Relazione Progetto "Il sacrificio di Isacco"

Lucchi Asia - Pantieri Andrea - Faedi Michele - Casadei Cristian

21 agosto 2019

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il software proposto prende ispirazione da "The Binding of Isaac", un videogioco in 2D in cui il giocatore dovrà esplorare varie stanze combattendo contro i nemici e raccogliendo oggetti di vario genere con l'obiettivo di uccidere il boss per accedere al livello successivo.

Requisiti funzionali

- La schermata iniziale conterrà il menù principale dal quale si potrà iniziare a giocare o modificare alcune impostazioni
- La stanza sarà visualizzata dall'alto, il giocatore si sposterà all'interno di una stanza sia orizzontalmente che verticalmente e cambierà stanza nel momento di collisione con una porta
- Sulla schermata saranno presenti i cuori che indicano la vita del giocatore e l'inventario degli oggetti che possiede
- In ogni stanza potranno essere presenti dei nemici che si muoveranno autonomamente grazie ad un AI e feriranno il personaggio in vari modi:
  - lanciando delle lacrime contro il player
  - scontrandosi con il player
- l'utente gestirà il player, che potrà muoversi liberamente fra le stanze di un livello, sparare lacrime ai nemici per ucciderli e raccogliere oggetti
- In ogni stanza potranno essere presenti degli oggetti con i quali il giocatore può interagire transitandoci sopra

  - Cuori: possono essere di diverso colore (che indica un diverso comportamento) ma genericamente aumentano la vita del giocatore
  - Chiavi: vengono aggiunte all'inventario del giocatore e permettono di aprire porte che conducono alle altre stanze
  - Bombe: vengono aggiunte all'inventario del giocatore che le può rilasciare premendo Q

Requisiti non funzionali

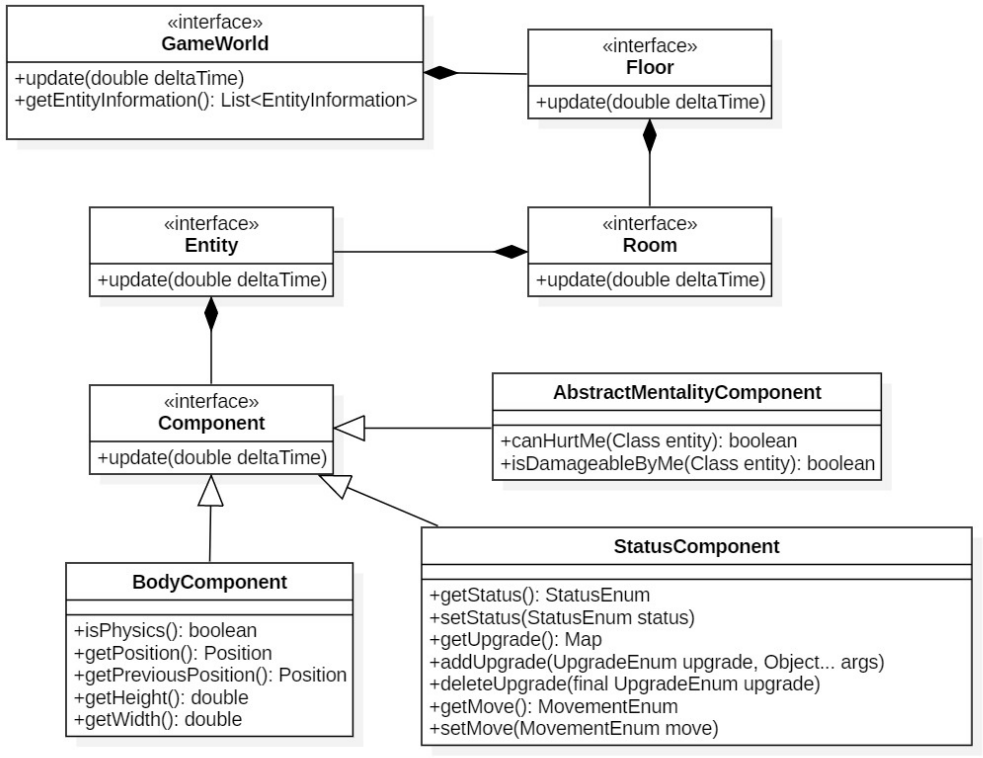
- Il codice dovrà essere sufficientemente efficiente da permettere di giocare con una grafica fluida anche con grafiche intel UHD 600 o superiori.

1.2 Analisi e modello del dominio

La challenge principale del gioco è la realizzazione di un mondo composto da varie stanze al cui interno sono presenti molteplici entità che interagiscono fra loro. Le entità hanno un corpo, si possono muovere tra le stanze e reagiscono in una specifica maniera al momento della collisione. Sono principalmente caratterizzate dai componenti che indicano cioè ogni l'entità può fare:

- Corpo: indica la posizione dell'entità nella stanza, la sua dimensione ed il suo peso.
- Salute: gestirà la vita dell'entità, che potrà essere danneggiata o ripristinata. Ad esempio, la vita di Isaac sarà decrementata quando entrerà in contatto con un nemico ed incrementata quando raccoglierà un cuore.
- Movimento: gestirà i movimenti e la velocità dell'entità nelle varie direzioni.
- Inventario: conterrà gli oggetti posseduti dall'entità e si occuperà di gestire il modo in cui vengono raccolti e rilasciati.
- Collisione: gestirà le conseguenze del contatto della nostra entità protagonista con un'altra. Ad esempio, se la lacrima tocca il nemico esso dovrà perdere vita e la lacrima dovrà sparire, mentre se la lacrima tocca una roccia o Isaac stesso essa dovrà solamente scomparire.
- Danno: indicherà quanto danno potrà fare l'entità alle altre entità.

- AI: una sorta di cervello che indica per ogni entità in che modo deve agire. Ogni nemico ne avrà una diversa, che ad esempio lo farò muovere in maniera casuale o tentando di inseguire Isaac. In generale ci saranno dei componenti di questo tipo che apparterranno a specifiche entità (ad esempio le porte o le lacrime) e ne definiranno il comportamento attraverso l'utilizzo di altri componenti (specialmente il movimento).
- Mentalità: indica la personalità dell'entità ovvero a quali entità farà danno e a quali no.



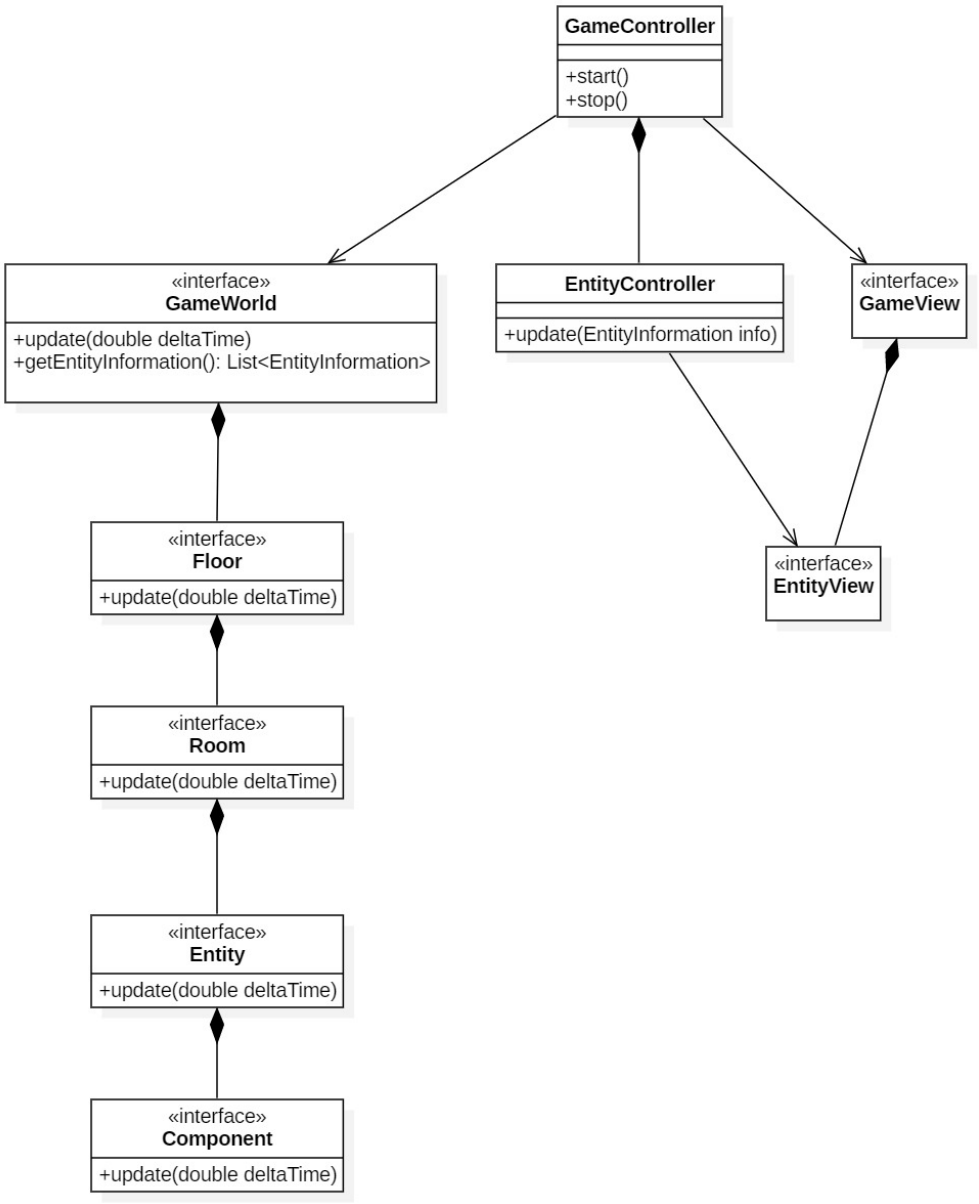
Schema del modello Entity - Components.

