Indice

ANALISI

1.1 Requisiti

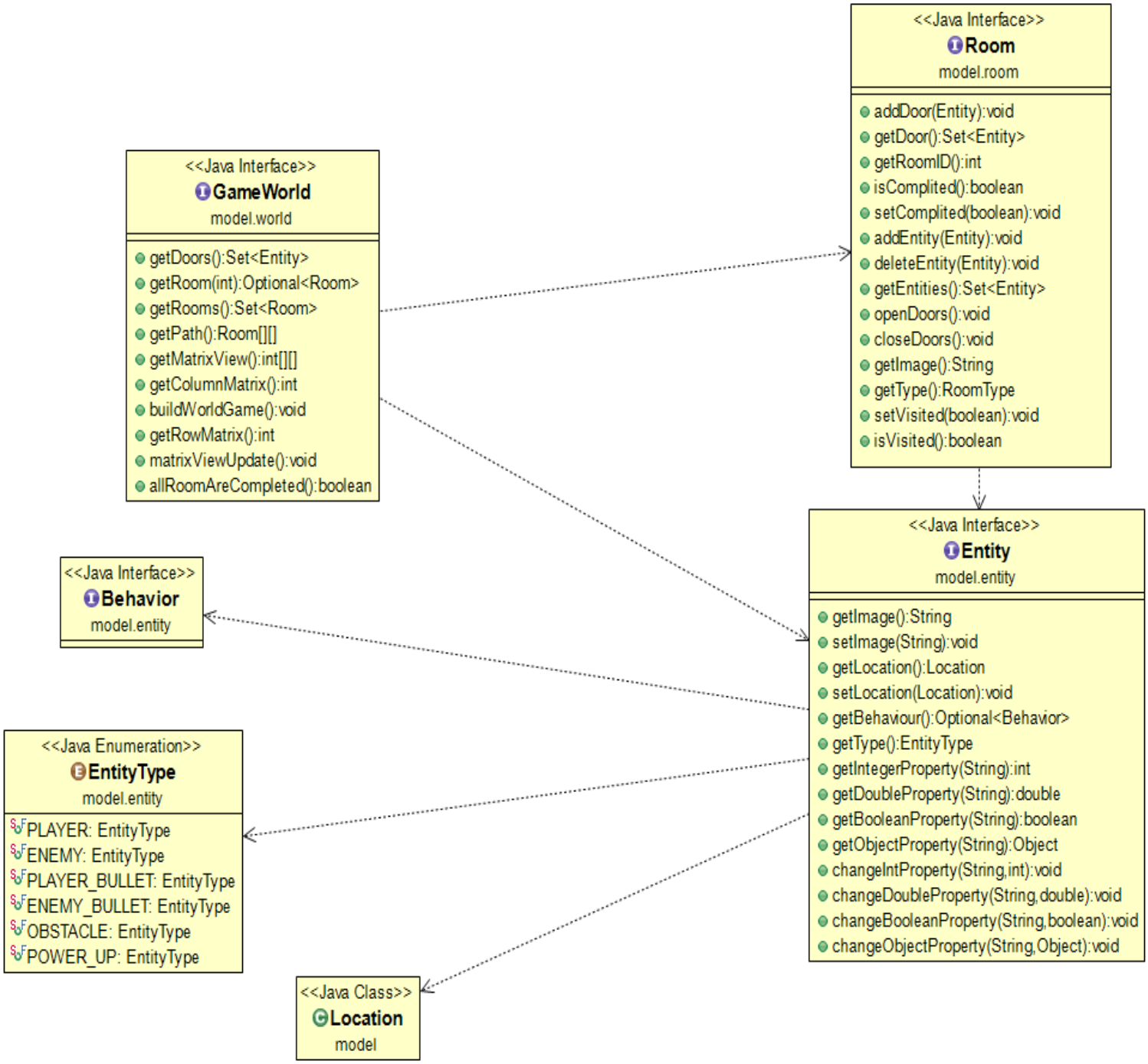
Il software mira alla realizzazione di un gioco simile a The Binding of Isaac. Dovrà quindi mostrare un personaggio (controllato dall'utente) che si muova, spari e possa cambiare stanza passando attraverso delle porte. Nelle varie stanze troverà nemici che dovrà sconfiggere per andare avanti controllati dal computer, power up e dei nemici più complessi i boss. Lo scopo del gioco è uccidere tutti i nemici di tutte le stanze nel minor tempo possibile. Il mondo di gioco sarà rappresentato da un insieme di stanze limitrofe la cui disposizione darà origine a uno o più percorsi.

- L'utente dovrà essere in grado di controllare il player, di farlo muovere e di farlo sparare
- Dovranno essere gestite le collisioni tra le varie parti del mondo di gioco
- Dovrà essere data all'utente la possibilità di scegliere tra quattro possibili personaggi per giocare
- Se l'utente raggiunge l'obbiettivo del gioco allora gli dovrà essere data la possibilità di salvare il suo tempo di gioco. Dovrà essere fatta una classifica che determini i dieci migliori tempi che possa essere conservata anche dopo la chiusura dell'applicativo
- Si dovrà poter visualizzare la mappa in cui siano rappresentate le stanze già visitate, e in cui si dovrà evidenziare la stanza in cui è presente attualmente il giocatore
- L'ambiente di gioco (Mappa) sarà suddiviso in stanze generato in modo pseudo-randomico. Per pseudo-randomicità si intende che la struttura delle varie stanze sia già predefinita ma sarà la loro predisposizione a cambiare. La scelta è stata fatta per evidenziare il fatto che l'obiettivo principale non è il semplice completamento delle stanze ma terminarle nel minor tempo possibile. In questo modo ogni partita viene resa equa ma comunque viene mantenuta una certa variabilità del contenuto.
- L'esperienza di gioco dovrà essere arricchita attraverso l'aggiunta di musica

1.2 Analisi e modello del dominio

Il mondo di gioco (GameWorld) sarà organizzato in stanze (Room) limitrofe collegate tra di loro da porte che definiscono vari percorsi all'interno dei quali il giocatore si potrà muovere. Nell'attuale versione del gioco si è deciso di creare tre percorsi separati alla fine dei quali si trova uno specifico boss. La motivazione di questa scelta è la seguente: il gioco punta sul fatto che vengano completate tutte le stanze nel minor tempo possibile quindi l'utente non deve perdere tempo nel ricercare la strada per delle stanze in cui non è entrato e per far ciò i percorsi devono essere più lineari possibili. La posizione delle stanze sarà diversa al momento dell'avvio del gioco ma il percorso sarà sempre continuo e non ci saranno stanze separate dalle altre. Ogni cosa presente nelle stanze sarà una entità (Entity). Ogni entità avrà una posizione e un'area che ne rappresenta la sua estensione nello spazio, una componente che ne determina il comportamento (Behavior), un tipo che le contraddistingue, e un'immagine che ne fornisce una rappresentazione grafica. A ogni entità sono poi associate delle proprietà che ne forniranno la loro caratterizzazione. Attualmente tutte le entità presenti nel gioco sono:

- 4 giocatori che sono la reificazione dei quattro ideatori del progetto. Ogni personaggio presenta le seguenti caratteristiche: una velocità di movimento, una certa frequenza di sparo, un determinato danno di attacco e una quantità di vita massima. Ogni giocatore possiederà una somma di denaro che inizialmente è 0 e che aumenterà uccidendo boss e nemici. Tramite questi potrà potenziarsi, avendo la necessaria somma di denaro e andandosi a scontrare con i power up.
- 3 tipi di nemici: uno somigliante a una mosca che insegue il giocatore, uno somigliante a uno spiritello con le ali che segue anch'esso il giocatore, e un altro spiritello che si presenta nella forma identico al primo che però spara anche al giocatore
- 3 tipi di boss: uno che insegue il giocatore e gli spara, uno che si muove in maniera casuale spara al giocatore e evoca dagli angoli della stanza dei proiettili che percorrono le diagonali della stanza e infine uno che si comporta come quello precedentemente descritto ma è in grado anche di evocare i nemici-mosca
- Le porte che saranno chiuse finché non saranno sconfitti tutti i nemici presenti nella stanza
- Dei sassi che impediranno il movimento
- I proiettili sparati dal giocatore e dai nemici che si distingueranno per una diversa rappresentazione grafica
- 4 power up: sigarette che permetteranno al giocatore di riacquistare la vita, una chitarra che aumenta la frequenza con cui si può sparare, una pistola d'oro che aumenta potenza di attacco e dello zucchero sintattico rappresentato da un bacchetto di zucchero che aumenta la velocità di movimento. Ogni power up avrà un costo.



Le collisioni tra le varie entità dovranno dar luogo a conseguenze diverse a seconda dei casi:

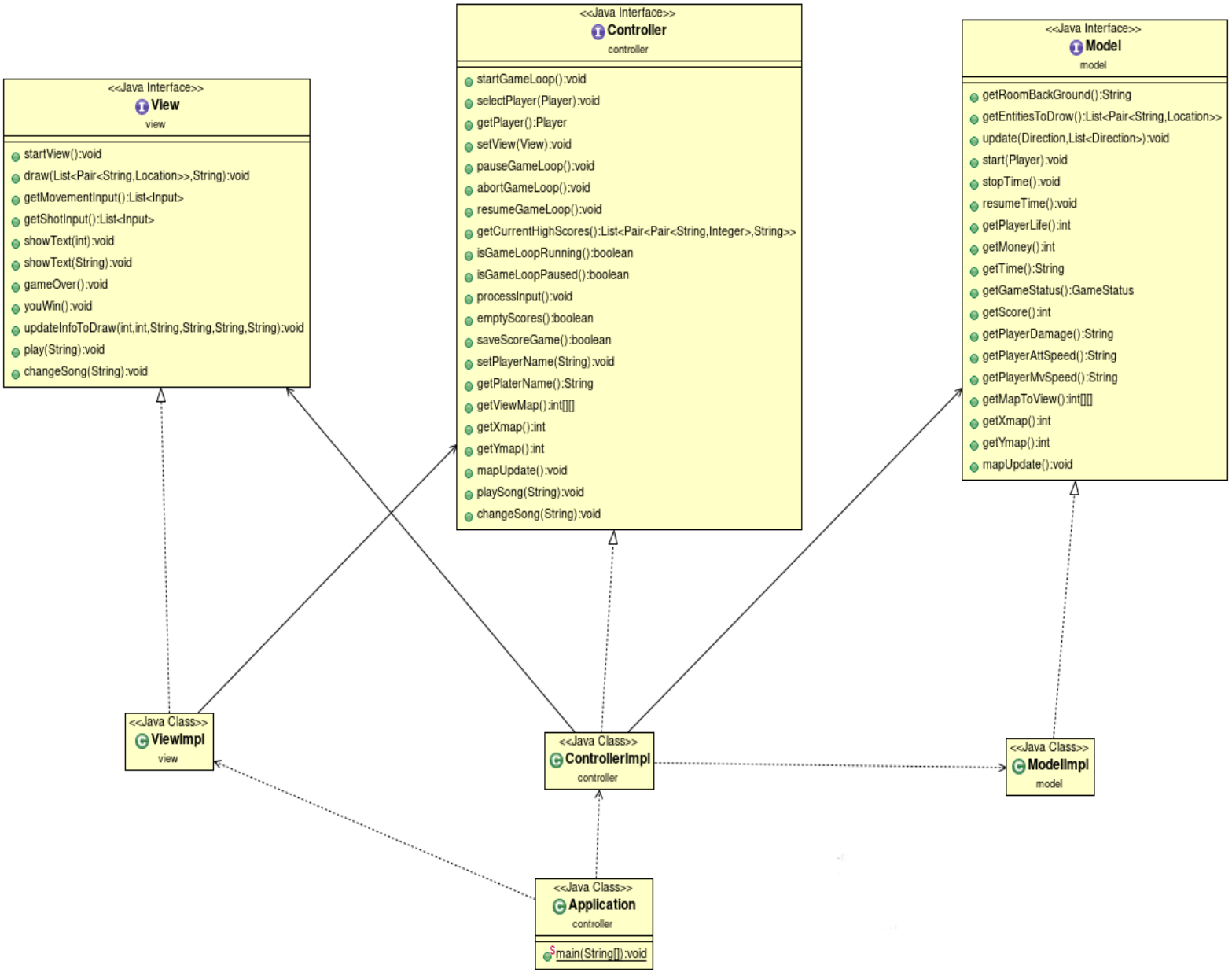
- Se un proiettile sparato dal giocatore colpisce un nemico questo deve perdere vita in base al danno del giocatore e viceversa
- Se un giocatore collide con un power up e ha denaro maggiore uguale al costo del power up lo ottiene e beneficia dei miglioramenti del power up. Diminuirà i soldi che ha a disposizione in base al costo di power up.
- Se un'entità mobile collide con un ostacolo non potrà oltrepassarlo ma rimarrà bloccato tranne i proiettili che dovranno essere eliminati dal mondo di gioco

- Se un’entità mobile collide con i bordi della stanza deve rimanere bloccata tranne i proiettili che dovranno essere eliminati dal mondo di gioco
- Se il giocatore collide con i nemici nella stanza deve perdere gli opportuni punti vita
- Se un giocatore collide con le porte e queste sono aperte deve poter passare nella stanza adiacente

Nota sul movimento dei nemici controllati dal computer: è vero che essi fanno delle valutazioni sugli ostacoli, sul giocatore che possono inseguire, e sui limiti spaziali che la stanza gli impone ma la loro intelligenza non gli permette di fare valutazioni su altri elementi della stanza ad esempio gli altri nemici. Si è scelto di semplificare questi comportamenti per rispettare il monte ore previsto.

