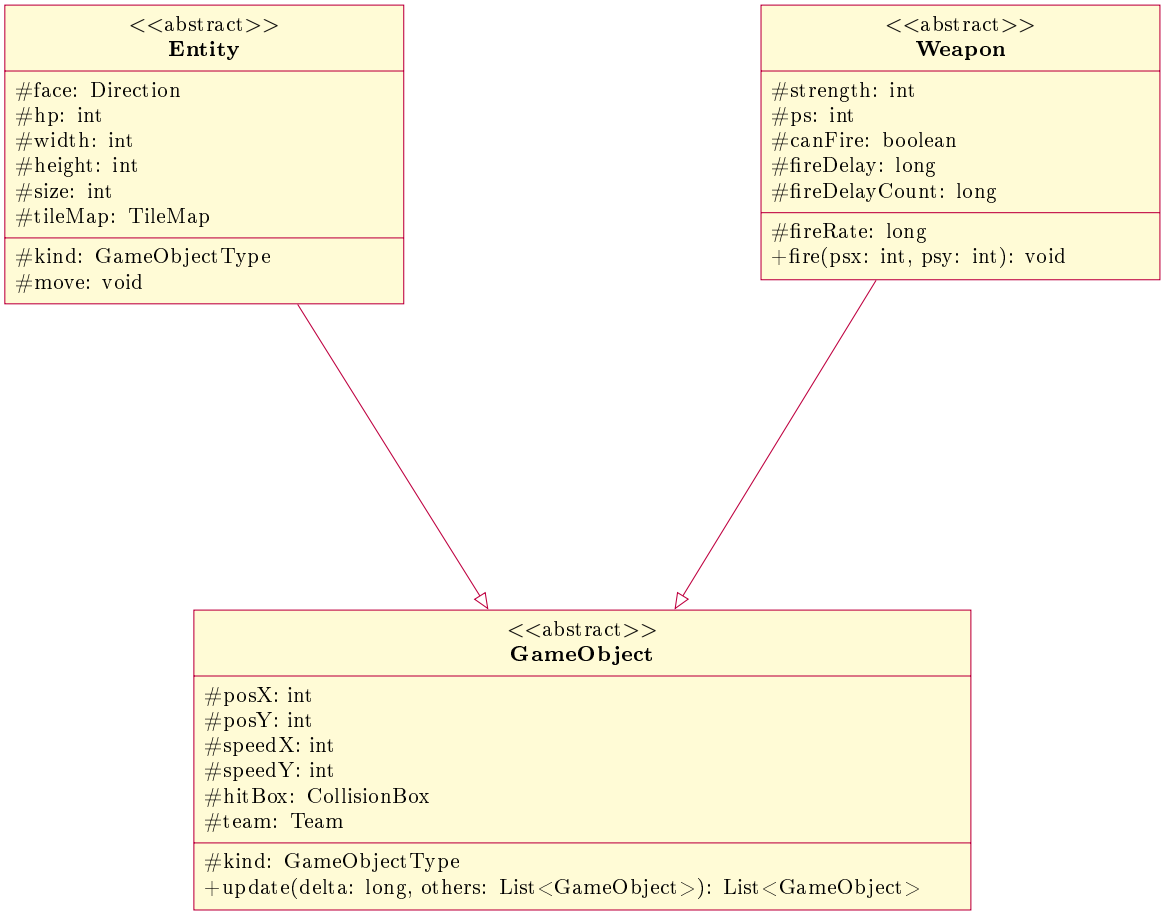
generale di buona programmazione ovvero DRY (Don't Repeat Yourself).