Capitolo 2

Design

2.1 Architettura

Il software sfrutta l'architettura MVC. La parte model è stata implementata utilizzando il pattern Entity-Component e il Publish/Subscribe (con l'ausilio degli Event Bus della libreria Guava) per la gestione delle collisioni. Si è fatto un'ampio uso di Reflection per gestire le composizioni di componenti che vanno a formare le varie entità.



Schema UML architetturale principale del ISDI.

L'entry point del model è il GameWorld, che si compone del player e dei piani, i quali loro volta sono composti da stanze che contengono le entità (fra le quali anche il player stesso). Si è scelto di considerare ogni entità come insieme di componenti, perciò nelle entità non ci sono metodi comportamentali ma solamente metodi relativi all'aggiunta o rimozione di componenti.

2.2 Design dettagliato

2.2.1 Cristian Casadei

Riguardo al model, mi sono occupato dello sviluppo dei componenti e nello specifico del package collision, delle mentality e le classi AbstractCollectableComponent, AbstractPickupableComponent e FactoryPlayersUtil . Il package collision è stato svilluppato in maniera gerarchica, ogni figlio aggiunge delle funzionalità al padre.

- CollisionComponent : è il padre di tutti i componenti che gestiscono le collisioni; al verificarsi di una collisione decide se inviare nell'event bus dell'entity un evento danno oppure no.

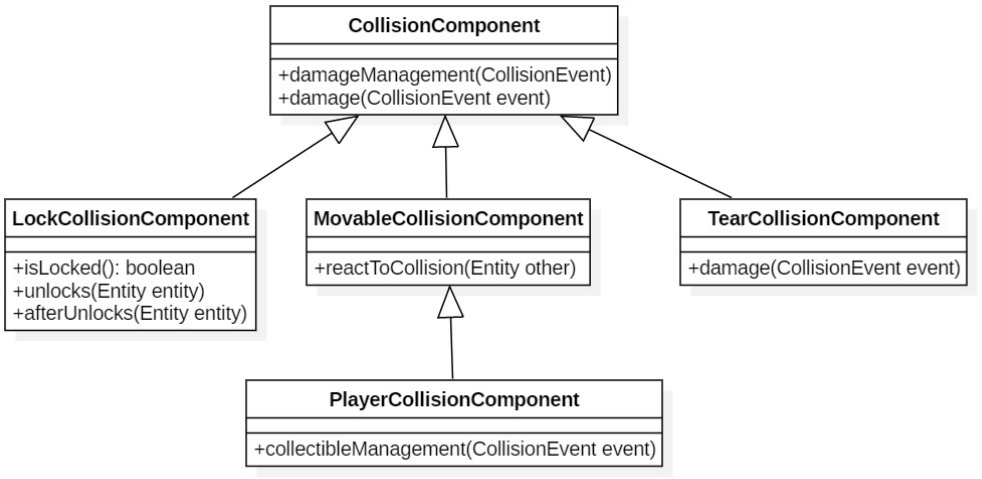
- MovableCollisionComponent : estende direttamente il CollisionComponent, aggiunge il riposizionamento di un'entity quando questa collide con un'altra entity. E' la gestione della fisica del gioco: questo componente è stato implementanto da Michele Faedi e Andrea Pantieri ma per dare una visione di insieme di tutto il package collision ho voluto citarlo anche qui.

- LockCollisionComponent : estende direttamente il CollisionComponent, è una classe astratta che aggiunge la modellizazione degli oggetti chiusi verificando da prima se la collisione è stata con il player. Successivamente si "apre" se il player ha nel inventario una chiave.

- PlayerCollisionComponent : estende il MovableCollisionComponent, aggiunge a questo la funzione di raccolta degli oggetti raccoglibili. Il meccanismo di come gli oggetti vengano raccolti verrà spiegato nel dettaglio successivamente.

- TearCollisionComponent : nonostante le lacrime si muovano, questo componente estende direttamente il CollisionComponent invece che il MovableCollisionComponent, in quanto la lacrima non si deve riposizionare una volta subita la collisione ma deve semplicemente scomparire.

- Note : alcune entity come lacrime e oggetti raccoglibili necessitano della proprietà di non collidere fisicamente: una qualunque entity che collida con questi non deve riposizionarsi ma deve potergli passare attraverso. Per la risoluzione di tale problema è stato utilizzato una variabile boolean nel BodyComponent che determina la "fisicità" o meno dell'entità stessa, questa variabile poi è letta dal MovableCollisionComponent.



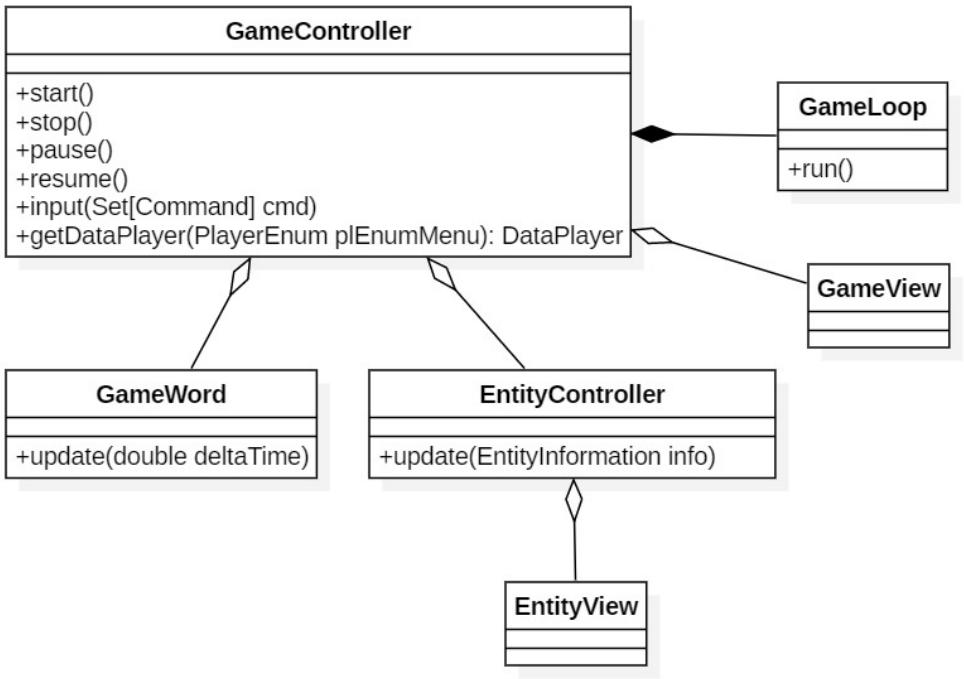
Il package delle mentality è stato progettato da me e Andrea Pantieri per risolvere il seguente problema: alcune entity nemiche non ricevono danno fino a quando altri tipi di entity nemiche sono presenti nella stanza. Perciò abbiamo ideato un componente che fosse selettivo delle varie Entity.Class: serve ad individuare con precisione da chi una certa entità può ricevere danno e a chi può fare danno. A prima vista queste due informazioni possono risultare ridondanti ma sono state molto utili ad esempio nella gestione della entity Tear: infatti questa è un' unica classe ma può essere sia nemica che amica del player. Per gestire il comportamento diverso, le varie lacrime ereditano la mentality di chi le lancia, perciò prendendo come esempio la lacrima lanciata dal player si ha che il palyer può far danno alla lacrima ma la lacrima non può ricevere danno dal player e quindi è come se non collidesse con esso.