DESIGN

2.1 Architettura



Per lo sviluppo dell'applicazione si è deciso di utilizzare il pattern MVC che permette di dividere il software in una parte logica, il Model, una parte di visualizzazione, la View, e una parte che mette in comunicazione gli altri elementi e coordina l'effettivo funzionamento del gioco, il Controller. Per la comunicazione tra le parti è previsto che la View non interagisca direttamente con il Model ma richieda le informazioni necessarie al Controller. Quest'ultimo invece interagisce con entrambi sia con il Model che con la View. All'avvio dell'applicativo vengono inizializzati Controller e View e solo successivamente all'interno del Controller sarà inizializzato il Model. Guardando a possibili migliorie future si è stati attenti a far comunicare le parti solamente utilizzando i metodi delle

interfacce in modo che se una sola parte di queste dovrà essere intaccata basterà modificarla senza cambiare le altre.

Il Model rappresenta e gestisce tutto il mondo di gioco. Si preoccupa di aggiornarsi anche in base alle informazioni che il controller gli passa dalla View e mette a disposizione del controller le informazioni necessarie sia per l'aggiornamento dell'interfaccia grafica sia per le operazioni proprie del Controller stesso.

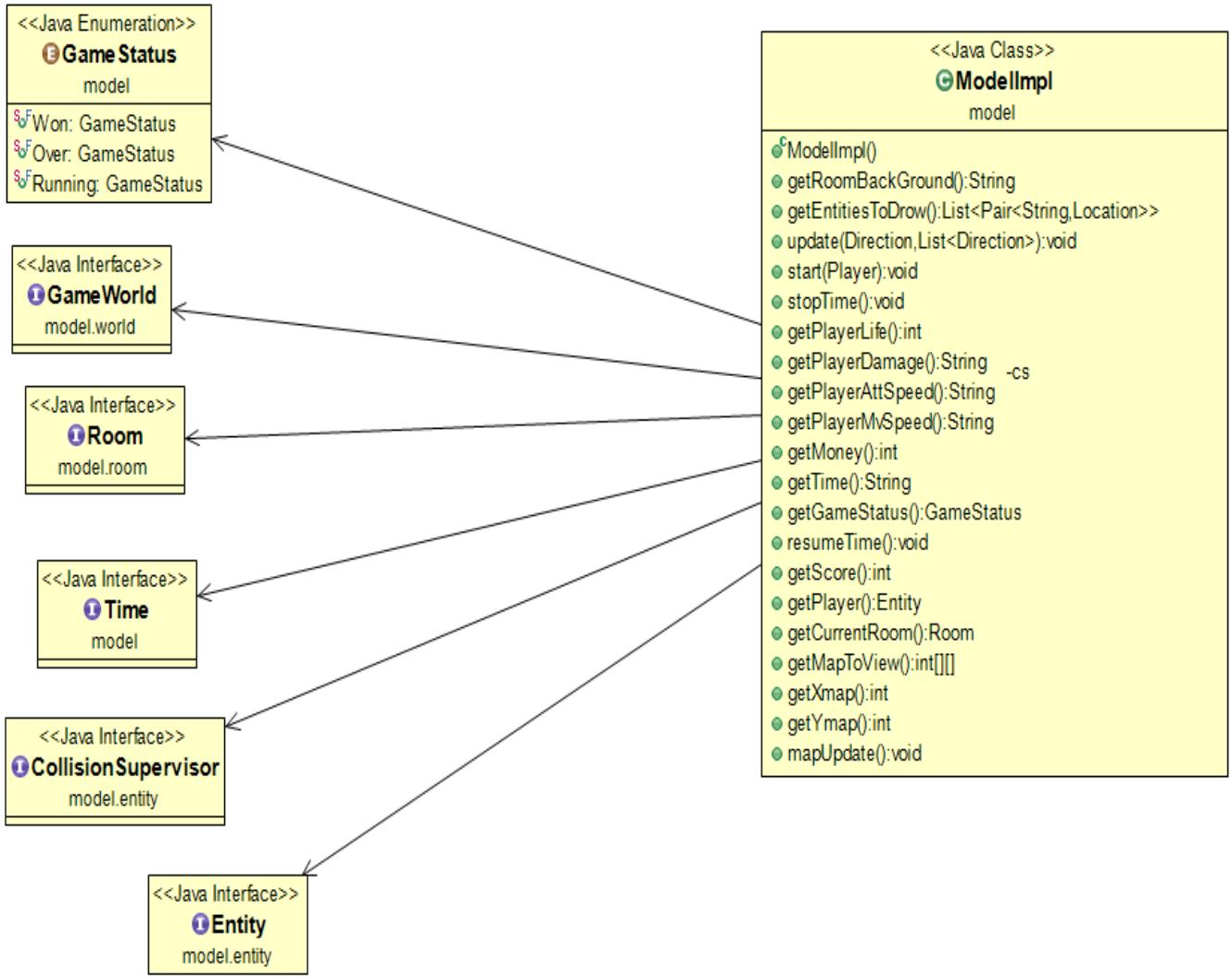
La View è stata pensata per essere unicamente quella parte del gioco che si occupa della rappresentazione e dell'interazione con l'utente evitando che si debba occupare di qualsiasi aspetto logico del gioco. Dal Model, tramite il Controller, sia aspetta di conoscere come disegnare la scena di gioco e quali sono le informazioni da stampare da rendere note all'utente come ad esempio la vita del giocatore, i soldi che ha a disposizione… Inoltre mette a disposizione del Controller gli input inseriti dall'utente.

Il Controller come già detto si pone come intermediario tra le due parti e come regista del videogioco: comanda al Model e alla View di aggiornarsi, traduce gli input della View per comunicarli al Model e passa le informazioni dal Model alla View. Il Controller si occupa anche della gestione dei salvataggi

2.2 Design Dettagliato

- Del Gatto

MODEL

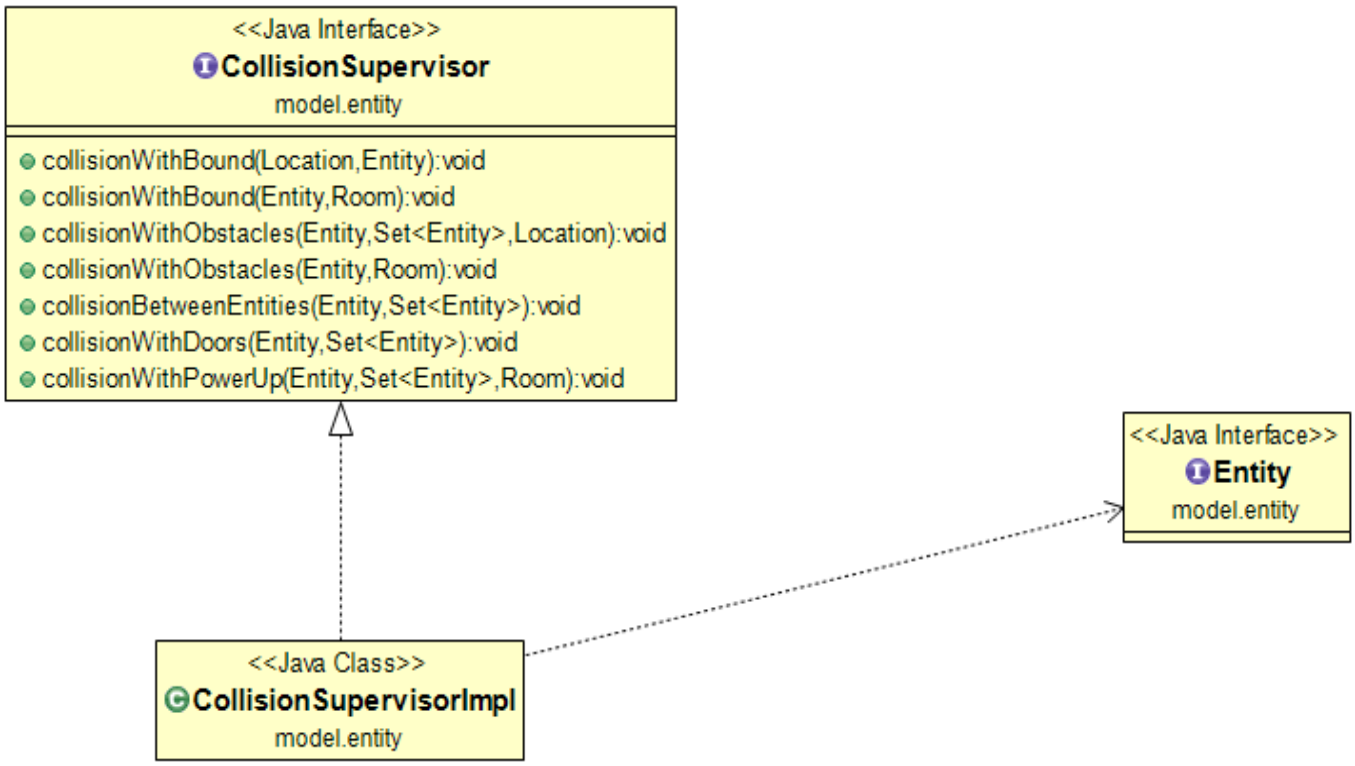


Il Model si occupa di gestire tutto quello che è il mondo di gioco dalle stanze a tutte le entità presenti, il giocatore e le interazioni tra le entità ed è implementato dalla classe ModelImpl.

Viene effettivamente creato durante l'inizializzazione del Controller attraverso il metodo start() che prende in ingresso un parametro che indica quale tra i quattro player presenti è stato selezionato dall'utente. In questo metodo vengono creati il gestore delle collisioni, la classe che gestisce la creazione delle entità, che è stata implementata utilizzando il pattern Factory, da cui viene creato il player in base al parametro in ingresso, viene creato il mondo di gioco con tutte le sue stanze ed entità tramite la classe GameWorldImpl, che è l'effettiva implementazione dell'interfaccia GameWorld e infine viene fatto partire il thread che gestisce il conteggio del tempo.

Ad ogni ciclo il GameLoop si occupa di far si che il mondo di gioco si aggiorni utilizzando il metodo update() del ModelImpl che prende in ingresso due parametri: una direzione che rappresenta la direzione verso cui l'utente vuole far spostare il player, e una lista di direzioni che rappresentano una lista di direzioni verso cui l'utente vorrebbe sparare. Le direzioni non sono altro che istanze dell'enumerazione Direction la quale oltre a interpretare in maniera digitale il concetto di direzione ha anche l'importante compito di permettere il movimento delle entità con il metodo changeLocation().

Nel metodo update() viene per prima cosa chiesto al player di aggiornarsi in base alla direzione richiesta è di effettuare gli spari e lo si fa chiamando sulla componente Behavior del player il metodo update(). Nel ModelImpl viene tenuta traccia della stanza corrente in cui il player è collocato e grazie a questo riferimento e al metodo getEntities(), che restituisce una collezioni di entità presenti nella stanza, è possibile aggiornare tutte le altre entità stanza chiamando, se la componente Behavior è presente, il metodo update() (tutto ciò risulterà più chiaro quando sarà spigato come sono effettivamente gestite le entità).



Nel Model viene sfruttato un CollisionSupervisor implementato dalla classe CollisionSupervisorImpl che controlla se le entità stanno collidendo tra di loro e in caso affermativo applica i giusti provvedimenti: se un proiettile del giocatore colpisce un nemico questo perde la giusta quantità di vita, se il giocatore si scontra con un nemico perde punti vita… Per fare questo vengono effettivamente utilizzati due metodi del ColllisionSupervisor che sono `collisionBetweenEntities()` e `collisionWithPowerUp()`. Entrambe prendono in ingresso una singola entità e una lista di entità con cui ipoteticamente potrebbe collidere che in questo caso specifico è la lista di entità presenti nella stanza. Il primo si occupa delle collisioni con le diverse

entità della stanza tranne i power up, di cui ci si occupa nel secondo metodo. La divisione dei due metodi nonostante operino sullo stesso insieme di elementi consente di mantenere il codice più pulito e inoltre di poter subito identificare in caso di eventuali cambiamenti quali sono le parti che gestiscono il comportamento che si vuole modificare.

L'ultima collisione che viene gestita, e anche in questo caso per i motivi precedentemente descritti viene fatto a parte, è la collisione con le porte. Prima viene fatto un controllo utilizzando il metodo isComplited() dell'interfaccia Room che permette di sapere se la stanza corrente è completa o meno. In caso affermativo si aprono le porte e se il player collide con una di queste (si utilizza il metodo collisionWithDoors()) viene posizionato in un'altra stanza.