2.2.3 Monti

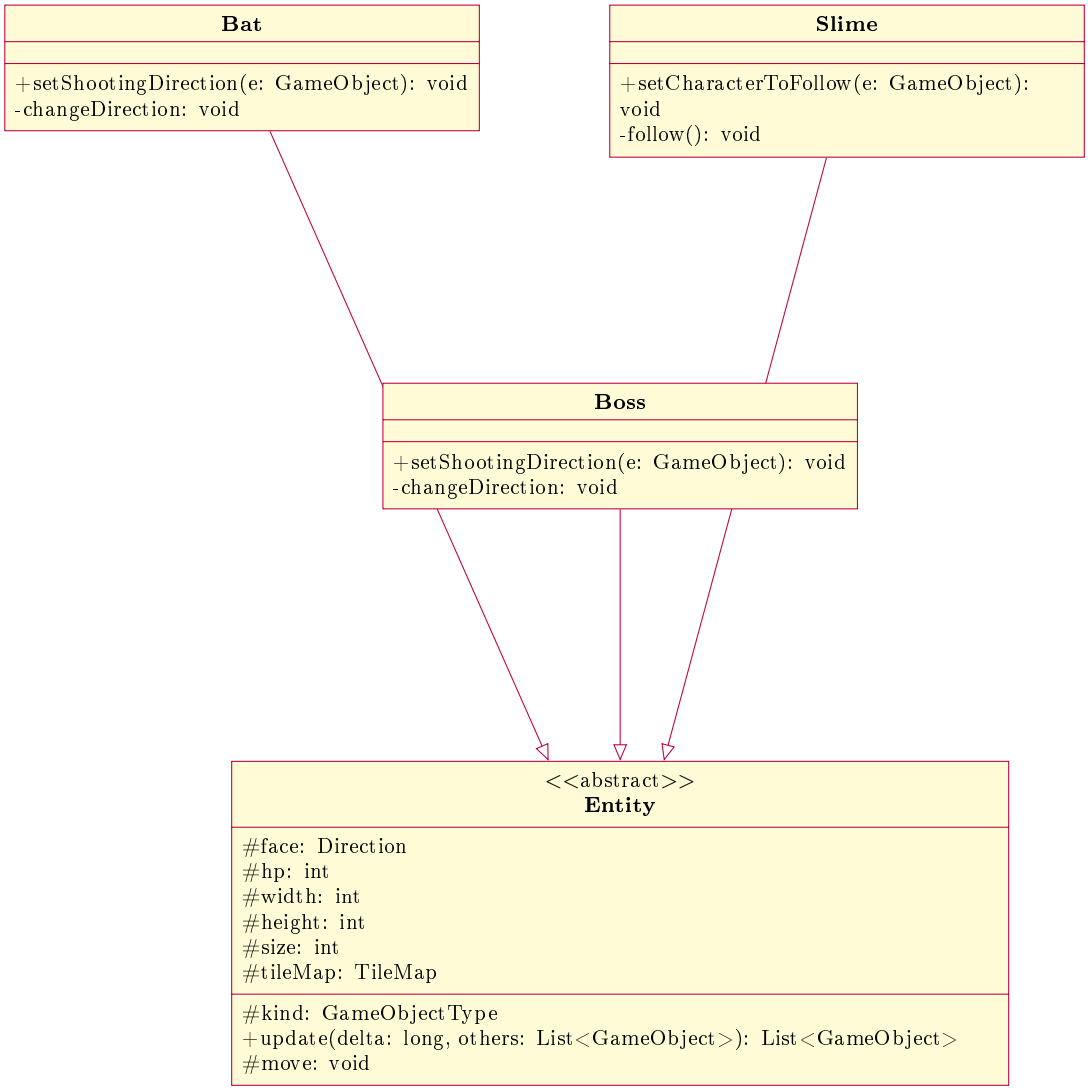
In questa sezione ci si concentrerà sugli aspetti riguardanti gli oggetti e le entità del gioco e della loro generazione.

Dovendo creare oggetti con elementi in comune ho deciso di utilizzare un Template Method che ha come template la classe astratta GameObject Questa ha lo scopo di identificare un oggetto di gioco caratterizzato da poche elementali caratteristiche come la posizione nella mappa, la velocità e le dimensioni che esso ha. Inoltre viene specificato il metodo update nel quale gli oggetti di gioco aggiorneranno i loro stati, si muoveranno e/o spareranno o compieranno altre azioni. Il template viene poi esteso alle classi Entity, Weapon, SimplePotion e Projectile.