AbstractPickupableComponent questo componente è il padre di tutti i componenti assegnati agli oggetti(Entity) raccoglibili e che ne modellizza la raccolta. Esso non è iscritto all'evento della collisione in quanto sarà il player che quando avviene la collisione con un oggetto raccoglibile, posterà nell'event bus del player un PickUpEvent che chiamerà il metodo init dell'oggetto. Alcuni oggetti hanno bisogno di modellizzare non solo la raccolta da parte del player ma anche il fatto di essere conservati all'interno dell'invetory component come ad esempio le key e le bomb. A queste entità verrà assegnato un componente che estende AbstractCollectableComponent che a sua volta estende AbstractPickupableComponent.

Esistono diversi player con cui si può iniziare il gioco. Dato che i player differiscono l'uno dall'altro solamente per quattro caratteristiche: velocità, vita, danno e velocità di fuoco; ho fatto una static factory per la generazione

dei vari player, la FactoryPlayersUtil. Questa classe oltre che a generare un player prendendo in ingresso una variabile di tipo PlayerEnum, ha anche un'altra funzione, quella di passare al game controller i dati di ciascun player senza generare nessun player. Questo è utile perché nel menù di selezione del personaggio, la view deve graficare le caratteristiche dei player, ma solo uno di questi alla fine verrà selezionato per far avviare il gioco.

Riguardo al controller, mi sono occupato della progettazione e implementazione delle classi EntityController e GameController, dove si è fatto largo uso della reflection. Il GameController è la classe del controller che si occupa di comunicare con il model, di fa partire il game loop e ad ogni update carica le informazione riguardanti le entity dal model e le invia alle EntityController. Queste ultime a loro volta, tramite una traduzione da enumeration a metodo caricata da file XML, chiamano dei metodi nella view per graficare le entity nel rispettivi stati di animazione.



2.2.2 Michele Faedi

Io mi sono occupato di gestire la struttura del controller, di gestire la parte visuale dei SubMenu e dei macro menu, gestire il Floor, la Room e il GameWorld.

Controller

Ho gestito il tutto con una struttura ad albero in cui ogni nodo contiene o un altro albero o una foglia chiamata child. Ogni nodo ha selezionato un solo nodo figlio per cui è possibile identificare univocamente partendo dalla radice il child selezionato. Un child può scegliere di selezionare altri nodi e tornare dormiente.

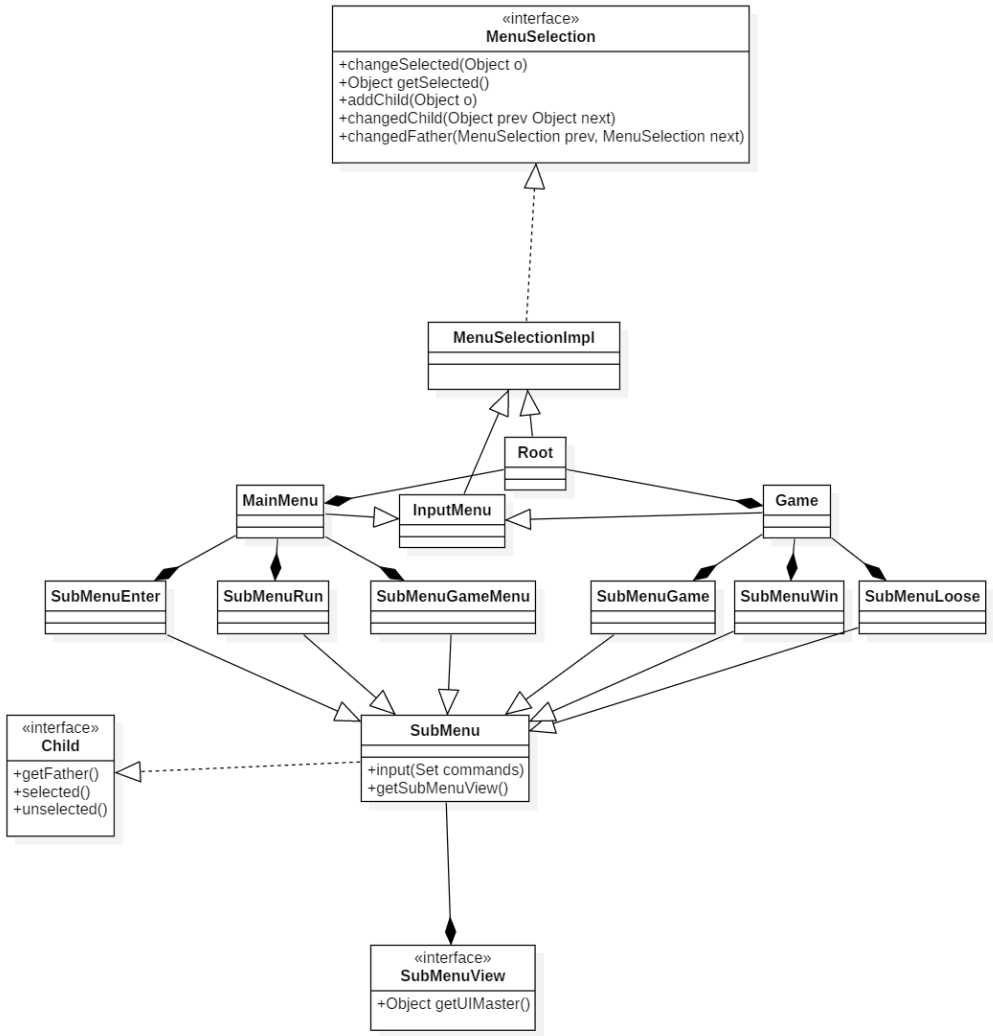


Figura 2.1: Struttura controller

Per gestire i menu del gioco ho utilizzato questa struttura ad albero (figura 2.1) usando come radice un albero che contiene nodi che ricevono input. E

come nodi della radice ho utilizzato 3 macro menu in particolare : Intro, Menu e Game. Ognuno si occupa di gestire le sezioni del programma nelle diverse fasi: quella di introduzione, quella in cui si sfoglia il menu principale e quella in cui si gioca.
Questi 3 nodi contengono dei SubMenu. I SubMenu corrispondo a dei child e gestiscono i vari sotto menu.
Ad esempio c'è il SubMenu del Menu che gestisce se andare nel sub menu delle opzioni o quello per iniziare una nuova partita.
I 3 macro nodi hanno un oggetto che tramite gestisce il modo di scambiare i SubMenu e ciò è gestito dalla view. Infatti nel Menu le pagine si scambiano visivamente in maniera diversa che dal Game. Anche i SubMenu hanno un oggetto che modula le diverse disposizioni di oggetti di view e ne visualizza il contenuto.