L'ultimo controllo che viene fatto è quello sulla vita del giocatore. Se la vita minore o uguale a zero il Model deve rendere disponibile questa informazione al GameLoop. Per farlo utilizza un campo della sua classe a cui si può accedere con un getter che rappresenta lo status di andamento del gioco: il gameStatus che non è altro che una istanza dell'enumerazione GameStatus. Quest'ultima rappresenta le tre fasi del gioco: Running, Over, Win. A seconda di quale è presente nel campo di cui prima si parlava il GameLoop attiverà diverse procedure.

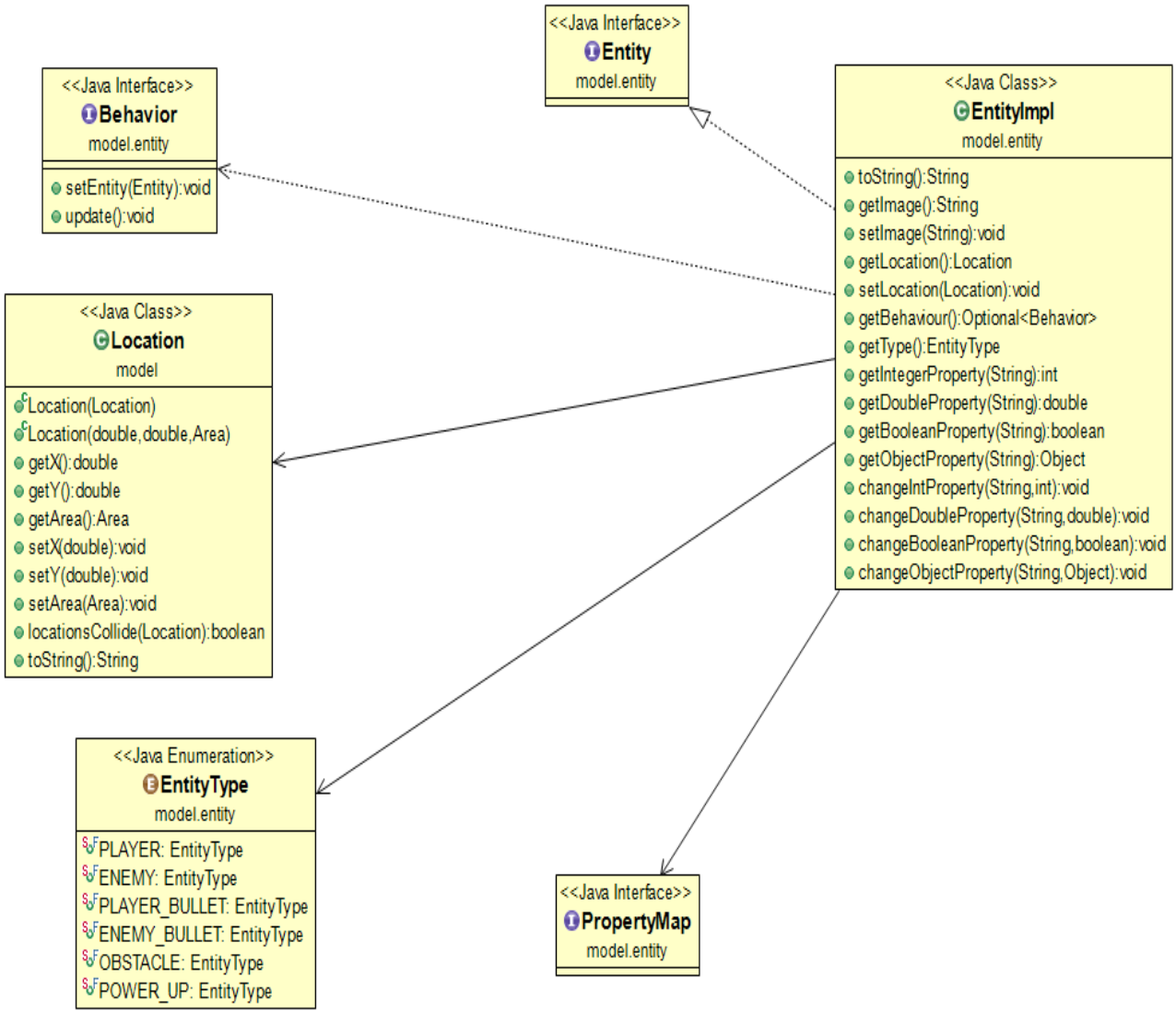
Altri metodi che sono richiamati del model ad ogni ciclo del GameLoop sono tutti quelli attraverso i quali si accede a alle statistiche del giocatore e al tempo di gioco, e i due metodi getEntitiesToDrow() e getRoomBackGround(). Il secondo restituisce semplicemente il path dell'immagine di background della stanza, il primo invece restituisce una lista di coppie stringhe e Location. Attraverso queste informazioni che il Controller poi passa alla View viene disegnata la scena.

Nelle sezioni successive si approfondiranno aspetti delle entità e della mappa che qui non erano presenti per questioni di chiarezza.

ENTITA'

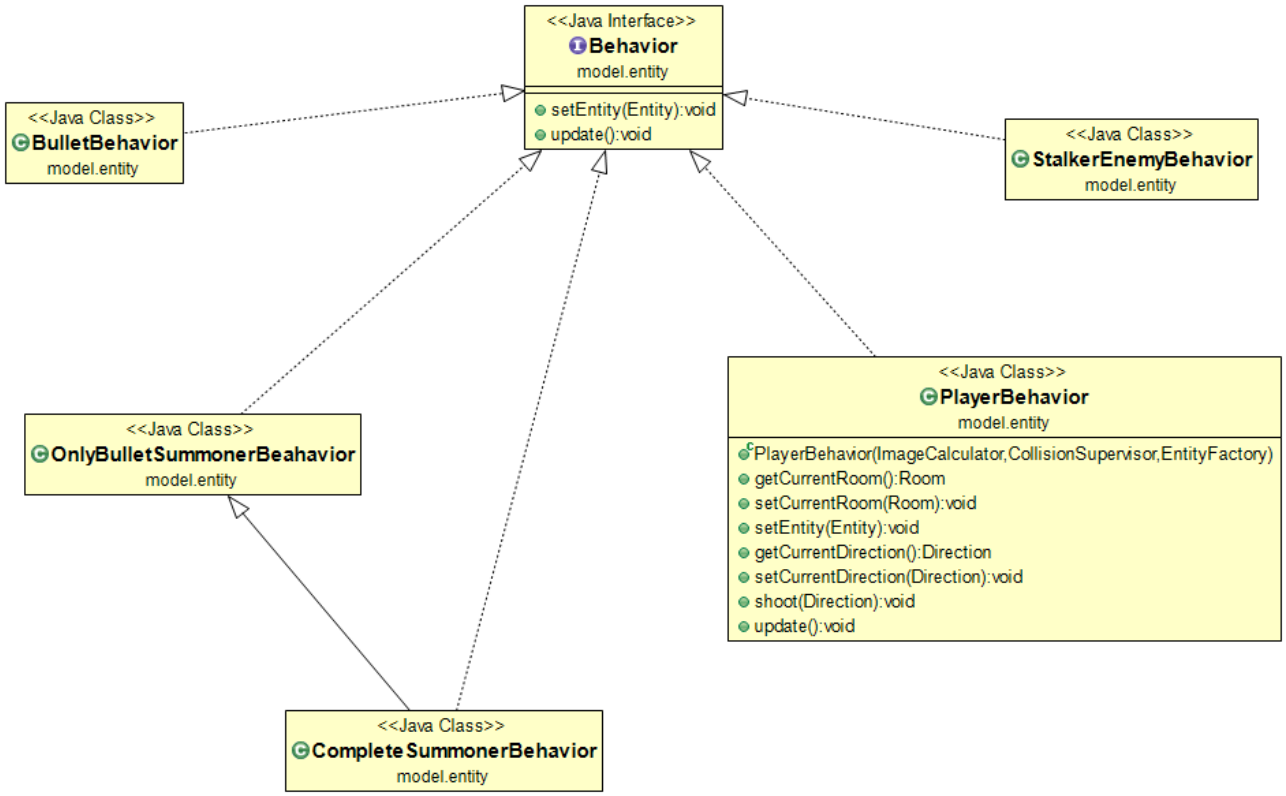
L'obbiettivo che si voleva raggiungere era trovare un modo per far si che la gestione delle entità fosse il più generale possibile e che fosse completamente riutilizzabile: l'idea è che la modalità con cui vengono create le entità potesse essere utilizzata non solo in questo progetto ma in qualsiasi progetto simile. Con questi presupposti si è cercato di identificare quali sono le caratteristiche comuni per tutte le entità che compaiono in un possibile mondo di gioco in due dimensioni: un'immagine che rappresenti graficamente l'entità, due double che rappresentano la sua posizione sull'asse delle ascisse e sull'asse delle ordinate, un'area, con altezza e larghezza dell'entità, e una componente che definisca il comportamento dell'entità. Le entità sono però in genere tutte diverse tra loro e con proprietà differenti che in genere si manifestano differenziando le entità in classi diverse. Questo approccio è chiaro che presenta una certa rigidità che modella le entità solo nello specifico del dominio applicativo. Si è quindi pensato di far si che le proprietà che denotano le caratteristiche di un'entità possano essere aggiunte, accedute e modificate liberamente utilizzando una stringa che definisce il nome che si vuole dare a questa proprietà e alcuni opportuni metodi. Le entità quindi non appartengono più a uno schema rigido fossilizzate in classi diverse ma fanno parte tutte di una stessa categoria e solo successivamente saranno modellate in maniera particolare. L'interfaccia Entity cerca quindi qualche modo di rappresentare buona parte delle entità che possono venire in mente creando un semplice videogioco in 2D. Ci

sono metodi per modificare proprietà di qualsiasi tipo (double, interi, oggetti), accedere alla Location di una entità, al suo EntityType, all'immagine che la rappresenta (che è la stringa del percorso dell'immagine) e, nel caso ne abbia uno, alla componente che ne definisce il comportamento, il suo Behavior.



Più nel dettaglio la Location è un oggetto che esprime le caratteristiche spaziali di una entità specificandone posizione lungo l'asse y, posizione lungo l'asse x e attraverso un oggetto della classe Area la sua altezza e larghezza.

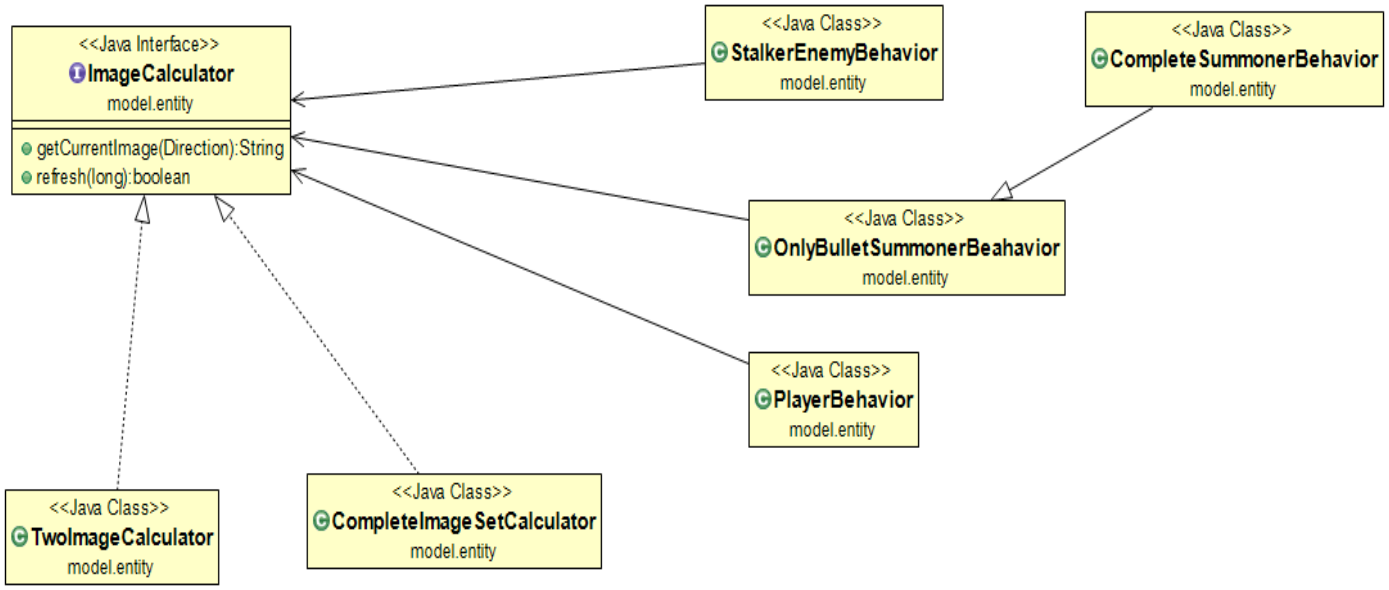
L'EntityType è un'enumerazione che definisce, almeno nel nostro dominio applicativo poi in altri contesti può essere ripensato, se l'entità è un nemico, un player, un ostacolo, un boss o un power up.