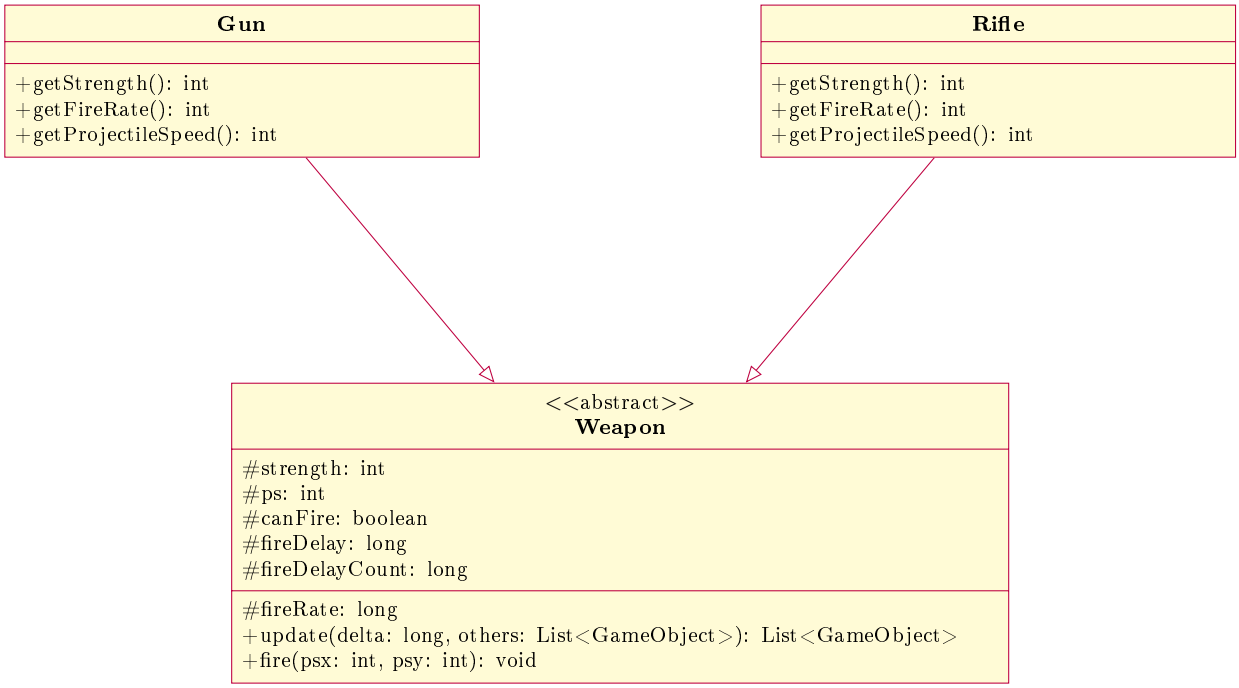


In figura è rappresentato lo schema del Template Method del GameObject

Sempre con in mente il Template Method ho deciso di rendere astratte le classi Entity e Weapon, essendo entrambe una generalizzazione di quelli che sono rispettivamente i diversi personaggi (come il Player e i nemici) e le diverse armi presenti nel gioco. In particolare Entity è una classe astratta che rappresenta l'insieme degli oggetti definiti Living ("vivi"), cioè che si muovono, hanno una salute e compiono azioni tramite input o in modo autonomo (come muoversi, sparare o lasciare oggetti). Nel gioco sono attualmente implementate solo alcune entità basilari, ovvero il Player e tre nemici: Bat, Slime e Boss. Il metodo template però permette facilmente di aggiungere in futuro altre entità (p.e.: nuovi nemici, NPC, ecc.). Allo stesso tempo anche la classe astratta Weapon definisce l'insieme delle armi utilizzabili. Come Entity ci sono per ora solamente due tipi di Weapon chiamate Gun e Rifle ma è possibile aggiungerne altre facilmente.



