**Progetto per il corso di OOP**
**A.A 2019/2020**

# Relazione per "Just Another Rogue-Like Game"

**Lucia Fabbri**
**Gian Luca Nediani**
**Marco Ragazzini**
**Federico Pirazzoli**

# Indice

## 1, Analisi



## 2, Design



## 3, Sviluppo



## 4, Commenti Finali



## Appendice

# Capitolo 1, Analisi

## 1.1 Requisiti

Il gruppo si pone come obiettivo quello di realizzare un gioco Rouge-like (https://it.wikipedia.org/wiki/Roguelike) vecchio stile, chiamato *Just Another Rouge-like Game (J.A.R.G.).*
Il gioco consiste nel prendere il controllo di un personaggio che deve affrontare una sequenza di livelli, formati da stanze, che contengono nemici, ostacoli, oggetti e potenziamenti con cui il giocatore può interagire.
La difficoltà dei livelli è progressiva, il livello finale prevede una battaglia contro un Boss.

Requisiti Funzionali:

- Creazione di un mondo composto da vari livelli, ognuno dei quali si differenzia dai precedenti nei quali vi possono essere elementi con i quali interagire.
- Ogni stanza di un livello deve essere esplorabile dal giocatore, il quale può muovere il protagonista nelle quattro direzioni cardinali.
- Realizzazione di varie tipologie di nemici e del Boss finale.
- Realizzazione di una piccola interfaccia grafica per il giocatore dove vengono descritte le varie caratteristiche del personaggio.

Requisiti non Funzionali:

- Realizzazione di un Boss a fine di ogni livello.
- Generazione casuale dei livelli di gioco
- Utilizzo di una libreria per il suono per generare effetti audio.
- Gestione avanzata del movimento e dell'interazione del personaggio con l'ambiente

## 1.2 Analisi e modello del dominio

Il giocatore verrà teletrasportato all'inizio di ogni livello e deve essere in grado di accedere alle stanze connesse a quella attuale.
Ciascuna stanza è composta da nemici, ostacoli ed oggetti vari che possono aiutare o arrecare danno al giocatore. In ogni livello vi sarà una scala presente in una stanza, che porterà il giocatore al livello successivo. Una volta arrivati all'ultimo livello non vi sarà una scala ma un Boss da sconfiggere per terminare con successo il gioco.
La difficoltà primaria sarà quella di far interagire i giocatori con i nemici e con l'ambiente, in maniera fluida e responsiva.

Gli elementi costituitivi dell'architettura sono rappresentati nel seguente schema UML:

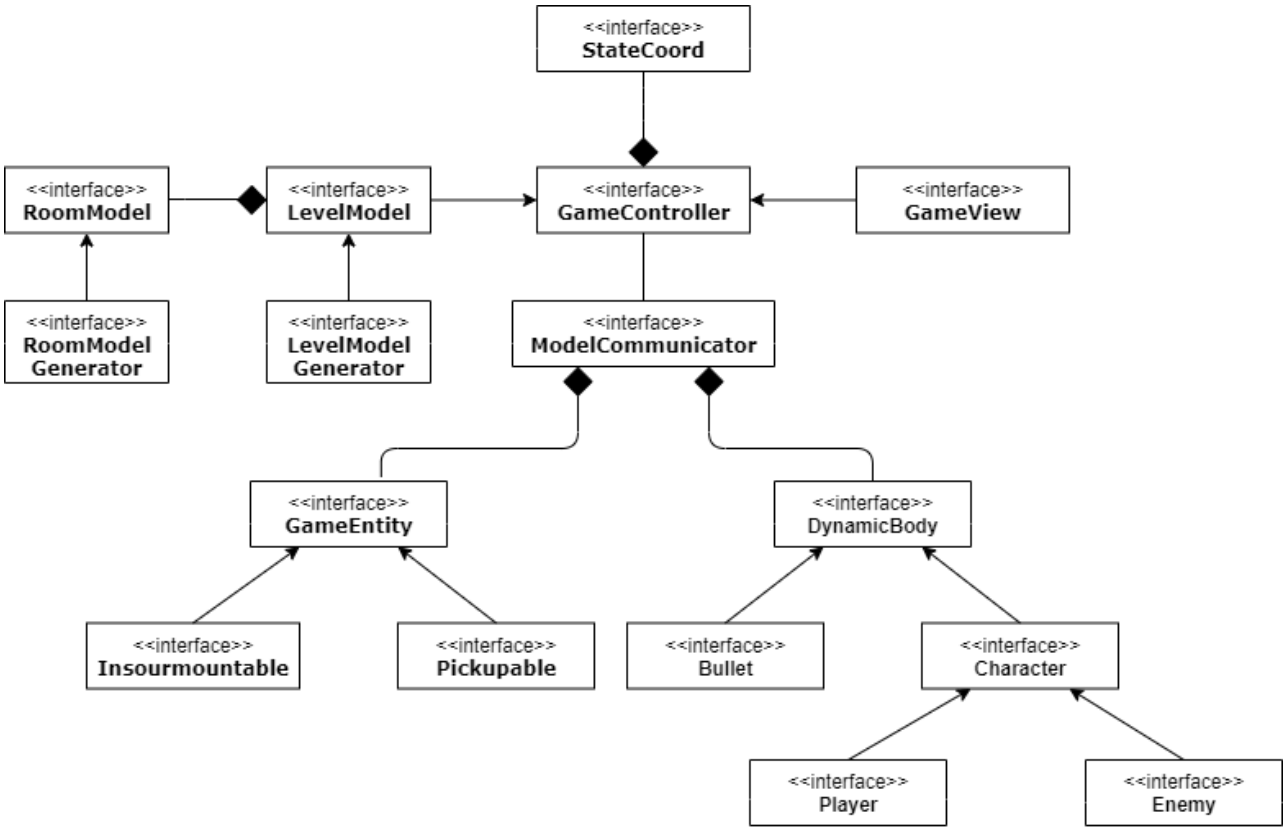



Figura 1.1: Schema UML dell'analisi dei requisiti indicante le principali entità e relazioni che comporranno l'applicazione

# Capitolo 2, Design

## 2.1, Architettura

L'architettura del software da noi realizzato segue il pattern architetturale *MVC* (model – view - controller).
La view è stata implementata attraverso l'interfaccia *GameView*. Tale interfaccia renderizza nella finestra nella finestra di gioco, nella corretta posizione le differenti entità che costituiscono il mondo di gioco.
Il Controller è implementato dalla classe *GameController* che si occupa di coordinare gli elementi grafici e gli elementi che compongono la grafica del gioco, facendoli comunicare ed interagire in maniera appropriata fra di loro.
La Model interagisce con il controller attraverso l'interfaccia *ModelCommunicator*. Il Model permette la creazione del mondo di gioco e lo spostamento del personaggio e dei nemici, gestendo anche le loro relative iterazioni.

## 2.2, Design Dettagliato

### 2.2.1 Lucia Fabbri

Il mio ruolo all'interno del progetto è stato quello di occuparmi della realizzazione del player e di tutto ciò che riguarda la sua interazione con l'ambiente di gioco.
Ho racchiuso le sue principali funzionalità all'interno della classe *PlayerImpl* che implementa l'interfaccia *Player* ed altre proprietà vengono ereditate poiché dislocate su più interfacce comuni.
In collaborazione con il mio collega Marco Ragazzini, abbiamo adattato soluzioni rivolte al riutilizzo di codice in modo da evitarne l'inutile duplicazione.
Per questo motivo abbiamo un'interfaccia principale *DynamicBody* che include metodi comuni alle varie entità dinamiche del gioco, quali i proiettili, i nemici ed il player e possiede due estensioni: l'interfaccia *Bullet* che si occupa della creazione dei proiettili e l'interfaccia *Character* che include metodi specifici riguardanti esclusivamente i personaggi.