Il Behavior invece come già detto e livello concettuale esprime il comportamento di una entità e si è scelto di averlo come opzionale perché ci sono alcuni elementi che chiaramente non hanno alcun tipo di comportamento (un sasso ad esempio). L'interfaccia Behavior richiede che vengano implementati due metodi: il metodo setEntity() che prende in ingresso una entità e il metodo update(). Il Behavior di un'entità necessita di avere un riferimento dell'entità a cui è associato per poter fare uso di tutte le proprietà che le appartengono e questo giustifica il primo metodo. Il secondo invece è il metodo che quando viene chiamato permette al Behavior di aggiornare lo stato dell'entità a cui è associato e di fargli compiere delle azioni. Risulta anche una cosa molto comoda perché quando si vogliono aggiornare più entità in una volta basta chiamare il metodo update() su tutte magari utilizzando una Lambda Expression come viene fatto poi in realtà nel Model quando viene aggiornato. Nel progetto ci sono diverse classi che implementano questa interfaccia ad esempio PlayerBehavior, oppure StalkerEnemyBehavior le quali implementano effettivamente i comportamenti delle entità decise in fase di progettazione.

Per fattorizzare meglio il codice si è deciso di isolare la parte con cui si sceglie con quale immagine l'entità deve essere visualizzata sullo schermo in modo da simulare anche il movimento se l'entità è mobile. A livello concettuale questo compito dovrebbe essere affidato al Behavior dell'entità. Nelle effettive implementazioni si è però deciso di delegare questo compito a un ImageCalulator(). Questo presenta attualmente due implementazioni TwoImageCalculator e CompleteImageSetCalculator. Si specifica ancora che per quanto riguarda la gestione delle immagini nel Model vengono tenuti soltanto i percorsi delle immagini da stampare a video. Il fatto che la gestione delle immagini sia gestita a parte presenta anche un ulteriore vantaggio. Si prenda ad esempio la classe StalkerEnemyBehavior. Questa classe è necessaria per creare tutti e tre i tipi di nemici sia quelli somiglianti a delle mosche sia quelli che somigliano a degli spiriti. Si usa per

tutti lo stesso Behavior ma si utilizzano due ImageCalculator differenti per dargli una caratterizzazione grafica differente. Nelle implementazioni dei Behavior in sostanza viene solamente chiamato il metodo getCurrentImage() e viene settata l'immagine dell'entità a cui è associato in maniera corretta.

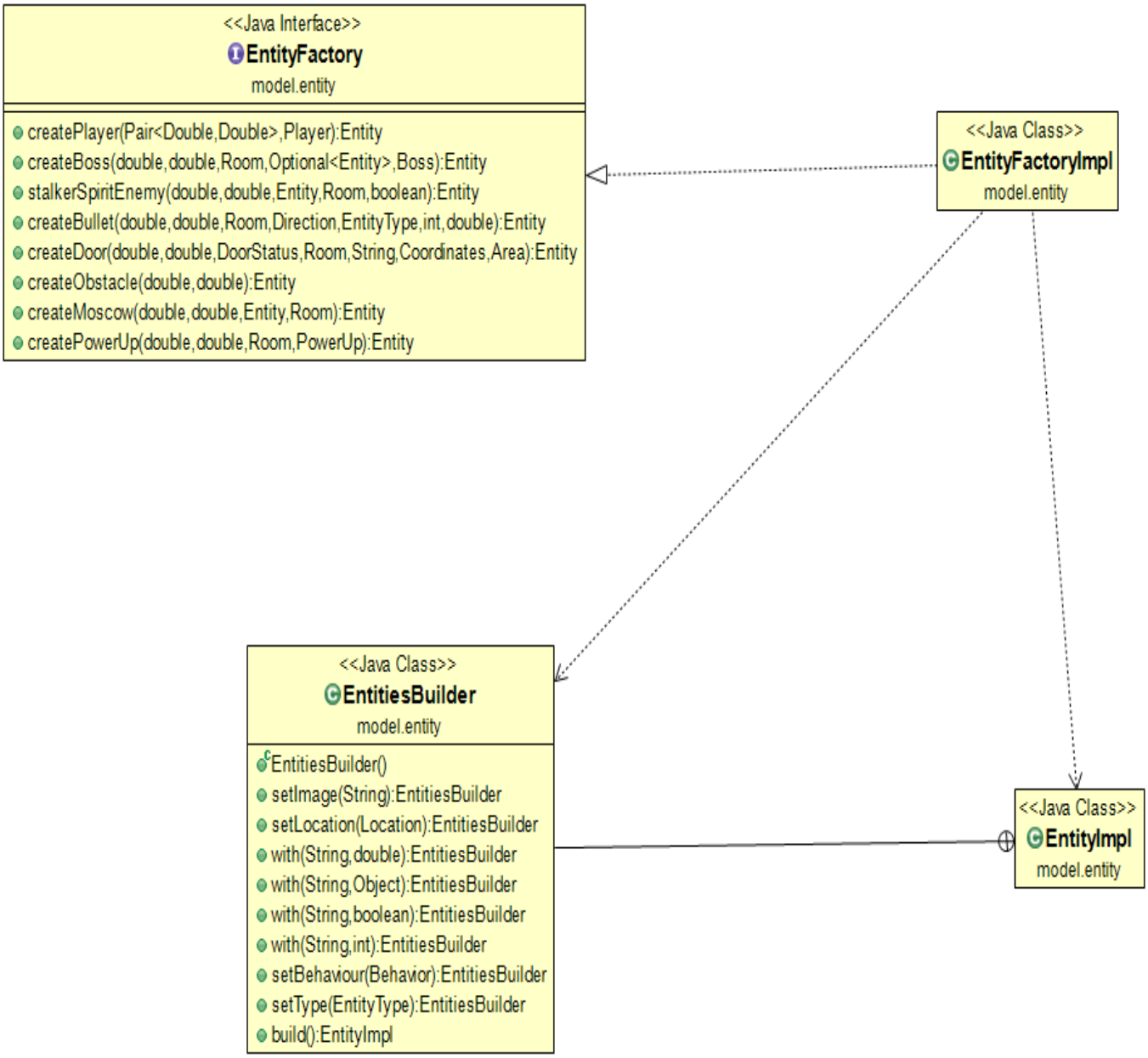


La classe che implementa l'interfaccia Entity è la classe EntityImpl. Ci sono due scelte importanti da notificare.

La prima è che per rendere ancora più fluida la creazione delle entità si è scelto di inizializzarle tramite il pattern Builder.

La seconda è che le varie proprietà che definiscono le entità si è deciso di conservarle in una PropertyMap la cui implementazione è PropertyMapImpl. Essa non è altro che una interfaccia a una mappa di oggetti con metodi per aggiungere e modificare proprietà di vario tipo. Questa scelta è motivata da questo fatto: gestire una mappa di oggetti senza un'interfaccia costringe il programmatore a operare operazioni di cast esplicito continuamente e questo abbruttisce il codice e rallenta la programmazione. Già i cast sono inevitabili in certi punti (ad esempio quando si vogliono utilizzare proprietà di tipo oggetto) però si è cercato di evitare che avvengano anche quando si vogliono manipolare i tipi primitivi più utilizzati.

Si è deciso di creare una classe utilizzando il pattern Factory che permettesse di creare in maniera semplice e pulita i 4 personaggi disponibili per il player, i 4 power up, i 4 boss, i tre tipi d nemici concordati nella fase di progettazione. La classe EntityFactoryImpl, che implementa l'interfaccia EntityFactory, prende in ingresso un CollisionSupervisor per passarlo a tutte quelle entità che hanno un Behavior che lo richiede. Se successivamente si vorrà introdurre un nuovo tipo di nemico basterà aggiungere un metodo o modificarne appena uno già esistente.

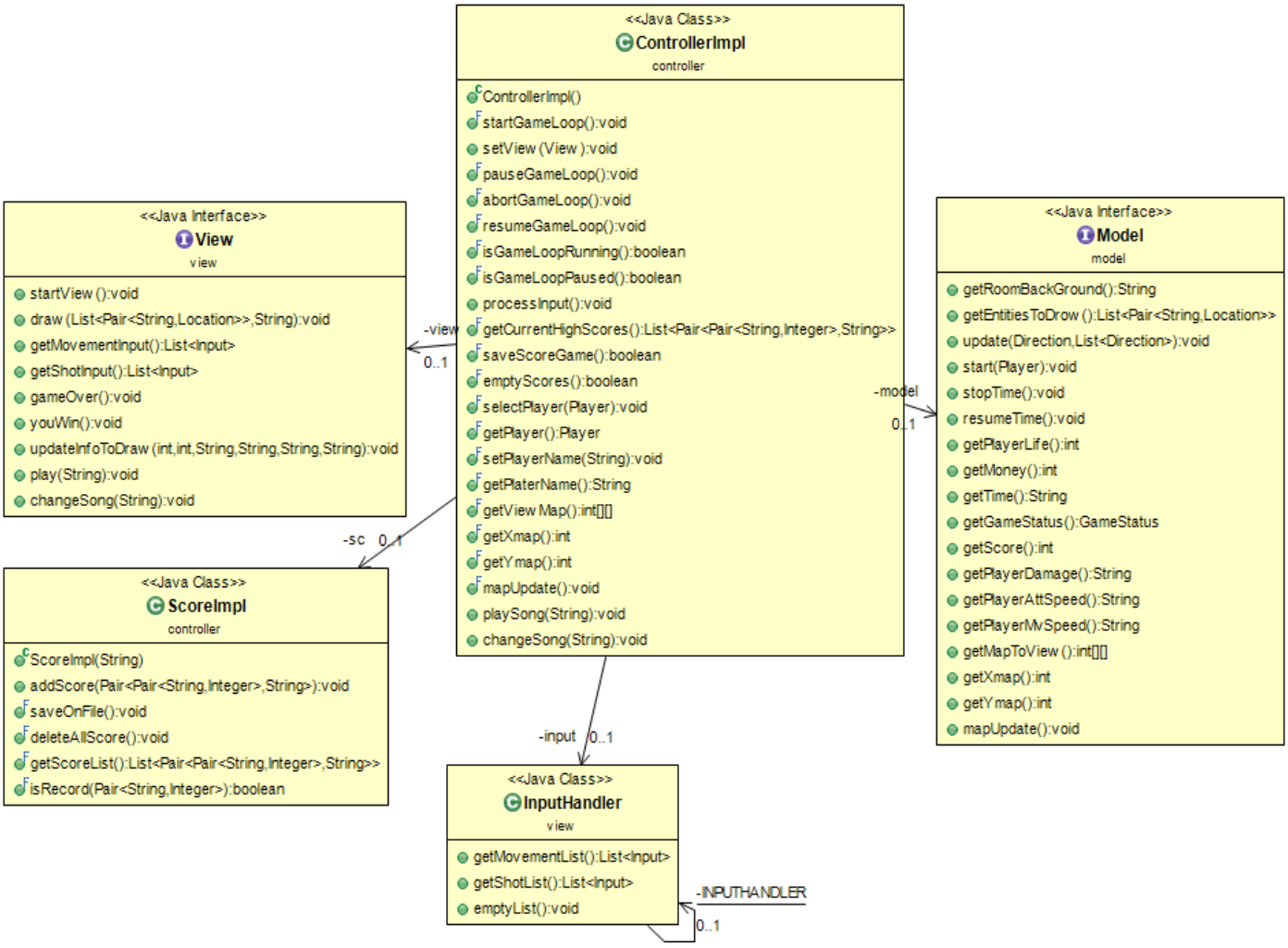


Si è presentato il problema di come gestire al meglio i boss e i player disponibili che sebbene abbiano tutti le stesse caratteristiche le presentano con valori differenti. Sia per i boss che per i personaggi per il player è stata creata un'enumerazione attraverso la quale sono reperibili velocemente i valori necessari. Stessa scelta è stata fatta per la gestione dei Power Up. In questo modo aggiungere un nuovo boss o personaggio giocabile diventa molto facile basta aggiungere un'istanza a queste enumerazioni e modificare leggermente la Factory che si occupa della creazione delle entità.

In generale si è cercato di lavorare con la consapevolezza che un videogioco ha una naturale propensione al cambiamento. Per questo si è cercato il più possibile di non far interagire direttamente le classi tra loro ma di farle comunicare attraverso le rispettive interfacce decise in fase di progettazione.