In figura è rappresentato lo schema del Template Method della Entity

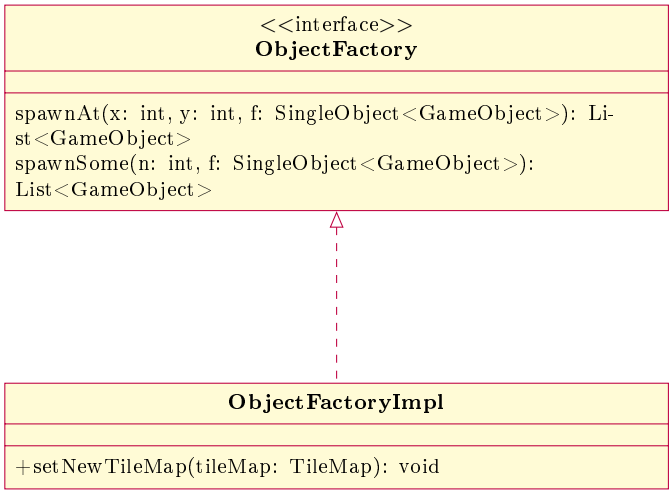


In figura è rappresentato lo schema del Template Method della Weapon

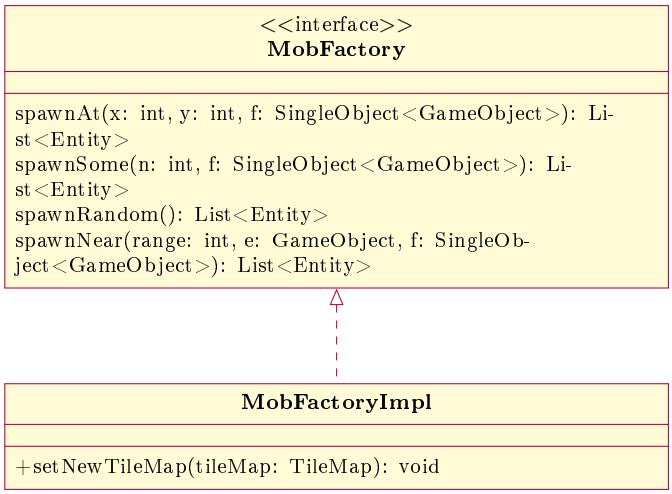
Per la generazione degli oggetti di gioco ho deciso di distinguere la generazione delle entità da quella degli oggetti. La decisione di implementare due factory invece che una deriva da due fattori:

- Chiarezza dei nomi delle factory in relazione agli elementi creati
- Crezione di sole Entity per facilitare l'utilizzo

Cosi facendo la MobFactory gestisce la creazione di tutte le Entity sulla mappa, mentre la ObjectFactory gestisce i GameObject. Le factory forniscono metodi semplici per lo spawn, che avviene solamente solamente se la posizione selezionata (o generata randomicamente) è accessibile nella mappa.



In figura è rappresentato lo schema del Factory Method della ObjectFactory

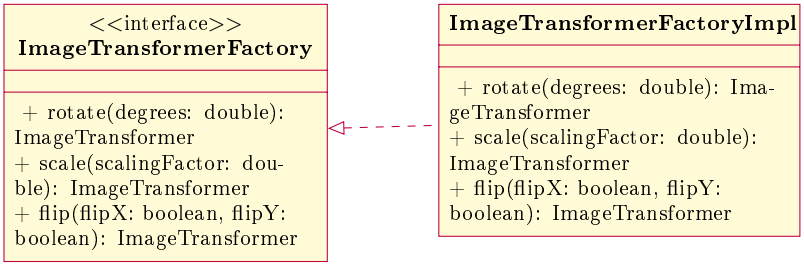


In figura è rappresentato lo schema del Factory Method della MobFactory

2.2.4 Tampieri

Durante lo sviluppo ho notato la necessità di utilizzare due diversi design pattern.