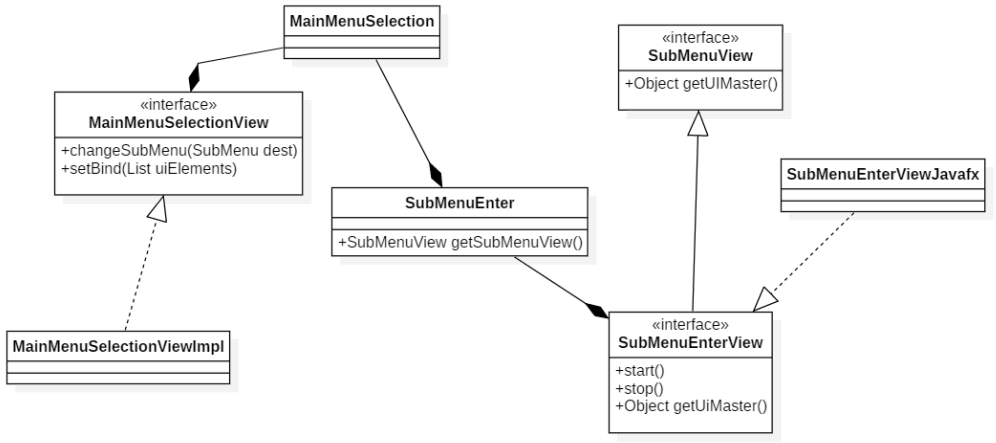


Figura 2.2: Esempio di macro menu e sub menu con relative classi di view

Questo è un esempio di gestione del SubMenu per la prima pagina del Menu. (figura 2.2) Si nota che il macro menu MainMenuSelection ha un oggetto di view che serve per modellare il ridimensionamento di tutti i SubMenu contenuti tramite il metodo "getUiMaster".

View

Per la gestione della libreria grafica ho usato JavaFx perché è un buon strumento open source(anche se ho dovuto gestire il ridimensionamento della finestra per adattarlo a quello del gioco originale). Per gestire il fatto che la

View dei SubMenu è creata dal Controller e non potendo passare i nodi come parametri perché violerebbe l'MVC ho usato un Singleton (ViewGetterUtil) per ottenere tutti i nodi di javafx inseriti nell'fxml.
Per sfogliare le pagine ho usato la classe TranslationPage tramite Strategy mostra il Pane desiderato. Ci sono 2 implementazioni: uno per il Menu e uno per il Game.
Per la pagina per creare la partita ho creato una classe che dispone dei nodi in un cerchio e tramite next e previous ruota i nodi in modo da avere in basso quello desiderato. Ci sono 2 implementazioni: uno normale e uno che randomizza e restituisce un elemento tra quelli presenti.
Per la SubMenu di ingresso(quello con la animazione) mi sono servito di 2 classi: una che cambia la source di una ImageView avendo diverse Image e una che cambia la precedente con un timer.

Model

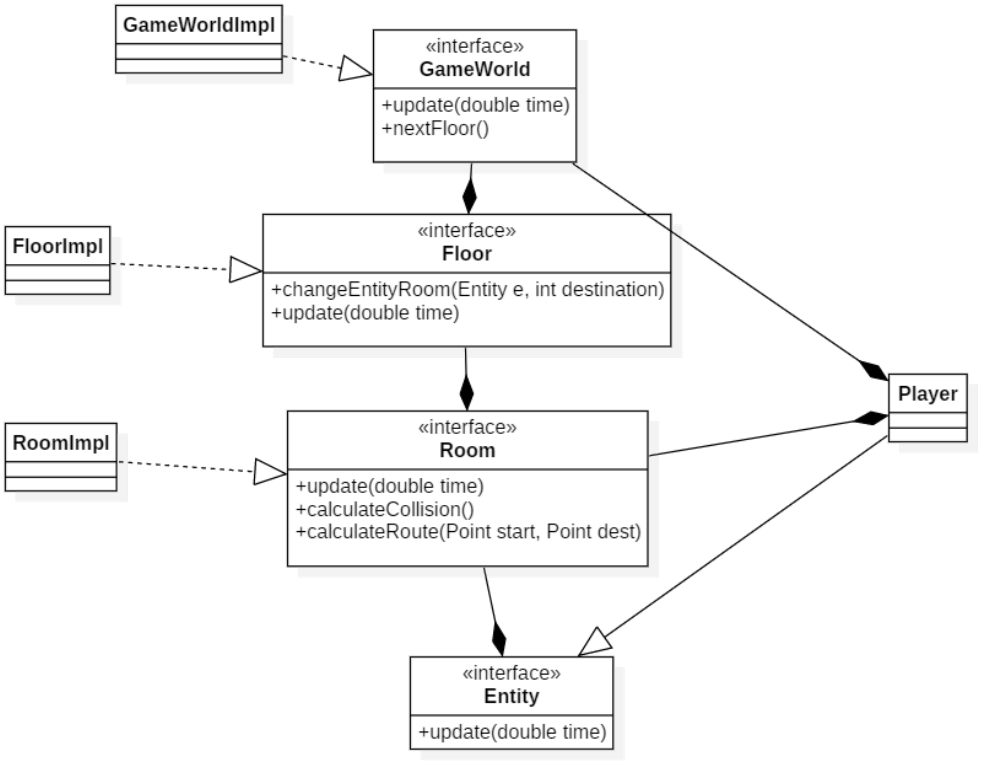


Figura 2.3:

Per il modello mi sono occupato di gestire il GameWorld, i Floor e le Rooms. In particolare della creazione randomica del piano. Il mondo di gioco contiene diversi piani ognuno dei quali contiene delle stanze e ogni stanza contiene delle entità. Per facilitare la creazione del piano ho preferito usare una creazione randomica del layout delle stanze in questo modo è più simile al gioco originale e più interessante da giocare. Ogni stanza contiene uno spazio che gestisce le collisioni e gli algoritmi di ricerca del percorso minimo. Da notare che aggiornare il GameWorld comprende anche aggiornare il piano selezionato che a sua volta aggiorna la stanza in cui c'è il giocatore che a sua volta aggiorna tutte le entità presenti. Sia che per il GameWorld, che per il Floor che per le Rooms è stato usato il pattern Strategy per disaccoppiare il più possibile la gestione del gioco per possibili modifiche future. Nel GameWorld e nel Floor è stato usato il pattern Observer che tramite l'event bus di guava permette di registrarsi a degli eventi importanti quali ad esempio cambio di stanza (Floor è Observable e altri nel model sono Observer), cambio di piano o vittoria del giocatore (GameWorld è Observable e altri model e controller sono Observer). Il player è contenuto sia nel GameWorld e sia nella Room ma la cambia ogni volta che entra in una porta.

2.2.3 Asia Lucchi

Model

Per quanto riguarda il model, mi sono occupata dell'implementazione dei componenti di base.

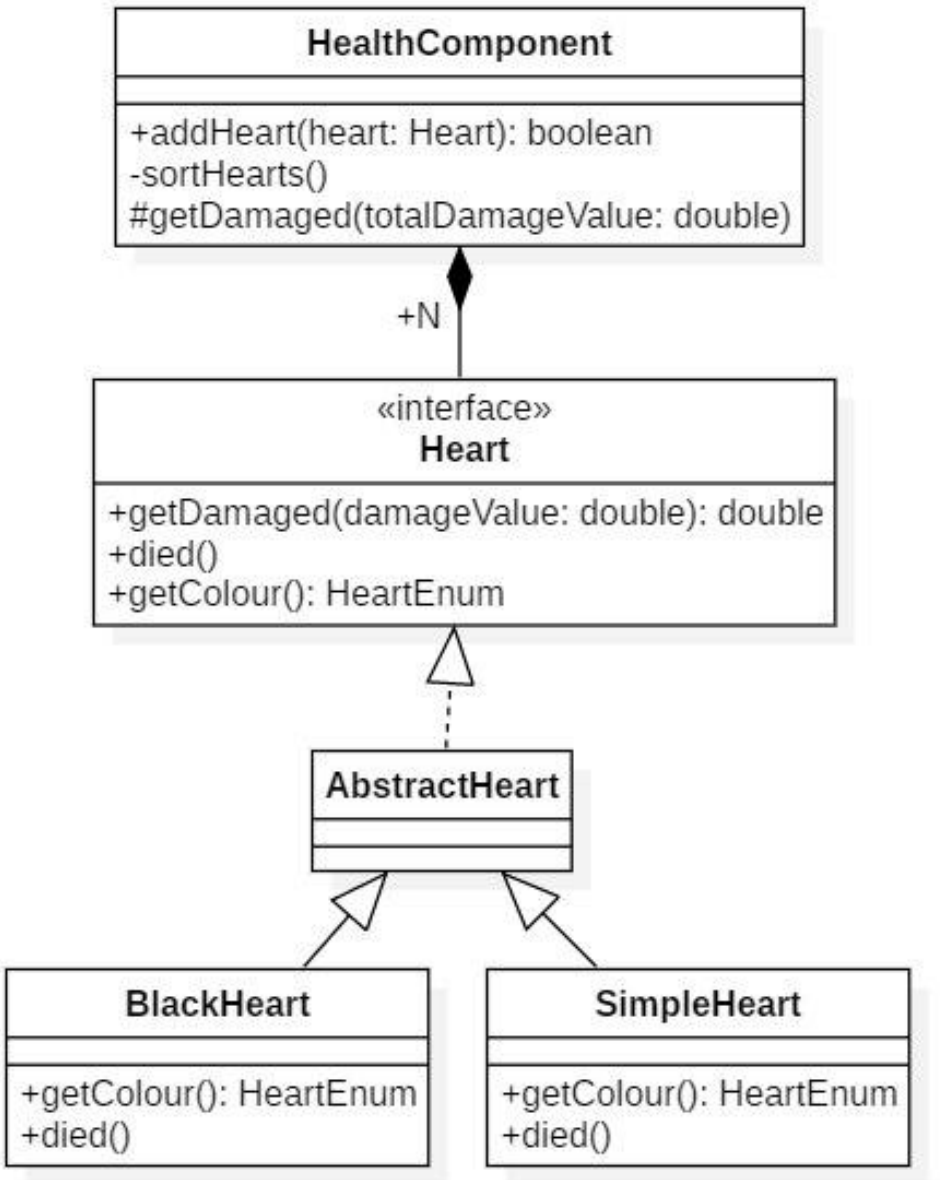
Ho fatto ampio uso del pattern Observer in buona parte dei construttori per registrare listeners degli eventi a cui il componente necessita di reagire quando gli eventi saranno postati sul Guava Bus dell'entità a cui l'istanza del componente appartiene.