- Tommaso Ghini

CONTROLLER



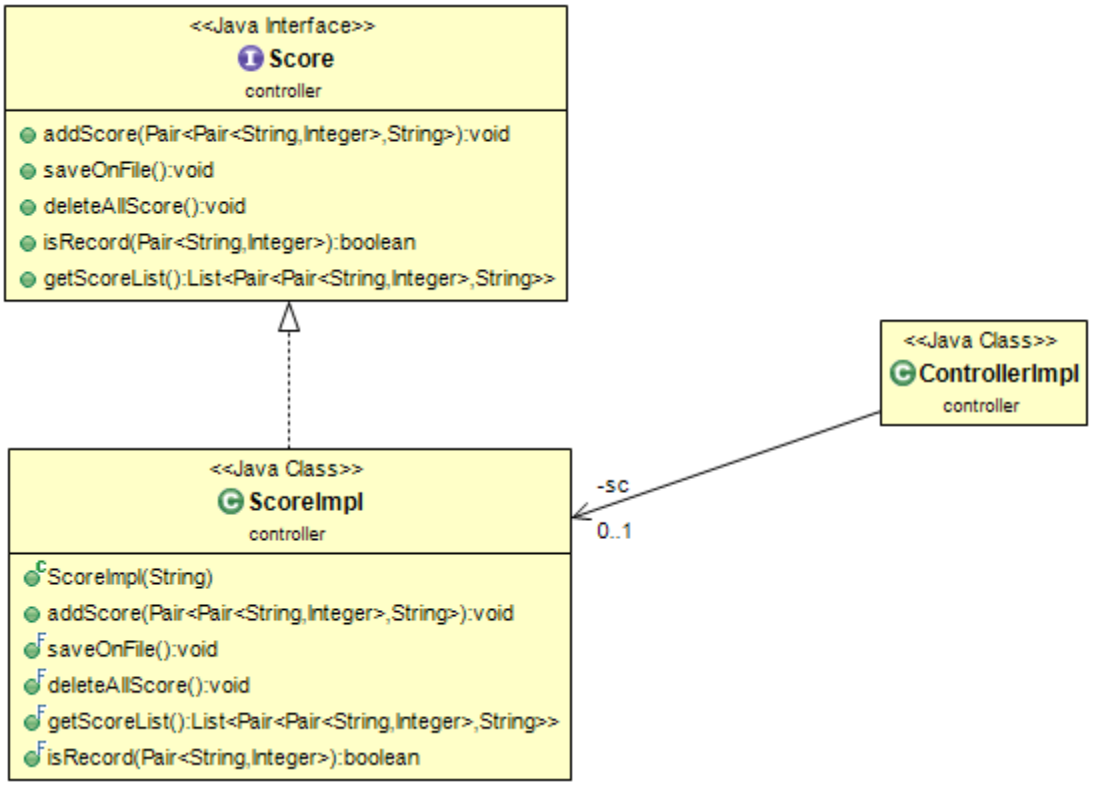
Il controller, in una progettazione con l'utilizzo di pattern MVC, è la parte di applicazione che permette di mettere in diretta comunicazione la componente di View e quella di Model, essi saranno così in grado di scambiarsi dati e di aggiornarsi quando richiesto, in questo modo sarà il controller stesso a coordinare le interazioni dell'utente con la data interfaccia.

La classe Controller dovrà interagire direttamente con le relative interfacce di View e Model che ne richiedono l'utilizzo, invece che con la vera implementazione delle due.

In questo modo sarà più facile se, in un eventuale futuro, si decida di modificare o aggiungere una nuova versione della classe esistente, in quanto agisce direttamente sull'interfaccia che verrà implementata dalla nuova classe, e questo sarà sufficiente perché tutto funzioni correttamente, facilitando cosi i futuri lavori sull'applicazione

Inoltre come vediamo nel diagramma UML, il controller presenta rapporti anche con le classi di gestione dello Score, quindi l'elaborazione della classifica dei migliori punteggi di gioco, e degli Input forniti dall'utente.

SCORE

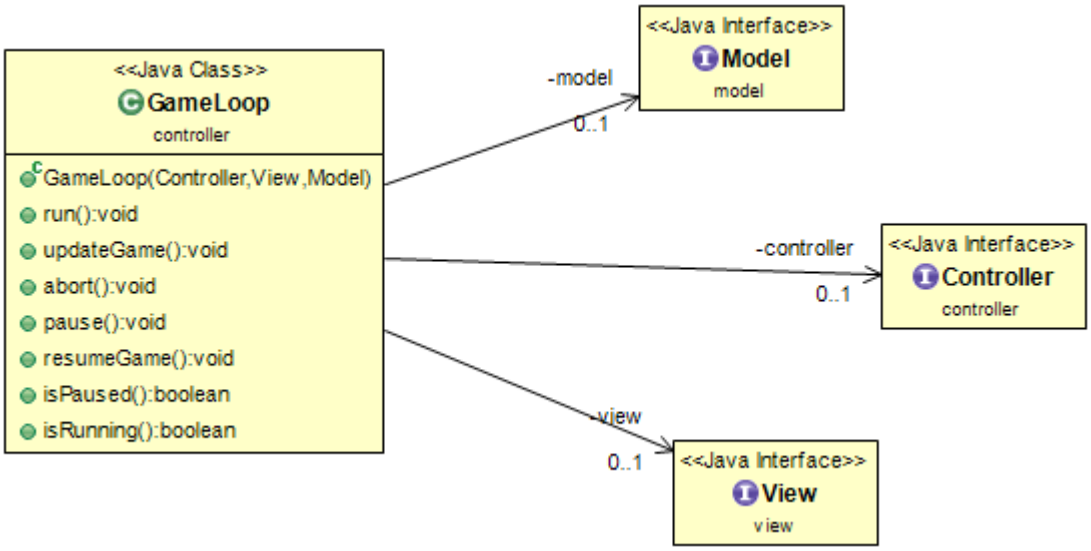


Il Controller sfruttando la classe ScoreImpl, è in grado di salvare il punteggio di gioco, che equivale al tempo impiegato per il completamento di tutte le stanze. Il controller dovrà eseguire due principali operazioni relative allo score, il salvataggio come citato in precedenza dei migliori 10 risultati e l'eliminazione di tutti i punteggi stabiliti fino a questo momento.

TIME

Il tempo di gioco, come appreso nel paragrafo precedente, è fondamentale in quanto caratterizza il punteggio della partita di un giocatore. La classe TImeImpl ci predispone alla possibilità di avviare, mettere in pausa e di riprendere il tempo di gioco, il quale verrà inizializzato ad ogni nuova partita, esso genera un thread che attraverso il metodo run sarà in grado di essere eseguito.

GAME LOOP



Il game loop, è l'elemento fondamentale del videogioco in quanto, è il ciclo "infinito" che tiene in vita il gioco, processando gli input, passando i dati aggiornati da model a view e il rendering della scena di gioco sullo schermo, cerca quindi di tenere il tempo di gioco fedele al tempo reale. Viene considerato un pattern nel libro "Game programming patterns " (di Robert Nystrom).

Per questi due motivi mi è parso opportuno citarlo come pattern.

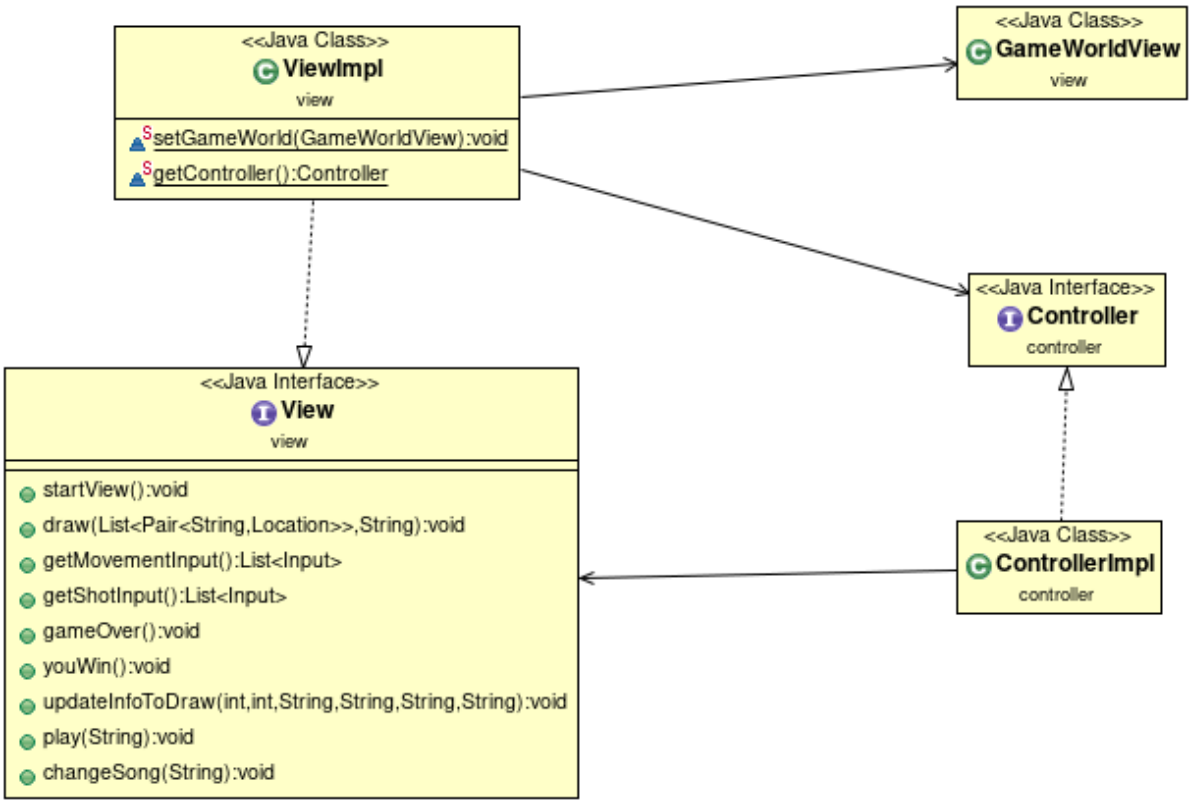
Nel nostro caso specifico, il game loop viene chiamato e fatto partire dalla view, esso alimenta il gioco fino a che non si riceve un cambio di stato dell'applicativo stesso, ad esempio quando il model scoprirà che il giocatore ha finito le vite a disposizione setterà lo stato a terminato, facendo così si uscirà dal loop. In caso contrario ogni ciclo, del metodo run, avrà il dovere di mantenere una velocità di gioco basata sul tempo reale e che sia cosi per ogni computer sul quale il nostro gioco dovrà essere eseguito, in questo modo si cerca di eliminare le differenti velocità di elaborazione di computer diversi mantenendo quella prestabilita. Inoltre il game loop ha il dovere di avvertire la view dei cambiamenti che il model comunica, sfruttando il controller, per esempio l'aggiornamento delle entità ancora in vita o la modifica di certe statistiche che la view dovrà poi rendere visibili all'utente. Il game loop pone la possibilità di essere messo in diversi tipo di stato, attraverso i quali verrà fermato, fatto ripartire o terminare. L'ultimo compito assegnatogli è la comunicazione degli input eseguiti dal giocatore, esempio i movimenti o gli spari, al model che in seguito ad essi dovrà eseguire determinati eventi.

- Lorenzo Casini

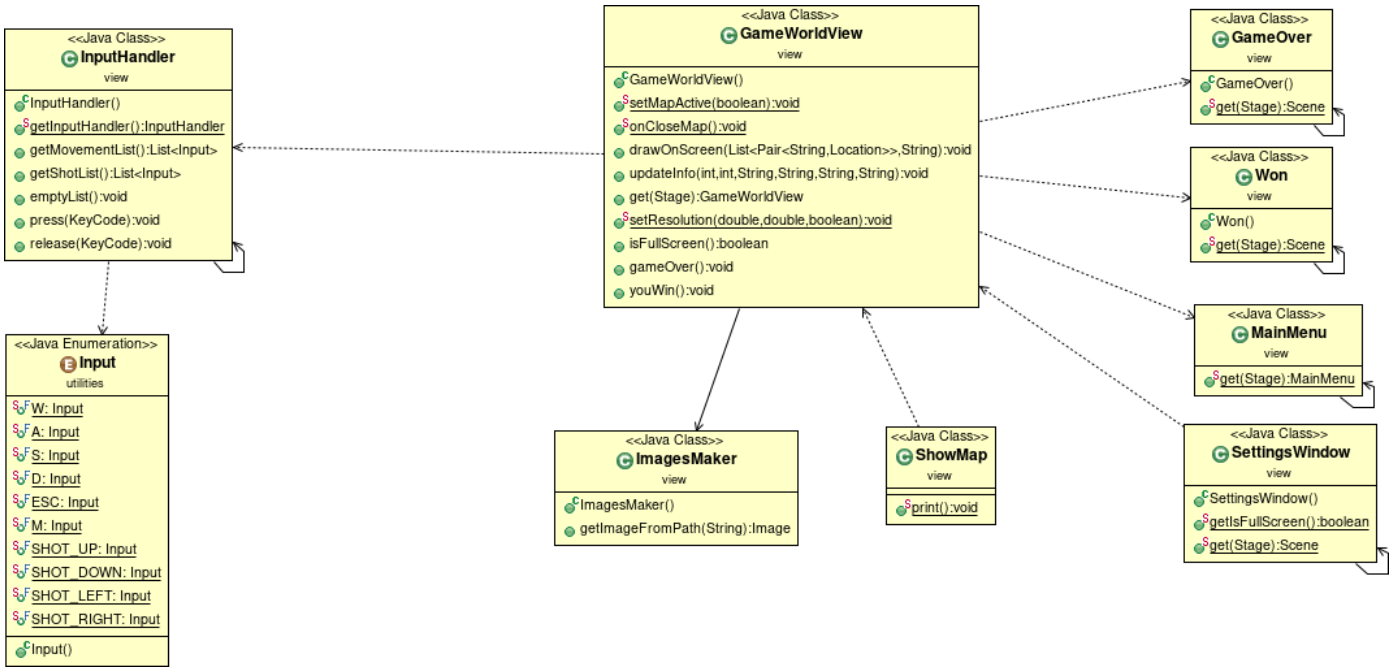
VIEW

All'interno del nostro progetto io mi sono dedicato allo sviluppo e implementazione della View , che si occupa principalmente di realizzare la grafica della nostra applicazione , della gestione degli input da parte dell'utente e di rappresentare graficamente gli oggetti creati dal Model. Per l'implementazione della View ho scelto, accordandomi anche con gli altri membri del team, di utilizzare JavaFX, la più recente libreria grafica disponibile per Java, in quanto ci sembrava utile saper lavorare con questa libreria nonostante durante l'anno accademico non l'avessimo utilizzata molto all'interno dei laboratori del corso, ma ci avrebbe permesso di rendere graficamente più "accattivante" l'applicazione, rispetto all'implementazione tramite la libreria Java Swing. Tutta l'architettura del progetto è basata su patter MVC, questo comporta che l'implementazione della View non prevede che essa in nessun modo conosca I dettagli dell'implementazione del Model, l'unico mezzo di comunicazione col Model è mediante l'interfaccia Controller. In alternativa sempre per restare nei canoni dell'MVC la View è in comunicazione col resto dell'applicazione mediante la sua interfaccia (View), la quale rende disponibili diversi metodi che per esmpio possono essere utilizzati dal Model o dal Controller per stampare a video o ricevere input da parte degli utenti.