`ImageTransformer` Abstract Factory



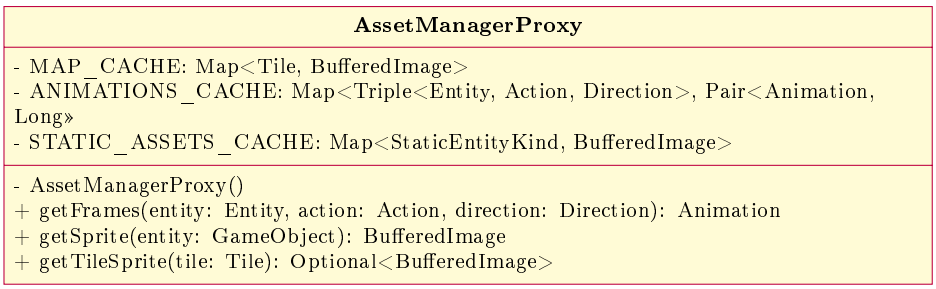
Questa Abstract Factory si è rivelata necessaria perché venivano effettuate diverse operazioni di trasformazione di immagini in più parti del codice.

Per favorire il riuso e ridurre la complessità ho deciso di creare un'interfaccia `ImageTransformer`, che si occupa di applicare una certa trasformazione ad una immagine.

Una factory di `ImageTransform` ritorna la trasformazione richiesta, che può essere applicata a più immagini.

Un vantaggio di questo pattern, che per ora non viene sfruttato, è fornire diversi algoritmi di trasformazione nel caso quelli implementati non siano sufficientemente efficienti.

AssetManager Proxy



Conscio del fatto che non si tratti di un Proxy in senso lato, ho deciso di adattare il pattern al nostro caso d'uso per consentire un migliore utilizzo dell'`AssetManager`, che prende in input stringhe, effettuando una validazione dei parametri di input e trasformandoli in stringhe, riducendo così le fonti d'errore.

Inoltre, tramite questa classe, effettuo il caching delle immagini, cosa che, soprattutto nel caso di immagini cui viene appplicata una trasformazione, riduce il tempo di rendering.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Il gruppo ha realizzato testing automatizzato tramite una suite JUnit 5 di diversi componenti chiave per verificare che aderissero alle specifiche.

In particolare:

- Per le componenti che gestiscono I/O (`AssetManagerTest`, `UIUtilsTest` e `GameSheetTest`) viene verificato che l'accesso a risorse inesistenti venga gestito correttamente
- Per le componenti che gestiscono la logica del gioco (`CharactersTest` e `EquipmentTest`) viene controllato che le operazioni sugli oggetti causino i risultati atteso, come ad esempio la diminuzione del punteggio salute dopo essere stati colpiti da un proiettile
- Per la generazione casuale della mappa (`MapTest`) viene controllato che le mappe generate rispondano a certe caratteristiche e che la loro generazione non generi eccezioni
- Per lo spawn delle entità (`MobFactoryTest`) viene testata la corretta generazione delle stesse
- Per la gestione della logica di visualizzazione e di interazione (`CameraTest`, `CollisionBoxTest` e `HIDTests`) viene controllato che la logica della visualizzazione sa corretta nel primo caso, e che vengano gestiti correttamente gli eventi nei restanti due

Si è inoltre deciso di avere una test coverage minima obbligatoria, fissata al 50%, e di testare tramite una pipeline (utilizzando Github Actions) per evitare di introdurre regressioni nel branch principale.

3.2 Metodologia di lavoro

3.2.1 Chelli

I file che ho sviluppato interamente io o quasi sono:

- Tile
- TileMap
- TileMapImpl
- TileMapFactory
- TileMapFactoryImpl
- Projectile

Le dipendenze della TileMap con il resto del progetto e' minima, e la progettazione iniziale e' stata abbastanza ben costituita in quanto ho dovuto aggiungere funzioni solo una volta su richiesta degli altri membri, e questa modifica e' stata molto semplice e di poche righe. Oltre ha questo ho contribuito in diverse classi per aggiungere feature o fare refactor di parti problematiche. Abbiamo tutti usato git creando nuovi branch per ogni cambiamento richiesto, facendo uso di task automtizzati su github per il controllo di errori e stile del codice, e richiedendo review ai collaboratori per ogni pull request al master.