Ho fatto utilizzo di Reflection, ad esempio nell'evento UseThingEvent e nell'InventoryComponent, che implementa un listener dell'evento in questione. L'utilizzo si deve riferire ad un'entità, ovvero una qualsiasi istanza della classe che rappresenta l'entità ed è presente nell'inventario. Utilizzare la specifica istanza invece della classe risulterebbe meno pratico al momento della creazione di un oggetto di tipo UseThingEvent. Di conseguenza all'uso della Reflection ho utilizzato una Wildcard per specificare che la classe passata come parametro all'evento dovesse essere un'entità.

Ho utilizzando l'util Position, integrandovi i metodi di aggiunta e scalo della dimensione, per realizzare i componenti che rappresentavano il corpo ed il movimento dell'entità.

Ho cercato di rendere i componenti di base più riutilizzabili possibile: ad esempio l'HealthComponent dà la possibilità di avere valori di salute diversi

da quelli interi o con valore decimale 5, sebbene nel gioco siano usati solo questi.
Inoltre ho utilizzato il pattern template method per gestire le due tipologie di cuore che si differenziano nel comportamento al momento della morte, infatti il cuore nero deve causare danno ai nemici al momento della scomparsa mentre il cuore semplice no.

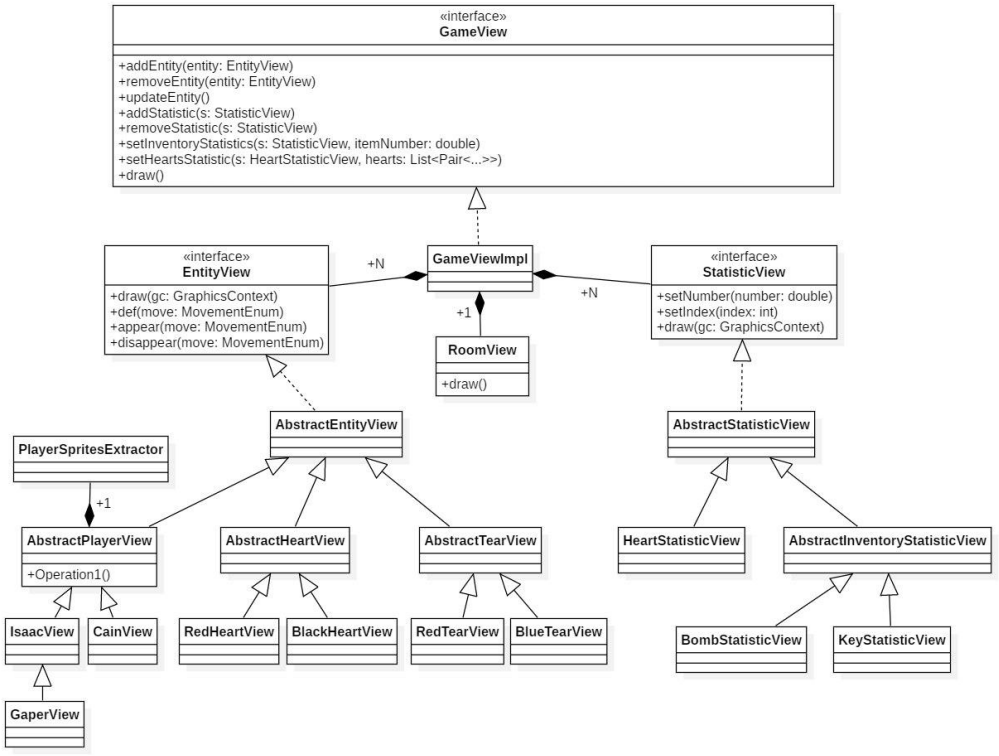


View

Per quanto riguarda la view, mi sono occupata della rappresentazione grafica di tutte le entità, utilizzando degli stati e degli indici per gestire le animazioni e creando le util SpritesExtractor e PlayerSpritesExtractor per reperire le immagini dagli sprite sheets.

Ho utilizzato una classe astratta per le entità (AbstractEntityView) ed una per le statistiche (StatistcEntityView) con la differenza che nel primo caso è sempre necessario specificare la posizione e la dimensione dell'entità per poterla disegnare mentre nel secondo è sufficiente indicare il numero di items di ogni statistica perchè saranno le classi dedicate a gestirne la rappresentazione grafica.

Ho utilizzato vari livelli di ereditarietà per minimizzare la ripetizione di codice, ad esempio nel caso dei player, dei cuori o degli oggetti presenti nell'inventario.



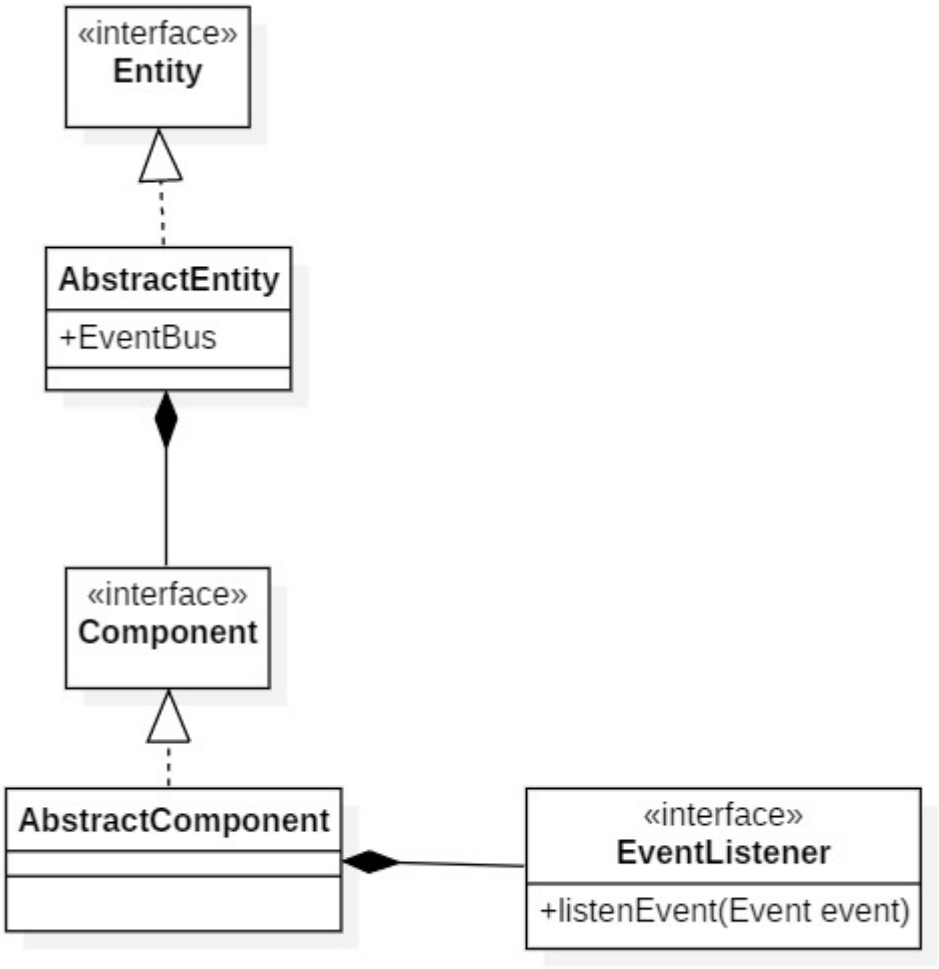
Gli sprite che compongono le animazioni sono contenuti in delle liste statiche riempite attraverso un blocco statico delle classi, in questo modo non è necessario ripetere l'operazione di estrazione che sarà eseguita una sola volta al momento del lancio dell'eseguibile. Al momento dell'istanziazione di un oggetto della classe in certi casi si specifica quali liste utilizzare (ad esempio