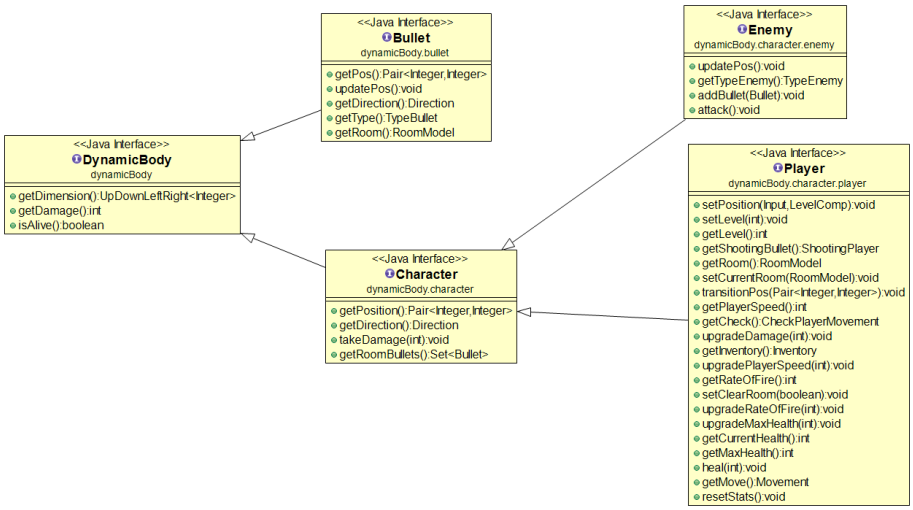


Figura 2.1: Struttura di interconnessione interfacce comuni

Per quanto riguarda il player, la sua funzionalità principale è quella di muoversi liberamente all'interno della finestra di gioco. Utilizzando la libreria esterna Slick2D(http://slick.ninjacave.com/javadoc/) per acquisire l'input da tastiera, ogni volta che viene premuto un pulsante ne effettuo il controllo. Nel nostro caso, abbiamo scelto di utilizzare i pulsanti 'W', 'A', 'S', 'D' per effettuare il movimento ed ognuno di essi rappresenta rispettivamente le direzioni alto, sinistra, basso e destra. Se il pulsante premuto corrisponde ad uno tra quelli elencati precedentemente, allora le variabili riguardanti la direzione corrente e le coordinate del player verranno coerentemente impostate utilizzando nuovi valori.
Nel realizzare questo mi sono avvalsa dell'enumerazione comune *Direction* e della classe *Pair* ed il tutto si trova all'interno della classe *MovementImpl* che implementa l'interfaccia *Movement*.
Prima di assegnare effettivamente la nuova posizione al player, ho controllato che esso non andasse in collisione con nessuno dei vari ostacoli presenti all'interno di ogni stanza e che rimanesse all'interno dei limiti di gioco prestabiliti sfruttando le classi comune *CheckPosImpl* e la relativa interfaccia *CheckPos*.
Ho poi deciso di estendere ulteriormente questi elementi, introducendo una nuova classe *CheckPlayerMovementImpl* che implementa l'interfaccia *CheckPlayerMovement* in modo tale da rendere l'implementazione più chiara e focalizzarmi sulle varie tipologie di controllo che riguardano solo ed esclusivamente il player. Muovendosi liberamente all'interno della finestra di gioco, esso può:
- scontrarsi con i nemici e ricevere da essi un danno più volte ripetutamente nel caso in cui rimanga in collisione per più di 1 secondo
- raccogliere vari potenziamenti e reagire coerentemente in base alla loro diversa tipologia
- collezionare monete e raccogliere chiavi
- trovandosi in prossimità di porte, gli è permesso lo spostamento tra le varie stanze
- una volta trovata la scala, può continuare la sua discesa nei livelli successivi

Per ciascuno dei punti appena elencati ho quindi realizzato uno specifico metodo di verifica, in modo tale da poter determinare la corretta posizione del player confrontandola con gli oggetti presi in esame e successivamente intraprendere le relative azioni da eseguire.

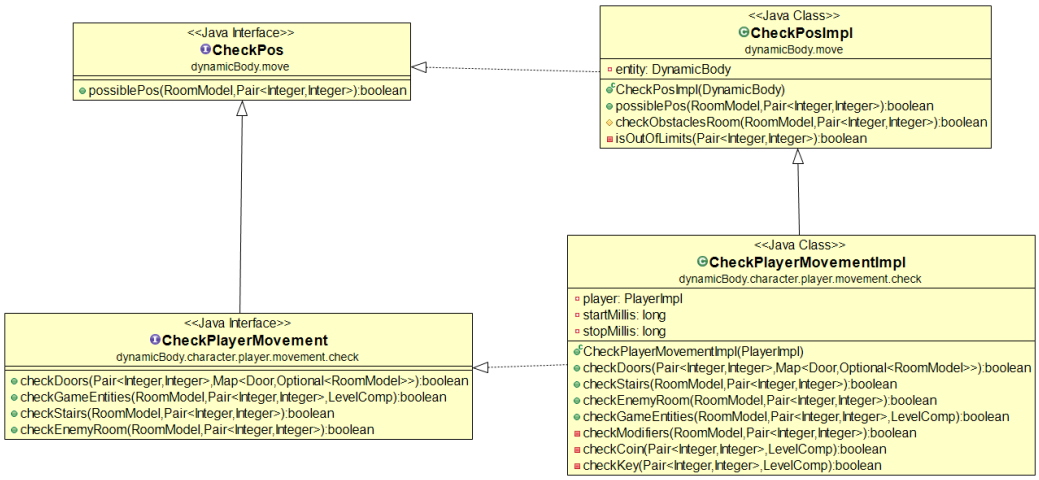



Figura 2.2: Struttura del controllo del player all'interno del mondo di gioco

Per tenere traccia degli oggetti che il player può raccogliere, quali monete e chiavi, ho successivamente creato una classe *InventoryImpl* che implementa l'interfaccia *Inventory* mentre per la gestione di tutte le informazioni e le funzionalità riguardanti la sua salute ho deciso di racchiuderle all'interno della classe H*ealthImpl* che implementa l'interfaccia *Health*. Queste informazioni riguardano la vita corrente, la vita massima e la possibilità di effettuare aggiornamenti in base ai danni ricevuti, ai potenziamenti e alle cure raccolte.