Per interfacciarmi col Controller ho creato un campo statico all'interno della View così mediante l'utilizzo di un getter tutte le classi della View potranno avervi accesso facilmente.



La View viene lanciata dal Controller mediante il metodo startView() che lancia un'instanza di JavaFX che crea la finestra principale del gioco , chiamata MainWindow , questa classe non fa altro che creare uno (Stage) principale che non verrà chiuso fino a quando l'utente non deciderà di uscire dal gioco. Per aumentare la scalabilità della View è stato deciso di creare classi indipendenti delle singole schermate (MainMenu,BestScore,Settings,GameOver ecc…) e renderle fruibili attraverso un getter , facendo cosi ci bastera chiamare dallo (Stage) principale solamente la (Scene) che vogliamo utilizzare in quel dato momento , questo ci permette in un futuro e come è gia successo durante lo sviluppo del progetto , se decidiamo di creare una nuova schermata esmepio (ShowCredits) ci bastera estendere (Scene) , creare I componenti grafici necessari al suo interno , creare un getter della scena , a questo punto possiamo richiamarla tramite il getter da un'altra classe esempio (MainMenu) aggiungendo un nuovo pulsante nel nostro menù iniziale.



Ogni volta che il giocatore si trova nella schermata di MainMenu , e decide di iniziare una nuova partita, la scena viene spostata nella classe SelectCharacter, nel momento in cui l'utente sceglie il personaggio viene avviata la classe principale che gestisce la schermata di gioco , ovvero, GameWoldView.

Questa classe presenta diversi metodi per la creazione della schermata di gioco, tra I più importanti vediamo (setResolution, drawOnScreen, updateInfo), si occupano rispettivamnete di :

- ridimensionare la finestra di gioco secondo quando specificato dall'utente nel menù iniziale [schermata Settings]
- rappresentare tutte le entità presenti nella schermata di gioco
- aggiornare le statistiche presenti nella status-bar

Altra funzione fondamentale di questa classe è la gestione degli input da parte dell'utente, la classe GameWolrdView cattura gli input e li manda alla classe InputHandler per essere processati

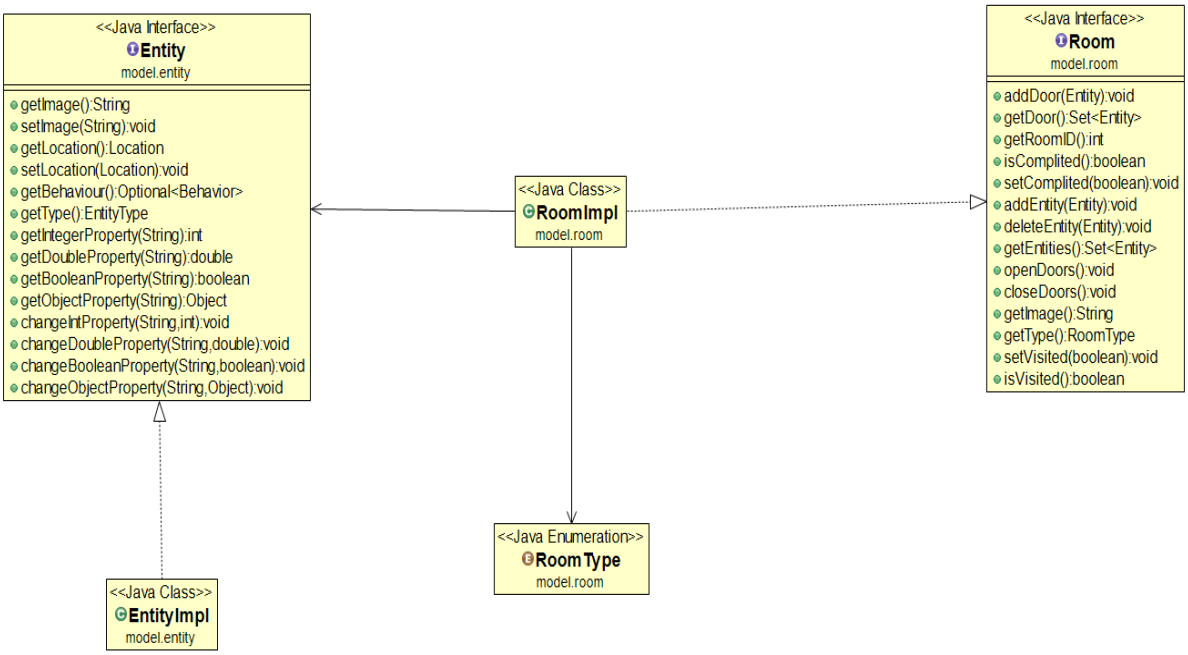
tramite 2 metodi (getMovementList,getShotList), così facendo , abbiamo diversificato I tasti relativi al movimento e quelli dello sparo rappresentati tramite una Enum , tutto questo ha comportato una maggiore fluidità di gioco. Questi input verranno richiesti ed elaborati dal Controller, che restituirà ad ogni frame una lista di coordinate da rappresentare nella shermata di gioco per creare il movimento del personaggio e dei sui spari, stesso discorso vale anche per I nemici ma il loro movimento non è gestito dall'utente ma da una "IA". Per l'implementazione della classe InputHandler è sato scelto di utilizzare il Pattern Singleton, in quanto dovevamo essere sicuri che ci fosse solo un'istanza di questa classe per la gestione dell'input. Oltre alla classe InputHandler , il Pattern Singleton è stato anche utilizzato per la creazione delle schermate di vittoria o sconfitta (Won,GameOver) , ovviamente sarà il GameLoop a controllare l'avanzamento della partita e decretare appunto la vittoria o la sconfitta in tal caso il Controller richiama la View.(youWin/gameOver) e la View si occuperà di cambiare scena e presentare la schermata corretta al giocatore che in caso di vittoria dovrà provvedere anche la salvataggio del punteggio del giocatore.

Per quanto rigurada l'aspetto grafiche un'altra cosa che ci ha spinto ad utilizzare JavaFX era la possibilità di usare dei fogli di stile(style.css) , che hanno semplificato e migliorato notevolmente la personalizzazione dei vari componenti delle schermate . Altro punto cruciale del gioco era la gestione delle dimensioni , per quanto rigurada le schermate del menù di gioco si è deciso di utilizzare una dimensione prefissata di 800x600px , ma è stata messa la possibilità , entrando nella sezione Settings, di scegliere tra 3 risoluzioni rispettivamente (Low,Mid,High)[1024x576,1280x720,1920x1080], il programma si prende carico di controllare la dimensione dello schermo dell'utente e decidere se effetturare il cambio di risoluzione e di conseguenza adattare anche tutti gli oggetti all'interno della shermata di gioco in base alla risoluzione selezionata.

- **ANIS LICO**

STANZE E PORTE

La mia parte di progetto consiste nell'implementare il mondo di gioco quindi la mappa di gioco logica e le stanze.

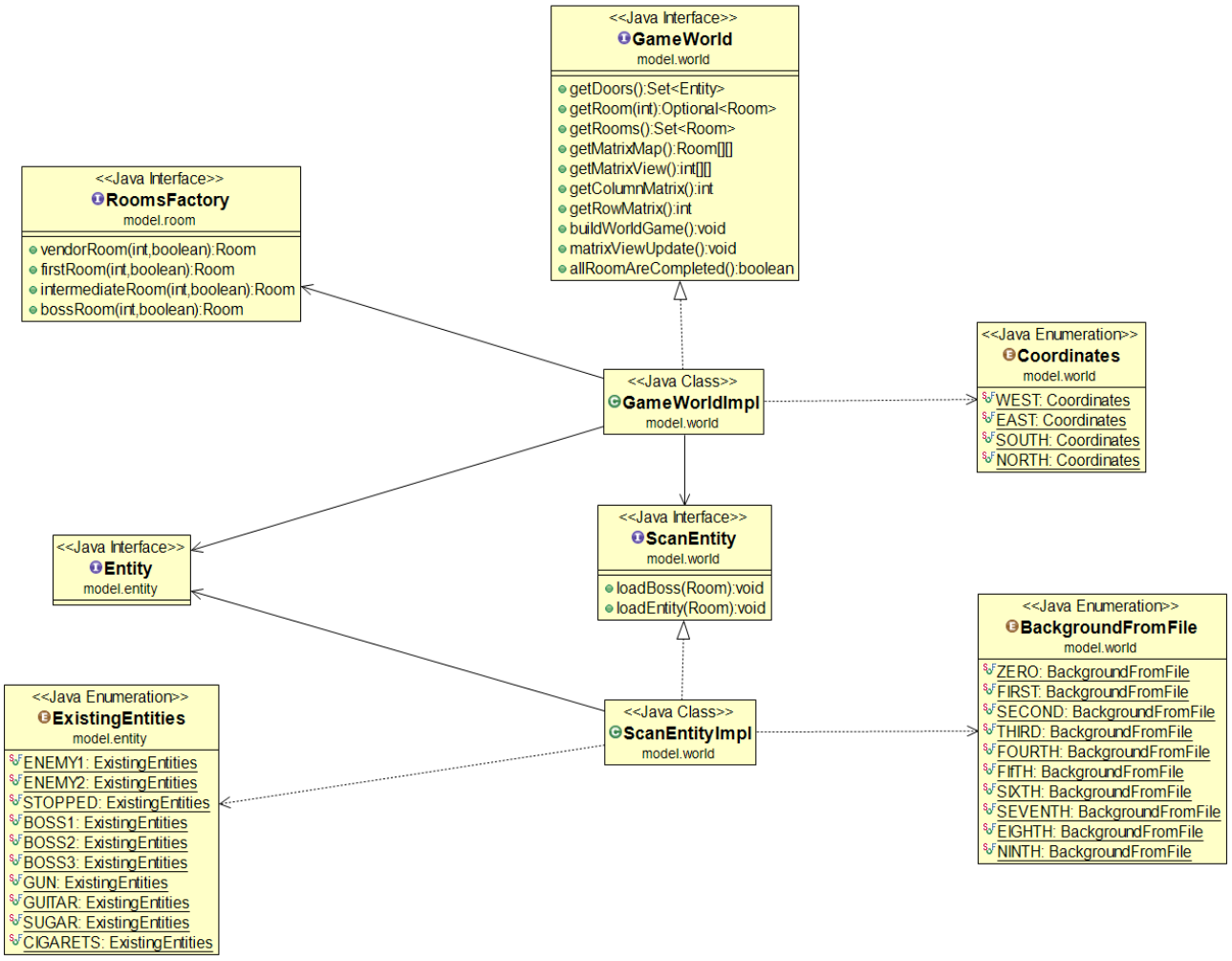


I contenitori delle entità sono le stanze di gioco, vengono implementate partendo da un'interfaccia Room che definisce i metodi che ogni stanza deve avere. I vari tipi di stanze che compaiono nel mondo di gioco vengono definite nella enumerazione RoomType. La differenziazione delle stanze definisce il numero di porte nelle stanze e il tipo di entità che compaiono in esse, tutto ciò è stato possibile anche grazie all'uso del pattern builder il quale semplifica la procedura di creazione di queste ultime. L' interfaccia EntityFactory permette di creare i quattro tipi di stanze FIRST, VENDOR, INTERMEDIATE, BOSS. In questo modo sarà molto semplice in futuro creare un nuovo tipo di stanza visto che basterà implementare un nuovo metodo nell' EntityFactory.