nel caso del cuore o della lacrima che possono avere diversi colori). Infine per ogni entità si settano posizione, dimensione e stato e si disegnano con il metodo draw.

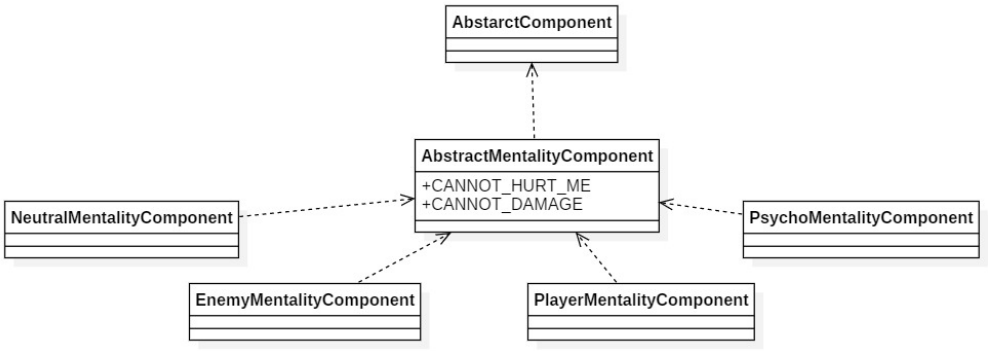
Non mi sono occupata della creazione del Controller ma ne ho dovuto interiorizzare il funzionamento per utilizzarlo nelle comunicazioni fra le classi di Model e di View che ho realizzato, rendendo i metodi coerenti con le richieste di input e i valori di output del controller, in particolare per quanto riguarda le classi GameController e EntityController.

2.2.4 Andrea Pantieri

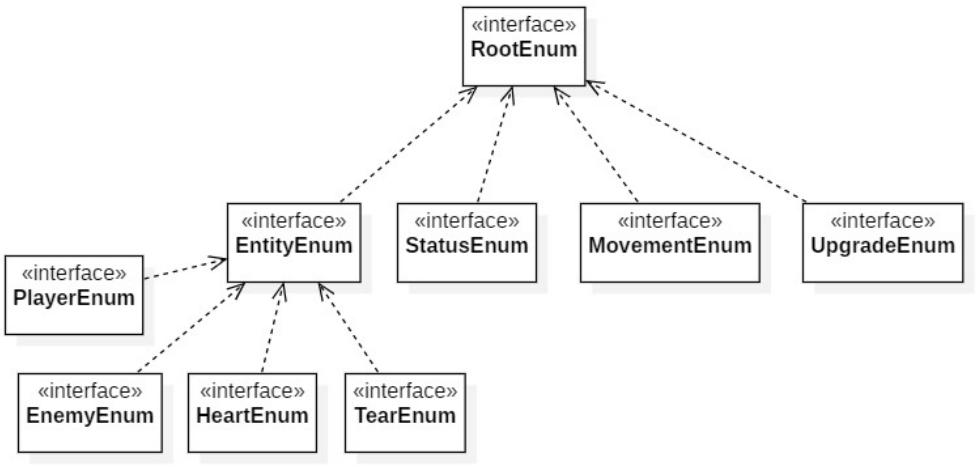
Per la mia parte di modello, ho sviluppato il pattern Publish/Subscribe per far comunicare i vari componenti e le varie entità fra di loro usufruendo dell'EventBus di Guava. In dettaglio, ho creato una gerarchia di eventi, ovvero i diversi messaggi che dovranno essere gestiti dai subscriber implementati da un event listener generico.



Sempre nel modello, ho anche contribuito allo sviluppo di alcuni componenti principali come quello della mentalità e quello delle collisioni per renderli più estendibili. Dato che ogni entità potrebbe avere una mentalità diversa oppure un modo diverso di gestire una collisione, abbiamo progettato un'ulteriore struttura gerarchica ad albero per questi due componenti. Quindi, per esempio, se in futuro si vorrà gestire la collisione in maniera diversa basterà estendere la classe principale del collision component.
Per il mentality component, esistono due set di entity che rappresentano le entità che non possono fare danno al possessore di tale mentalità e a chi il possessore non può danneggiare. Tramite dei metodi è possibile aggiungere o togliere entità a tali set in modo da personalizzare le 4 mentalità di base o anche di crearne una totalmente nuova.



Inoltre, ho sviluppato interamente il GameWorld con la possibilità di creare tutto il gioco tramite file XML per rendere più facile l'aggiunta di nuovi livelli con nuove stanze.
Per finire col modello, ho gestito ed implementato lo scheletro dei componenti, delle varie entità e della struttura ad albero per le enumerazioni utili alla sincronia tra model e view passando dal controller facendo un abbondante uso di Reflection per la gestione dei componenti nell'entity e per la creazione di particolare util statiche tra le quali un Equals e un HashCode generico.



Per la vista, ho sviluppato la gestione del canvas nella classe GameView e ho dato una mano nello sviluppo della util SpritesExtractor implementando inizialmente l'estrazione degli sprites del player. In seguito, mi sono occupato di come far scegliere alle varie viste delle entità quale animazione disegnare a schermo. Per implementare ciò in maniera estendibile, ho aggiunto alle entity nel model un componente per lo stato contenente tre tipologie di enum:

- StatusEnum, l'enumerazione principale che descrive lo stato dell'entità se viva, morta o danneggiata

- MovementEnum, descrive il movimento dell'entità
- UpgradeEnum, necessaria per stati/potenziamenti extra, tornerà utile se dovrà essere necessario fare ulteriori modifiche successivamente alla vista

In base a questi valori, verranno chiamati dei metodi implementati all'interno delle entity view per selezionare quale animazione/sprite mostrare a schermo. Per rendere funzionante ed estendibile questo collegamento, ho mappato via XML i vari valori delle enum con i vari metodi. Questa mappatura rimane poi salvata nel controller in modo tale da automatizzare tutta la procedura. Parlando sempre di XML, ne ho creato anche uno per collegare in maniera univoca le varie entity con le varie entity view usando come chiave i valori dell'EntityEnum salvati nel modello di ciascuna entità.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per testing delle varie parti del codice ci siamo affidati alla liberia JUnit 5 di cui si è fatto utilizzo anche per alcuni test provvisori, in quanto per l'uso della reflections e per il parsing dei vari XML prima di arrivare ad una implementazione di metodi concreti, si è passati da una fase di testing. Un altro strumento utilizzato in fase di testing, è stato lo stesso strumento di debug fornito con l'IDE Eclipse.