3.2.2 Sanità

In questa parte esporrò il mio contributo per il progetto relativo alle interfacce grafiche al caricamento di risorse grafiche e testuali , alla gestione degli asset e delle animazioni del gioco , alla struttura del loop principale attraverso il quale si sviluppa il gioco e alla interazione con dispositivi esterni quali mouse e tastiera.

La parte di cui mi sono maggiormente e occupato la GUI del gioco nello specifico la gestione degli stati attraverso i quali il gioco si sviluppa partendo dalla creazione di un generico stato `State` dal quale si estendono poi altri stati quali: `MenuState` , `GameState` e `EndGameState`.

Un'altra parte di cui mi sono occupato riguarda le transizioni ovvero il cambiamento da uno stato a l'altro attraverso la creazione , per mezzo di `TransitionFactory` , di un interfaccia che fa da ponte tra due stati. La gestione di degli stati e dei loro cambiamenti è affidata ad un'altra classe DungeonGame anche questa implementata da me. Questa classe si occupa

di controllare in quale stato si trova il gioco e in quale stato dovrà trovarsi se il player muore o se finisce il gioco, in poche parole la classe che gestisce la visualizzazione il funzionamento e il susseguirsi degli stati di gioco. Inoltre fornisce informazioni riguardanti la finestra principale del gioco creata dalla classe `MainWindow` di mia implementazione anche se la totalità delle informazioni riguardanti la finestra è disponibile a chi ne facesse uso nella mia classe `GameConfiguration`.

Il `DungeonGame` estende la classe `Game` che contiene i metodi necessari all'inizializzazione della finestra di gioco, il render degli aspetti di base ma soprattutto si occupa del loop principale del gioco.

Un'altra classe di del mio codice è la gestione degli input di dispositivi esterni quale la tastiera `KeyManager` che gestisce i vari input da tastiera.

Una altra parte di mia responsabilità è il caricamento di risorse come si può vedere dal package `catacombs/ui/utils (FontUtils, ImageLoader e ImageRotator)` e dall'`AssetManager` che fa uso di un'altra classe, `GameSheet` che si occupa di ritagliare le immagini in immagini più piccole in modo da separare i singoli frame di movimento dei personaggi. Tali frames verranno poi utilizzati dalla classe `Animation` che provvede a sequenziarli in modo da farli risultare parte di un unico fluido movimento.

Usufruendo delle funzinalità di branching che Git offre è stato facile rimanere aggiornato sugli sviluppi del progetto anche non relativi alle mie implenatazioni. la metodologia utilizzata cosnisteva nell' utilizzo di un master principale da aggiornale solo con il codice ultimato e più volte controllato da me e dai miei compagni. Ogni parte di codice da me realizzata si sviluppava all'interno di branch in modo da non sporcare il codice definitivo contenuto nel master. solo dopo aver ultimato ogni modifica ed aver ottenuto l'approvazione dei miei colleghi il branch veniva mergiato con il master. La suddivisione dei lavori precedentemente stabilita veniva rispecchiata nella spartizione degli issue ovvero degli obbiettivi parziali da risolvere che uniti andavano a comporre una parte sostanziosa di codice.

3.2.3 Monti

Mi sono interessato per la quasi totalità ai GameObject e alle classi che lo estendono ad esclusione del Player, SimplePotions e Projectiles. In particolare mi sono occupato dell'analisi e della creazione di GameObject, di Entity, di Weapon e del package model.gen contenenti le Factory MobFactory e ObjectFactory. Sviluppo dei nemici Bat, Slime e Boss, della Weapon e classi derivate Gun e Rifle. Refactoring di alcune classi e metodi. Implementazione di piccole feature in classi altrui. Per ogni cambiamento o feature è stato

creato un branch ed è stata richiesta la review per la pull request al master. Inoltre ho aggiunto qualche sprite al gioco.