L’interfaccia Room viene caratterizzate da due metodi di grande rilievo isVisited e isComplited, il primo indica se il player è passato dalla stanza e permette di renderla visibile nel momento in cui il giocatore vorrà interfacciarsi con la mappa di gioco, il secondo invece indica l’avanzamento nel gioco ricordando infatti che il gioco verrà completato quando tutte le stanze restituiranno al metodo isComplited il valore true.

All’interno delle stanze vengono predisposte le entità di gioco tra cui i nemici e porte, si è scelto di dichiarare le porte come entità confrontandomi con Simone Del Gatto, quest’ultime svolgono un ruolo fondamentale nella logica del gioco in quanto permettono lo spostamento del player nei vari percorsi del sistema predisponendo un vero e proprio collegamento tra due stanze. Le porte vengono definite come entità ma hanno un comportamento differente dalle altre, questo viene descritto nell’enumerazione Coordinates dove vengono gestite le informazioni sulla dimensione, l’area e le immagini da settare in base al posizionamento e lo stato nella stanza. Le porte hanno una proprietà definita nell’enumerazione DoorStatus, che definisce il comportamento del player quando collide con essa. Nella classe Room compaiono due metodi, openDoors e closeDoors che manipolano il comportamento delle porte.

GAMEWORLD



L'Interfaccia GameWorld è ciò che crea e definisce il vero è proprio dominio del mondo di gioco con cui il giocatore si interfaccia.

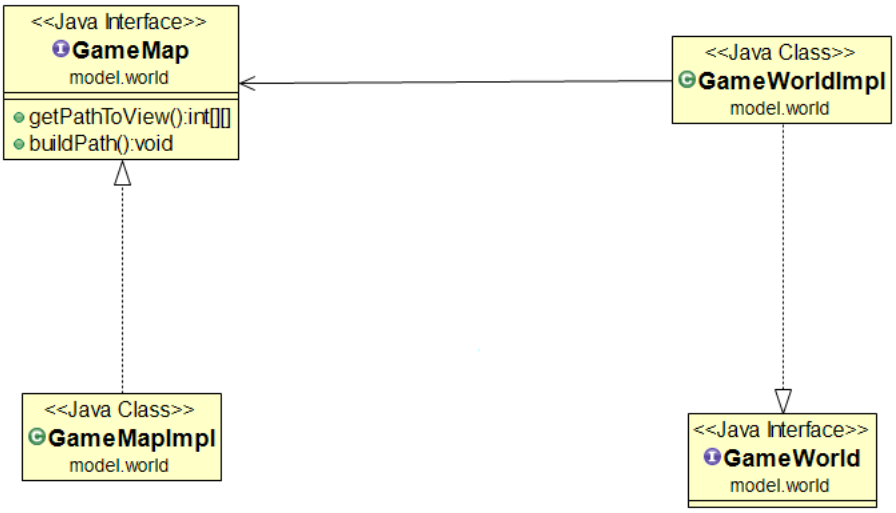
GameWorld viene implementato dalla classe GameWorldImpl che nel suo costruttore prende in ingresso il player scelto dal giocatore e un EntityFactory per la creazione delle entità di gioco, in più inizializza il RoomFactory che come descritto in precedenza è un'implementazione del pattern Factory usata per la creazione di stanze. Una volta creata la classe GameWorld va richiamato il suo metodo buildWorldGame che inizializza il procedimanto di creazione del mondo di gioco. BuildWorldGame richiama il metodo initializeMapBuilding che crea una matrice di Room e la popola con le prime stanze, queste sono statiche cioè ogni volta che si inizierà una nuova partita di gioco saranno predisposte sempre nella stessa posizione. Durante l'esecuzione del metodo ogni nuova stanza creata verrà aggiunta all'iniseme di room e ogni collegamento tra stanze verrà inserito nell'inisieme di Entity di tipo Door.

InizializeMapBuilding richiama il metodo completePath per tre volte per creare i tre percorsi che dispongono le stanze in modo casuale ma continuo. Come deciso in fase di progettazione i tre percorsi non si intersecano e questo è stato possibile grazie all'implementazione di vari check.

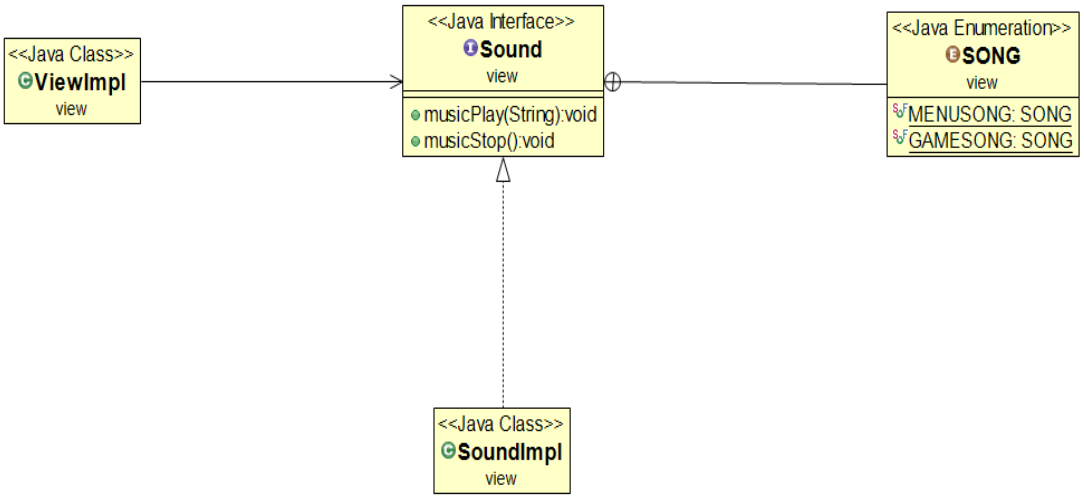
Ho creato diversi file dove vengono definite le configurazioni possibili delle varie stanze cioè il posizionamento delle entità in esse. L'enumerazione BackgroundFromFile contiene i vari file di configurazione e mette a disposizione il metodo getRandomPath per ottenere un file in modo randomico. Per aggiungere nuovi file di configurazione in futuro basterà inserire un nuovo campo nell'enumerazione.

Ho sviluppato l'interfaccia ScanEntity implementata dalla classe ScanEntityImpl che rende possibile il popolamento delle stanze, essa prende in ingresso nel costruttore il player e un EntityFactory. Quando il metodo buildWorldGame chiama populateNormalRoom e populateBossRoom della classe GameWorld passa a un oggetto di tipo ScanEntity una stanza e questo la riempirà prendendo in modo casuale uno dei diversi file usando il metodo statico getRandomPath. ScanEntity legge il file nello stesso modo in cui si legge una matrice e quando incontra gli specifici caratteri presenti nell'enumerazione ExistingEntities aggiunge l'entità rappresentata da quello specifico carattere e lo inserisce nella stanza nella locazione giusta.

GAMEMAP E SOUND



Ho incapsulato la necessità di vedere a video la mappa di gioco nella classe GameMapImpl anche qui ho cercato di far comunicare le classi però solo con i metodi dell'interfaccia da essa implementata GameMap. La mappa di gioco viene gestita come una matrice, il giocatore nel momento in cui si vorrà interfacciare con la mappa vedrà solo le stanze visitate e la stanza in cui il player si trova, la currentRoom. Ogni volta che si premerà il tasto M della tastiera il GameWorld richiamerà il metodo update del GameMap per aggiornare la mappa e quest'ultimo li ritornerà la mappa aggiornata.



Ho creato la classe SoundImpl per gestire le musiche di gioco. Questa classe non fa altro che creare un oggetto di tipo Clip e passandogli un file di tipo wav verrà riprodotto in loop, effettuando vari check per verificare la corretta esecuzione delle musiche. L'interfaccia implementata da SoundImpl è Sound quest'ultima dà la possibilità di far partire e cambiare la musica da riprodurre. E' stato il mio compagno Lorenzo Casini a gestire i richiami dell'interfaccia Sound. La musica di gioco viene gestita direttamente dalla View.

Sviluppo

3.1 Testing automatizzato

Abbiamo sviluppato diversi test avvalendoci della libreria JUnit per controllare il funzionamento di alcune parti: la creazione di entità, il comportamento del Player (PlayerBehavior), dei nemici (StalkerEnemyBehavior) e dei proiettili e un test per le collisioni, un test per il funzionamento del Model, un test per verificare il corretto popolamento di alcune stanze principali e un test perché queste siano effettivamente create. Altri test sono stati eseguiti in modalità manuale in ambiente Windows 10, Arch Linux e macOs.

3.2 Metodologia di Lavoro

Il gruppo nei primi momenti si è riunito per la definizione delle principali interfacce e per suddividere il carico di lavoro il più equamente possibile e cercano di rispettare le preferenze di ognuno. Nonostante sia stato speso diverso tempo per la fase di progettazione, l'inesperienza ha reso comunque necessario la rivisitazione di quanto era stato fatto senza però operare grossi stravolgimenti.

Visti gli impegni diversi che hanno portato a sviluppare le varie parti dell'applicazione comunque in tempi diversi si è deciso di ritrovarsi a per mettere insieme definitivamente tutte le parti e per occuparsi della parte "artistica" del lavoro (come sono effettivamente popolate le stanze, i parametri che definiscono i vari personaggi ecc.).

Nel caso qualche membro del gruppo si trovasse in difficoltà è stato possibile aiutarlo dato che in fase di progettazione sono state discusse tutte le parti del progetto insieme e quindi anche se ognuno si è focalizzato poi su una sua specifica parte chiunque era a conoscenza del funzionamento generale dell'applicazione.

Per l'uso del DVCS si è deciso di operare in questa maniera: ognuno aveva una copia in locale del master e un ramo di sviluppo proprio. Prima di operare una push sul master remoto venivano fatte le pull dei cambiamenti operati dagli altri membri del gruppo e veniva fatto il merge con il ramo di sviluppo personale controllando che il gioco effettivamente compilasse e funzionasse. Corretti eventuali conflitti si procedeva condividendo le proprie modifiche con una push sul master remoto.

- Del Gatto

  Ho realizzato tutte le classi e le interfacce del package model, tranne quelle relative alla modellazione del tempo (Time e TimeImpl) e della classe Application, e del package model.entity tranne (DoorStatus, EntityProperties) e ho contribuito alla realizzazione della View con il metodo drowOnScreen(). La creazione e la manipolazione delle entità "porte" invece ha richiesto la collaborazione di Anis Lico vista la stretta correlazione con la parte da lui sviluppata. Inoltre mi sono occupato di modellare alcune immagini presenti nel gioco (immagini relative al nemico mosca, immagini relative al background della stanza, immagini relative al nemico spiritello) che sono state però effettivamente reperite da Lorenzo Casini. Un

aiuto mi è stato dato dagli altri per la creazione dei power up (in particolare da Tommaso Ghini) e per la scelta dei settaggi dei vari parametri che definiscono in comportamento dei personaggi in modo da rendere il gameplay il più piacevole possibile. Oltre a quanto riportato la cooperazione con gli altri è stata necessaria per la definizione di alcune interfacce principali in particolare quella del Model. Come tutti ho dato il mio contributo nella progettazione del gioco sia dal punto di vista architetturale sia dal punto di vista creativo.

- Tommaso Ghini

Per la realizzazione di questo progetto la mia parte è stata la realizzazione di tutte le classi all'interno del package controller e della classe e interfaccia del time nel package model.

- Lorenzo Casini

Personalmente mi sono occupato della creazione di tutte le classi all'interno del package view , tranne alcune come :

- Sound - sviluppata insime ad Anis Lico
- InputHandler - sviluppata insieme a Tommaso Ghini
- GameWolrdView - Simone del Gatto ha sviluppato il metodo drawOnScreen

Per quanto riguarda la realizzazione della grafica ci siamo confrontati più e più volte cambiando , spesso font, colori e suoni , fino ad arrivare ad una versione finale che potesse soddisfare I gusti di tutti.

- Anis Lico

Ho realizzato tutte le classi presenti nei sottopackage model.room e model.world l'enumerazione ExistingEntities nel package model.entities e le classi Sound e SoundImpl nel package View. Mi sono confrontato Lorenzo Casini per lo sviluppo dei suoni. Il gruppo ha scelto insieme la musica di gioco io ho gestito e tagliato i file wav rendendogli adeguati al nostro gioco. Nella prima parte di analisi mi sono confrontato con Simone Del Gatto per decidere in che modo le nostre parti di progetto avrebbero dovuto comunicare e come avrei dovuto implementare le porte perché fossero assimilabili dalla sua parte di progetto.

3.3 Note di sviluppo

- Del Gatto

Optional: ho utilizzato la classe optional nello sviluppo delle entità in quanto alcune potrebbero non possedere una componente Behavior.

Generici: utilizzati nella classe Pair.