3.2 Metodologia di lavoro

3.2.1 Cristian Casadei

Classi sviluppate autonomamente: StatusComponent, AbstractCollectableComponent, AbstractPickupableComponent, BombCollectableComponent, CollisionComponent, LockCollisionComponent, PlayerCollisionComponent, TearCollisionComponent, Player, SimplePsychopathEntity, FactoryPlayersUtils, EntityInformation, Position

Classi sviluppate in collaborazione: EntityController, GameController, InventoryComponent, DoorComponent, AbstractMentalityComponent, EnemyMentalityComponent, NeutralMentalityComponent, PlayerMentalityComponent, PsychoMentalityComponent, Bomb, Tear, le varie enum, PickUpEvent, UseThingEvent, TearWeaponComponent, DataPlayer, StatisticsInformations, StaticMethodsUtils

3.2.2 Michele Faedi

Menu di gioco sia controller sia view, audio del menu, gestione dei controller, gestore di stanza, piano e mondo, algoritmo di ricerca del percorso minimo, ricerca collisioni, mappatura input, creazione randomica del piano, selezione del layout delle stanze da file, risoluzione collisioni (adattamenti per non passare sopra le rocce). Le interfacce delle stanze, dei piani e del mondo è stata decisa in fase di analisi dal gruppo, io mi sono occupato di implementarla e di aggiungere funzionalità trovate in fase di sviluppo come la ricerca del percorso minimo.
Le parti in comune con gli altri membri sono estremamente ridotte per quanto riguarda il menu di gioco. Infatti è stato sviluppato prima di avere il GameController. Invece le parti in comune sono state in gran parte delle aggiunte non richieste che mirano ad abbellire il gioco come ad esempio la AI di ricerca o la accelerazione.
Il DVCS è stato usato in maniera coerente a quanto descritto nelle slide di laboratorio. Abbiamo deciso di usare un ibrido tra quello semplice e quello complesso: nello specifico abbiamo si creato tutti i branch consigliati ma sono stati creati pochi branch per delle feature aggiuntive(solo 3 macro branch per il model, controller, view e qualchedun altro). Come richiesto dai docenti sono stati tracciati anche i file di risorse binarie come le immagini e i video anche contro la mia riluttanza nel farlo. Io sono l'amministratore del progetto e solo io possiedo i permessi di modificare il develop, il master e il branch di release. Gli altri possono creare nuovi branch e fare tutti i merge tra quelli non citati prima. Come gruppo ci siamo imposti di utilizzare per le descrizioni dei commit l'inglese anche se taluni membri non hanno molta familiarità con esso.

Classi sviluppate autonomamente:

Controller

Launcher, CharacterInfo, Child, ConfigurationManager, GameSelection, InputMenu, IntroSelection, MainMenuSelection, MenuSelectionImpl, Root, SubMenu, SubMenuGame, SubMenuEnter, SubMenuGameLoose, SubMenuGameMenu, SubMenuInGameOption, SubMenuIntro, SubMenuOption, SubMenuRun, SubMenuWin.

Model

Cup, GaperEnemy, FollowAiComponent, LifeTimeComponent, WinComponent, WinOnCollisionComponent, InputEvent, RoomChangedEvent, AStar,

Command, Lambdas, Single, Space.

View (escludendo l'implementazione)

SubMenuView, TypeOfAudio, EntryPointView, AnimatedView, Sound, TimedViews, GameIntroView, GameSelectionView, MainMenuSelectionView, SubMenuEnterView, SubMenuGameLooseView, SubMenuGameMenuView, SubMenuGameView, SubMenuInGameOptionView, SubMenuIntroView, SubMenuOptionView, SubMenuRunView, SubMenuWinView, CircleList, RotatingNode, SelectList, TranslationPages, TwoStateNode.

Classi sviluppate in collaborazione:
GameController, MoveComponent, EnemyHealthComponent, FlyAIComponent, DoorComponent, Bomb, Door, FlyEnemy, Rock, Player, Wall, Floor, FloorImpl, GameWorld, GameWorldImpl, Room, RoomImpl, TestModel, StaticMethodUtils.

Integrazione

Per integrare il controller del menu al controller del gioco è stato aggiunto un SubMenu chiamato SubMenuGame contenuto nel GameSelection che ha come campo privato il GameController e nella propria view un campo chiamato GameView che viene creato dalla implementazione di SubMenuGameView utilizzata e passata direttamente al GameController che la utilizza per visualizzare a schermo il modello.

3.2.3 Asia Lucchi

Classi sviluppate autonomamente: Heart, AbstractHeart, SimpleHeart, BlackHeart, BodyComponent, FlyAIComponent, HealthComponent, MoveComponent, PlayerSpritesExtractor, AbstractHeartView, AbstractInventoryStatisticView, AbstractPlayerView, AbstractStatisticView, AbstractTearView, BlackHeartView, BlueTearView, BombStatisticView, BombView, CainView, DanksquirtView, FireView, FlyView, GaperView, HeartStatisticView, KeyStastisticView, KeyView, MonstroView, RedHeartView, RedTearView, RockView, StatisticView.

Classi sviluppate in collaborazione: Position, InventoryComponent, TearWeaponComponent, TearAIComponent, PickUpEvent, UseThingEvent, TearWeaponComponent, GameView, GameViewImpl, EntityView, AbstractEn-

tityView, IsaacView, SpritesExtractor, GameController.

3.2.4 Andrea Pantieri

Classi sviluppate autonomamente: MonstroAICompoent, EightInchNailsPickupableComponent, HostHatPickupableComponent, MawOfTheVoidPickupableComponent, StickyBombsPickupableComponent, Entity, AbstractEntity, EightInchNails, HostHat, MawOfTheVoid, MonstroBoss, StickyBombs, Event, AbstractEvent, EventListener, EightInchNailsView, HostHatView, MawOfTheVoidView, StickyBombsView, RoomView, Classi sviluppate in collaborazione: StaticMethodsUtils, Component, AbstractComponent, GameWorld, GameWorldImpl, GameView, GameViewImpl, GameController, EntityController, StatusComponent, CollisionComponent, MovableCollisionComponent, AbstractMentalityComponent, le varie enum, EntityView, AbstractEntityView, IsaacView, SpritesExtractor, i vari mentality component

3.3 Note di sviluppo

3.3.1 Cristian Casadei

Ho fatto uso delle seguenti feature avanzate:

- Reflection nell'entity controller per mappare enumerations in classi e metodi da chiamare.
- Lambda e Stream utilizzate soprattutto nella ricerca e/o aggiornamento di componenti e entità.
- Libreria esterna : Guava per la comunicazione tra componenti e tra entità utilizzando l'EventBus e iscrivendo e disiscrivendo i vari event listener.
- Ho gestito il parsing del XML nel metodo enumFromXmlToDataPlayer contenuto nelle StaticMethodsUtils e utilizzato dalla FactoryPlayersUtils per mappare l'enumerations di DataPlayer.

3.3.2 Michele Faedi

Il MenuSelection gestisce autonomamente l'inserimento di figli, tramite i generics e le wildcard, o altri MenuSelection e impostarsi come padre di quel

figlio. Se un MenuSelection viene inserito in un altro MenuSelection allora gli viene settato il padre, di conseguenza il padre è un Optional di MenuSelection. MenuSelection ha inoltre una funzione asStream che restituisce uno stream con tutti i nodi contenuti.
Per gestire il volume dei suoni ho creato un singleton privato chiamato VolumeManager della classe SoundJavafx che in base al nuovo valore e alla tipologia di audio aggiorna il volume di tutti i suoni con quella tipologia. Per gestire i tipi di audio è stato necessario utilizzare un generic Bounded chiamato Single che tramite una map contiene il valore del volume di ciascuna tipologia.
Per gestire il gran numero di lambda expressions abbiamo creato l'intefaccia Lambda che rappresenta una funzione senza parametro e Lambdas che rappresenta una funzione con una parametro generico. Sono state utilizzate in gran parte per gestire gli eventi dei JavaFx e far eseguire funzioni di controller agli oggetti di view(ad esempio cambio menu quando l'audio finisce) senza violare MVC.
Per caricare da file i dati dei giocatori ho usato una struttura xml in cui viene indicato l'immagine di anteprima, il giocatore da caricare e il percorso del package (per renderlo più estendibile, in questo modo è facile aggiungere nuovi giocatori) ottenendo così, tramite reflection, una enum che indica il giocatore selezionato. Su come convertire l'xml mi sono servito di questa guida:
https://www.tutorialspoint.com/java_xml/java_dom_parse_document.html .
Per gestire l'input in maniera asincrona tra il thread di javafx e il thread del model ho usato un Semaphore, quando il thread di javafx immette l'input nel GameController si blocca finche non viene rilasciato un biglietto, poi aggiorna un set contente i comandi inseriti.
Per l'algoritmo di ricerca mi sono basato su questa guida:
http://www.codebytes.in/2015/02/a-shortest-path-finding-algorithm.html
Operando delle modifiche per non dover rigenerare tutto quando rieseguo l'algoritmo.
Per la struttura dell'interfaccia è stato usato un file fxml che ha la funzione di modello. Poi siccome ho dovuto fare in modo che l'interfaccia si ridimensioni mantenendo i rapporti e spostando i Pane mantenendoli al centro ho usato le Property di javafx che permettono di legare un valore ad un altro, per esempio alla scala del Pane che a sua volta è legata alla dimensione dell'interfaccia. Per questo non mi sono servito di nessuna guida se non quella di javafx per gestire le Property.