3.2.4 Tampieri

Ho realizzato le seguenti classi:

- `eu.eutampieri.catacombs.model.Action`
- `eu.eutampieri.catacombs.model.Direction`
- `eu.eutampieri.catacombs.model.HealthModifier`
- `eu.eutampieri.catacombs.model.LivingCharacter`
- `eu.eutampieri.catacombs.model.Player`, con contributi minimi di Monti
- `eu.eutampieri.catacombs.model.SimplePotion`
- `eu.eutampieri.catacombs.ui.gamefx.Animatable`
- `eu.eutampieri.catacombs.ui.gamefx.AssetManagerProxy`
- `eu.eutampieri.catacombs.ui.utils.ImageTransformer`
- `eu.eutampieri.catacombs.ui.ImageTransformerFactory`
- `eu.eutampieri.catacombs.ui.ImageTransformerFactoryImpl`
- `eu.eutampieri.catacombs.ui.World`

Ho inoltre realizzato alcune bugfixes nel codice da me usato, in particolare in `eu.eutampieri.catacombs.ui.Game`, `eu.eutampieri.catacombs.model.GameObject` e `eu.eutampieri.catacombs.model.Entity`.

Tranne che per le modifiche di classi o interfacce comuni, non ho avuto particolari problemi dovuti allo sviluppo in parallelo, anche grazie all'utilizzo di `git`, di cui ho sfruttato il branching e il cherry pick.

3.3 Note di sviluppo

3.3.1 Chelli

- Uso di stream per facilitarmi nell'algoritmo di generazione della mappa

- Uso di lambda expression insieme agli stream
- Abbiamo usato gradle
- L'algoritmo usato per generare la mappa, essendo molto specifico non e' fornito da librerie esterne

3.3.2 Sanità

- Uso di Optional per la gestione degli eventuali file non trovati
- Uso di basilari lambda expressions nei test automatizzati

3.3.3 Monti

- Streams
- Lambda expressions
- Generici nelle factory
- Apache commons per l'utilizzo dei pair

3.3.4 Tampieri

Ho utilizzato le seguenti funzionalità avanzate del linguaggio:

- Generici bounded, utilizzati per estendere il `SingleObject` realizzato da Giacomo Monti
- Uso di lambda expression in `AssetManagerProxy` e `ImageTransformerFactoryImpl`
- Uso di `Stream` in `AssetManagerProxy`
- Utilizzato (seppur marginalmente) Apache Commons Lang: nello specifico `Pair` e `Triple`
- Configurato Gradle, partendo dall'esempio fornitoci e integrandolo con la static code analysis

Ho utilizzato i seguenti snippet di codice:

- How to scale a `BufferedImage`[1], `src/main/java/eu/eutampieri/catacombs/ui/utils/ImageTransformerFactoryImpl.j` linee 23-29.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Chelli

Il mio lavoro si concentrava su una parte chiave nel progetto anche se non centrale,
per questo mi e' stato facile lavorare sul mio progetto in modo indipendente, riportando un risultato piu che soddisfacente senza particolari problemi.
Anche se il progetto potrebbe essere portato avanti, non e' nelle mie intenzioni farlo,
visto che come gioco non porta avanti niente di nuovo o interessante.

4.1.2 Sanità

Questo progetto è stato veramente impegnativo ed è stato in grado di far risaltare le mie lacune riguardo sopratutto l'utilizzo di `git` che come primo approccio è stato abbastanza brusco, mi sono ritrovato catapultato in una gestione condivisa dei file che non avevo mai usato.

Per quanto riguarda l'implementazione dei metodi e delle interfacce la difficoltà non è stata esagerata in quanto è bastato trovare una soluzione architetturale decente prima di scrivere righe di codice.