Stream e Lambda: ho cercato di sviluppare il codice in modo che si potessero utilizzare il più possibile queste strutture sintattiche. Nel model in particolare nel metodo update() ne faccio largo utilizzo in quanto mi trovo spesso a utilizzare delle collezioni di entità. Lo sviluppo di queste ultime inoltre facilità l'utilizzo di queste strutture

Spunti, codice riutilizzato, ispirazioni: poiché non ho mai sviluppato un videogioco mi è sembrato giusto consultare i progetti resi disponibili. Da questi ho preso due classi in particolare: Location e Direction. La classe di utilità Pair l'ho ripresa dai testi d'esame consegnati per la preparazione della prova scritta. Inoltre la gestione delle entità l'ho sviluppata sulla falsa riga della libreria FXGL che non abbiamo potuto utilizzare in quanto non riusciva ad adattarsi al pattern architetturale proposto.

- Tommaso Ghini

Optional: Per quanto riguarda la mia parte svolta nel controller, si è utilizzato un campo Optional relativo al game loop, dato che il controller deve utilizzare il game loop, deve essere certo che esso sia presente in quanto il game loop deve essere creato in un secondo momento rispetto al controller.

Lambda e Stream: Sempre nel controller, nel metodo processInput è stata utilizzata una semplice lambda per trasformare i dati nella giusta direzione relativa agli spari del giocatore nell'apposita lista che verrà poi utilizzata dal model per eseguire le corrette azioni.

Spunti, codice riutilizzato, ispirazioni: Lo sviluppo della mia parte è stata supportata dall'utilizzo di diversi materiali, come i vecchi progetti degli anni precedenti, l'utilizzo di pagine online e dal libro "Games Programming Patterns" di Robert Nystrom.

- Lorenzo Casini

Stream e Lambda: Quando è stato possibile nella realizzazione delle mie classi ho utilizzato le lambda expressions.

Librerie, codice riutilizzato, strumenti extra: Per quanto rigurada l'utilizzo di librerie ho utilizzato JavaFX e visto che nel corso non era stata molto approfondita, per imparare ad utilizzarla al meglio mi sono basato sui progetti passati che ne avevano fatto una buona implementazione. Nel progetto è presente un foglio di stile (style.css), ovviamente tutto il contenuto è stato adattato per il nostro gioco ma inizialmente per capire il funzionamento dei vari componenti mi sono affidato a https://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html e vari esempi di file.

- Anis Lico

Stream e Lambda: dove possibile ho utilizzato lambda experession e stream soprattutto per manipolare le collezioni presenti nelle classi

Optional: ho fatto uso degli Optional per le ricerche delle classi

Algoritmi: ho sviluppato un algoritmo per la creazione di percorsi che non si intersecano nel metodo privato buildPath() nella classe GameWorldImpl.

Commenti Finali

4.1 Autovalutazione e lavori futuri

- Del Gatto

Il progetto nel complesso mi soddisfa soprattutto la parte della gestione delle entità che può essere discutibile ma secondo me ha un grande pregio: la riutilizzabilità. Le entità infatti non sono state concepite in maniera statica ma veramente si è cercato di dare una struttura generale che possa essere a mio avviso specializzabile in qualsiasi contesto e nella maggior parte dei videogiochi 2D. Poi può essere discutibile, ad esempio, la scelta che le proprietà delle varie entità siano accessibili attraverso l'uso di una stringa anziché con un metodo specifico come solitamente si fa ma personalmente non mi crea troppo disturbo. Una possibile miglioria che però non cambia in se il meccanismo da me elaborato è utilizzare un'enumerazione in cui vengano conservate tutte le proprietà utilizzate nel gioco e utilizzare quelle anziché delle stringhe.

Il codice per come è stato organizzato risulta abbastanza malleabile e lascia spazio a correzioni e modifiche (questo giudizio lo riporto perché così ho sperimentato soprattutto nell'ultima fase di lavoro in cui ho dovuto operare qualche cambiamento su richiesta degli altri componenti del gruppo o in fase di debug). Se infuturo, ad esempio, vorrò creare un nuovo nemico mi basterà di fatto creare una nuova classe che estenda Behavior e aggiungere o modificare un metodo a EntityFactory, se voglio modificare il valore delle proprietà di un personaggio mi basterà modificare la relativa enumerazione e così via. Risultano però tutti piccoli cambiamenti.

Questo progetto mi ha aiutato anche e soprattutto a dovermi confrontare con un lavoro molto più ampio di quelli con cui sono solito a scontrarmi e ho veramente dovuto prendere atto di come sia importante seguire buone pratiche di programmazione e fare una buona progettazione iniziale.

Non so se in futuro mi dedicherò ancora a questo progetto, forse lo farò per pura passione personale. Sicuramente alcune delle cose che si potrebbero fare è migliorare l'intelligenza artificiale dei nemici e l'aggiunta di vari suoni durante gli spari, o anche volendo, di animazioni nel caso di collisioni.

- Tommaso Ghini

Parlando della mia esperienza maturata da questo progetto, mi sono reso conto dell'importanza di una buona progettazione iniziale, dello studio della giusta divisione di compiti e dell'immaginazione di come si dovrà andare a scrivere codice per essere in grado di collegare le varie parti, in quanto ci ha dato la possibilità di lavorare in modo coordinato e di risolvere problemi che sarebbero potuti sorgere se la progettazione non fosse stata tale, mi ritengo quindi molto soddisfatto del lavoro che è stato svolto come gruppo, a livello personale ho trovato molto entusiasmate il poter applicare la propria fantasia in un progetto come questo e di scoprire quanto sia una vera "palestra per programmatori" lo sviluppo di un

videogioco, le abilità imparate in questo corso assieme ad una adeguata ricerca su alcuni concetti si è rivelata utile per imparare a programmare questa tipologia di applicazioni.

La mia più grande difficoltà, inizialmente, è stata il capire veramente il funzionamento del Controller e del suo efficace utilizzo, ma maggiormente la creazione e l'utilizzo del game loop si era rivelata un'incognita assai più complessa, per questo è stato un duro lavoro il suo apprendimento relativo alla sua importanza in un videogioco e alla sua implementazione. È stato molto interessante la ricerca svolta su libri e pagine online per capire il funzionamento del game loop, tutte le varie proposte di tale implementazione, caratterizzate da vantaggi e svantaggi rispetto ad altre. Tutto sommato è stata una messa alla prova delle abilità imparate in questo corso molto importante per la mia crescita ma anche di ampliamento della mia visione su un mondo per me nuovo dell' informatica, mi reputo quindi molto contento di questo progetto che se anche con molta fatica, ha regalato grandi soddisfazioni a livello personale e di gruppo, facendomi capire che le abilità imparate in questo corso di studi sono "reali", e che con un po' di duro lavoro è possibile creare le proprie idee. Sono quindi molto convinto dell'importanza che ha avuto, per la nostra carriera informatica, l'implementazione di questo progetto.

- Lorenzo Casini

Alla fine di tutto mi ritengo molto soddisfatto ma alquanto sorpreso anche dell'ottimo lavoro svolto dagli altri membri del gruppo in quanto ,non mi sarei mai aspettato di riuscire a realizzare questo gioco cosi come è oggi , quando lo immaginavamo duranto il corso di OOP non era minimamente paragonabile(dato anche la poco esperienza e I tantissimi interrogativi su come poter realizzare le varie componenti). C'è stato a mio parere un buon lavoro di gruppo, anche Git ha contribuito al mantenimento dell'ambiente di lavoro e più e più volte ha salvato l'intero progetto da errori.Nel futuro anche in accordo comune con gli altri membri pensiamo di mantenere il gioco solo per passione ed interesse personale in quanto ci ha coinvolto molto nello sviluppo e sarebbe un peccato farlo cadere nel dimenticatoio.

- Anis Lico

Sono molto soddisfatto del lavoro svolto sia mio che dei miei compagni, tutto ciò è stato possibile grazie a una buona analisi e una buona progettazione inziale che ha portato a una divisione dei compiti adeguata ed equilibrata. All'inizio ho avuto qualche difficolta per quanto riguarda la metodologia di creazione della mappa logica ma dopo varie riflessioni ho deciso di utilizzare una classica matrice per rappresentarla. La buona progettazione mi ha permesso di poter effettuare varie modifiche alle classi senza stravolgere il codice già scritto.

Per quanto riguarda la mia parte, ritengo che in relazione al tempo concesso, di aver effettuato un lavoro completo portando a termine gli obiettivi a me assegnati, ovviamente in molti punti è possibile effettuare miglioramenti.

Un esempio di ottimizzazione che può essere apportata è l'implementazione della mappa logica tramite un grafo e non una matrice, tutto ciò comporterebbe uno studio attento dei grafi e una successiva implementazione di un algoritmo per il controllo dei percorsi intersecati.

Una successiva feature del gioco potrebbe posizionare le entità nelle stanze in modo randomico senza la lettura da file di entità già preimpostate e inoltre potrebbe aggiungere i suoni di gioco che non sono riuscito ad implementare in questa versione per mancanza di ore. In fondo a tutto il lavoro svolto mi sento soddisfatto e orgoglioso di essere riuscito a portare a termine tutto ciò insieme al mio gruppo.

Guida per l’utente:

Tasti di movimento:

W: movimento in direzione Nord.

S: movimento in direzione Sud.

A: movimento in direzione Ovest.

D: movimento in direzione Est.

M: visualizzare la mappa.

P: tasto per la pausa.

Esc: tasto per l’uscita.

Backspace: ritorno al menu.

Combinando questi tasti è possibile muoversi nelle varie direzioni intermedie.

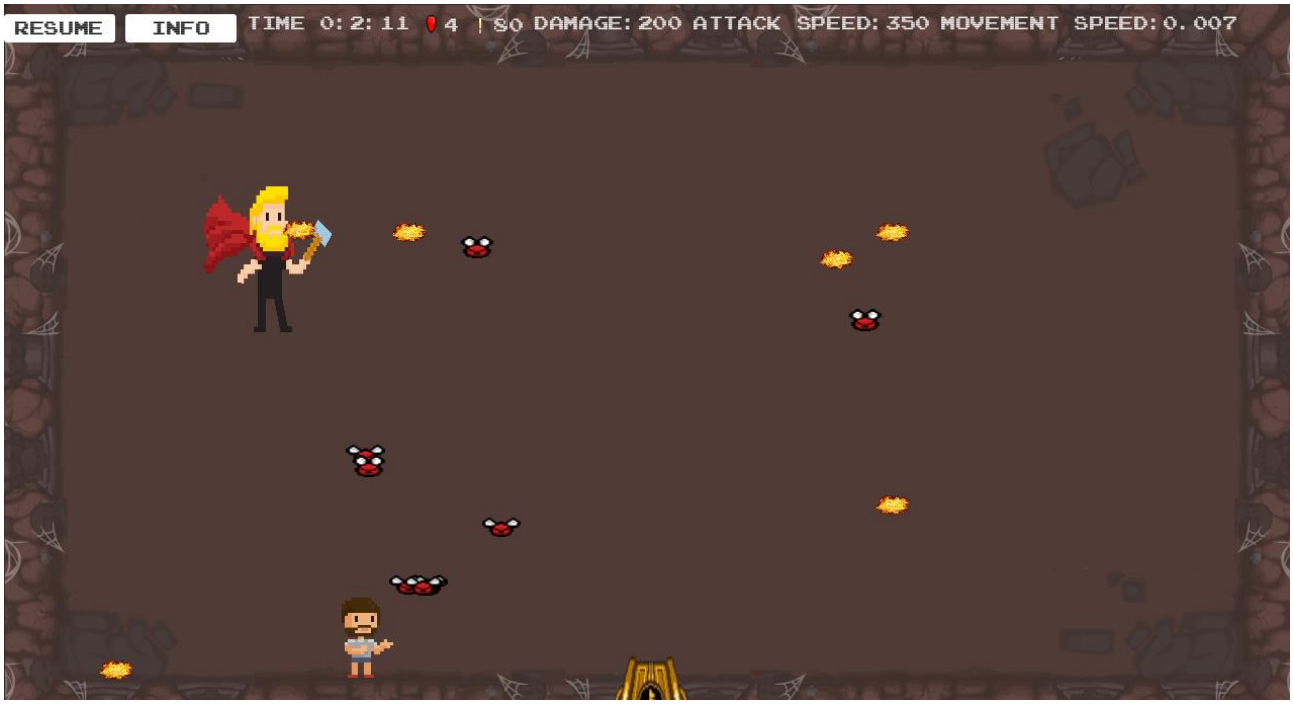
← Freccia Sx: sparo a sinistra.

→ Freccia Dx: sparo a destra.

↑ Freccia in Su: sparo in alto.

↓ Freccia in Giù: sparo in basso.

Informazioni sulla schermata di gioco:



TIME: tempo di gioco.

Cuore: vita rimasta.

Moneta: soldi a disposizione.

DAMAGE: danno di attacco del personaggio.

ATTACK SPEED: numero di millisecondi minimo che possono intercorrere tra uno sparo e l'altro del giocatore.

MOVEMENT SPEED: spazio percorso con un passo dal giocatore rispetto la totalità della grandezza della stanza.