3.3.3 Asia Lucchi

Non sono riuscita a utilizzare buona parte dei pattern discussi a lezione in quanto ho speso la maggior parte del tempo a rendere funzionanti le classi da me implementate, mi sono principalmente impegnata a rimanere coerente con i pattern principali che si è deciso di utilizzare: MVC, EntityController e Publish-Subscribe.
Ho fatto uso delle seguenti feature:

- Stream per l'accesso o la generazione degli elementi di liste o mappe.
- Reflection per accedere ai componenti delle varie entità.
- Generici e WildCard nei parametri dei metodi: per renderli riutilizzabili ed allo stesso tempo ridurre le possibiltà a quelle plausibili in base allo specifico utilizzo del parametro nel metodo.
- Libreria esterna Guava: utilizzo dell'EventBus, in particolare registrare e de-registrare listener, postare eventi.

3.3.4 Andrea Pantieri

Per le parti di codice da me sviluppate, ho cercato di utilizzare il più possibile feature avanzate di Java e di librerie esterne: Guava e JavaFX. Di Guava ho utilizzato molto l'EventBus per la gestione degli eventi all'interno di un'entità e della comunicazione fra le varie entità quindi implementando il pattern Publish/Subscribe. Per JavaFX ho utilizzato il canvas per disegnare la parte grafica del gioco comprendente le stanze, nemici e il player. Tornando alle feature di Java ho usato principalmente i generics specialmente bounded per tenere limitare i salvataggio nell'entity ai soli componenti che implementassero l'interfaccia Component e per la creazione di un EventListener generico. Ho anche usufruito di stream, contenenti lambda expression per le flat map e filter, e di optional in vari metodi nel model per migliorare certe ricerche e/o operazione all'interno delle diverse collezioni di dati specialmente nei set/liste per i componenti e i listener degli eventi. Assieme agli stream ho fatto un abbondante uso di reflection specialmente nei metodi statici contenuti nella util StaticMethodsUtils per risolvere varie necessità. Per finire ho gestito tutto il parsing XML per la creazione automatica del gioco e per sincronizzare in maniera automatica model e vista.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Cristian Casadei

Sono molto soddisfatto di come sia stato realizzato il progetto, abbiamo avuto sempre un occhio di riguardo sull'estendibilità, caratteristica che ritengo essere il vero punto di forza del progetto. Penso che un altro punto di forza del progetto sia il fatto che le gerarchie implementate ad esempio nei package collision, mentality e collectible rendano minima la ripetizione del codice. L'unico vero punto di debolezza che trovo è quello di aver implementato pochi test per testare fino in fondo quanto il codice sia corretto a livello computazionale.

4.1.2 Michele Faedi

Mi ritengo pienamente soddisfatto di come ho realizzato la gestione dei controller. Mi riferisco al MenuSelection che permette alla applicazione di avere molta estendibilità grazie al fatto che gestisce autonomamente il poter contenere direttamente dei figli o altri MenuSelection.
Sono molto felice di poter veder il menu di gioco molto simile al menu originale considerando il poco tempo disponibile personale per questo progetto. Dovendomi occupare del Menu sia del Controller sia della View secondo me è venuto un ottimo lavoro considerando che come obbiettivo c'era quello di imitare il più verosimilmente possibile il gioco originale. Ho trovato complicanze dovute ad una sottovalutazione della difficoltà nelle fase di analisi nel gestire tutta la parte di Controller e di View che riguarda il menu principale il che mi ha portato a dover creare un gran numero di classi, interfacce e algoritmi per far funzionare tutto e renderlo estendibile al massimo delle mie

capacità. Come punti di debolezza può essere stato il fatto di non essere riusciti, come gruppo, ad implementare l'audio dei nemici e delle stanze.

4.1.3 Asia Lucchi

Sono soddisfatta del lavoro da me svolto, in particolare delle classi riguardanti la grafica delle entità che ho sviluppato, praticamente, in maniera autonoma riuscendo a mettere in pratica gli aspetti fondamentali della programmazione ad oggetti che sono stati affrontati durante il corso. Il mio codice, soprattutto quello del model, ha certamente bisogno di ulteriore refactoring sebbene sia stato riscritto ed analizzato più volte, perciò non mi ritengo pienamente soddisfatta. Ho cercato di contribuire il più possibile anche nelle fasi di progettazione comune ma essendo la prima volta che utilizzavo un linguaggio ad oggetti mi sono lasciata guidare da altri elementi del gruppo che vi avevano già avuto a che fare.
Ritengo invece che il progetto nella sua interezza sia ben fatto ed estendibile, quest'ultimo aspetto è importante in quanto è stato uno dei punti chiave della filosofia del gruppo.
Ho l'impressione che nel codice vi siano dei punti di debolezza dato che a mio parere esso non è stato adeguatamente testato e refattorizzato di conseguenza. Cioè è dovuto al fatto che abbiamo utilizzato buona parte del tempo a cercare di rendere più estendibile il codice, lavorando più sulla teoria e lo stile che sulla pratica facendo debug.

4.1.4 Andrea Pantieri

Sono onestamente soddisfatto di come ho ed abbiamo portato avanti il progetto, specialmente per come sono state progettate alcuni parti del programma per renderle più estendibili. Io personalmente penso di aver contribuito il giusto specialmente nella parte iniziale di progettazione ed, in seguito, anche nel momento di sviluppo.
Come punti di debolezza, avrei voluto strutturare meglio alcune parti del programma quali quella dei potenziamenti all'interno del model.
Come punti di forza invece, credo che ci siano l'analisi di certi problemi, tra cui rendere il codice pulito, estendibile ed anche automatizzare certi processi evitando possibilmente if else oppure switch cases. Inoltre, mi sento soddisfatto per la struttura data ai componenti principali (Mentality e Collision) che è stata pensata per facilitare l'aggiunta di nuove versione in futuro. Altro punto di nota, secondo me, è stata la gestione ad albero delle enum (per poter simulare anche qui un codice estendibile) e la tale sincronia automatizzata con i vari metodi della view tramite xml.

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Michele Faedi

Io personalmente avrei preferito se si fossero state più lezioni riguardanti i pattern da utilizzare. Ad esempio il pattern EntityComponent è stato trovato quasi per caso dalla Asia Lucchi. Se fosse stato spiegato a lezione sarebbe stato meglio, non mi immagino quali altri pattern potrebbero essere utilizzati o che abbiamo utizzato senza saperlo.

4.2.2 Asia Lucchi

Mi sono trovata in difficoltà soprattutto nella fase iniziale, avrei apprezzato se nella parte finale del corso ci fosse stata qualche lezione in più riguardante la progettazione utilizzando MVC.

Appendice A

Guida utente

L'applicativo ha un Menu abbastanza intuitivo. Per selezionare le varie caselle si usano le frecce della tastiera. Per selezionare una casella si usa invio. Nel gioco per mettere in pausa si usa il tasto Esc. Nel gioco ci si muove con i tasti WASD, invece per sparare le lacrime si usano le frecce. Per posizionare una bomba si usa la q.
In alto a sinistra ci sono delle informazioni relative allo stato del giocatore.
Per terminare il gioco basta cercare il boss del piano, sconfiggerlo e esso rilascerà un oggetto. Per vincere basta toccare quell'oggetto.