4.1.3 Monti

Le difficoltà principali le ho riscontrate durante l'analisi del modello, essendo la prima esperienza di game design. Le debolezze del gioco sono la poca varietà di armi e nemici e la grafica non troppo moderna, ma grazie al lavoro

per ora svolto aggiungere nemici e armi risulterà molto semplice se mai si vorrà.

4.1.4 Tampieri

Personalmente ritengo che la parte di progettazione delle mie classi sia nel complesso positiva, perché ho strutturato il codice in modo abbastanza flessibile, soprattutto grazie all'uso di interfacce per astrarre i comportamenti dalle implementazioni.

Inoltre, il codice è strutturato in modo da rendere abbastanza semplice creare un gioco multiplayer, cosa che potrebbe essere realizzata in una versione successiva.

Ciononostante, ci sono alcune parti ampiamente migliorabili, come il collegamento degli asset realizzato nell'`AssetManagerProxy`, che renderà complessa l'aggiunta di nuove entità in futuro.

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Tampieri

Durante lo svolgimento del progetto mi è stato abbastanza difficile coordinarmi con il resto del gruppo per realizzare le parti di mia responsabilità.

Ritengo che questo sia dovuto al non aver raggiunto un livello di dettaglio in fase di progettazione a me sufficiente per capire effettivamente la struttura del progetto.

Il deficit sottolineato sopra ha portato ad alcune discussioni sul design delle classi, soprattutto su come mantenerle retrocompatibili ma adeguandole ad esigenze non emerse in fase di progettazione.

Appendice A

Guida utente

Il gioco comincerà con una schermata di menu che permette di cambiare l'opzione che si desidera scegliere attraverso l'uso dei tasti direzionali o WASD, per selezionare la chiusura della finestra o l'avvio del gioco.

Una volta che l'indicatore di selezione si sarà spostato sull'opzione desiderata è sufficiente premere il tasto invio per confermarla.

Iniziata la partita, il personaggio si potrà spostare la mappa utilizzando i tasti indicati in precedenza.

Il giocatore potrà sparare ai nemici per eliminarli usando il tasto spazio, sia tenendolo premuto che premendolo ripetutamente, e per poter mirare al nemico basterà rivolgere il corpo del giocatore nella direzione desiderata (e.g se un nemico si trova alla proria destra, l'utente dovrà premere D e poi premere la barra dello spazio).

Nella mappa sono presenti alcuni oggetti utili, che possono essere raccolti camminandoci sopra.

Nello specifico, sono presenti delle pozioni (rappresentate da un'ampolla) per aumentare la propria vita e delle armi. Queste ultime possono essere:

- Una pistola, che è l'arma con cui il giocatore inizia il gioco.
- Un fucile automatico che, rispetto alla pistola, permette di sparare più frequentemente.

Dopo aver ucciso un nemico, è possibile che questo lasci cadere un'arma o una pozione.

Il gioco termina dopo aver ucciso il boss e tutti i nemici.

Appendice B

Esercitazioni di laboratorio

B.1 Tampieri

- Laboratorio 5: https://virtuale.unibo.it/mod/forum/discuss.php?d=62684#p101017
- Laboratorio 6: https://virtuale.unibo.it/mod/forum/discuss.php?d=62579#p100872
- Laboratorio 7: https://virtuale.unibo.it/mod/forum/discuss.php?d=62582#p100877
- Laboratorio 8: https://virtuale.unibo.it/mod/forum/discuss.php?d=63632

Bibliografia

[1] trashgod. *How to scale a BufferedImage*. https://stackoverflow.com/questions/4216123/how-to-scale-a-bufferedimage\#4216635. 2018.

[2] Wikipedia. *Entity component system — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Entity\%20component\%20system&oldid=1017633559. [Online; consultato il 24 Aprile 2021]. 2021.

[3] Wikipedia. *Roguelike — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Roguelike&oldid=1009091012. [Online; consultato il 02 Marzo 2021]. 2021.