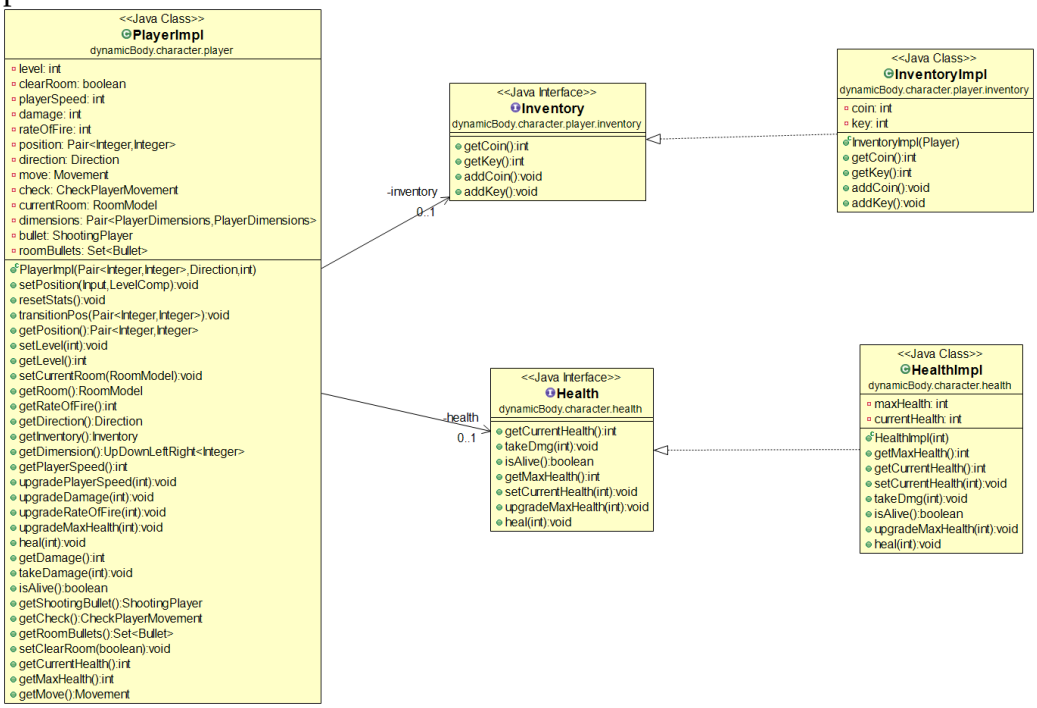



Figura 2.3: Struttura dell'inventario e della vita relative al player

La rappresentazione grafica del player nel mondo di gioco è stata affidata al mio collega Federico Pirazzoli. Sfruttando la classe generica comune *UpDownLeftRight* che restituisce le dimensioni esatte dei pixel di ciascuna immagine passatagli, il mio compito è stato quello di realizzare l'enumerazione *PlayerDimensions*.
Al suo interno ho richiamato la costruzione di un oggetto della classe appena descritta, passando come argomento i numeri interi corrispondenti alle dimensioni

esatte del player precedentemente calcolate, in modo tale da rispecchiare la direzione corrente del player.
Per quanto riguarda la gestione dell'attacco del player, ho deciso di creare la classe *ShootingPlayerImpl* che implementa l'interfaccia *ShootingPlayer*, all'interno della quale ho nuovamente utilizzato la libreria esterna Slick2D per effettuare l'acquisizione dell'input da tastiera. Il controllo viene effettuato sul pulsante della barra spaziatrice ed è possibile sparare proiettili ripetutamente tenendo premuto quest'ultimo per un tempo di default maggiore di 1 secondo. Per ricavare la corretta posizione di creazione del proiettile sulla base della posizione e della direzione correnti del player, ho utilizzato la classe comune statica *DistanceBull* ed ho richiamato il suo metodo statico 'calculateBullPos'.
Richiamando tutte le caratteristiche e le funzionalità comuni del proiettile racchiuse nella classe *BulletImpl* che implementa l'interfaccia *Bullet*, ho poi deciso di realizzare la classe *BulletPlayerImp* che si occupa della creazione del proiettile specifico del player, come specificato dall'enumerazione *TypeBullet*.
Il punto di differenziazione tra i proiettili si trova proprio nel metodo di controllo 'findCheck(TypeBullet type)' che permette di scegliere quale tipologia di controllo si vuole utilizzare relativamente ad un singolo proiettile.

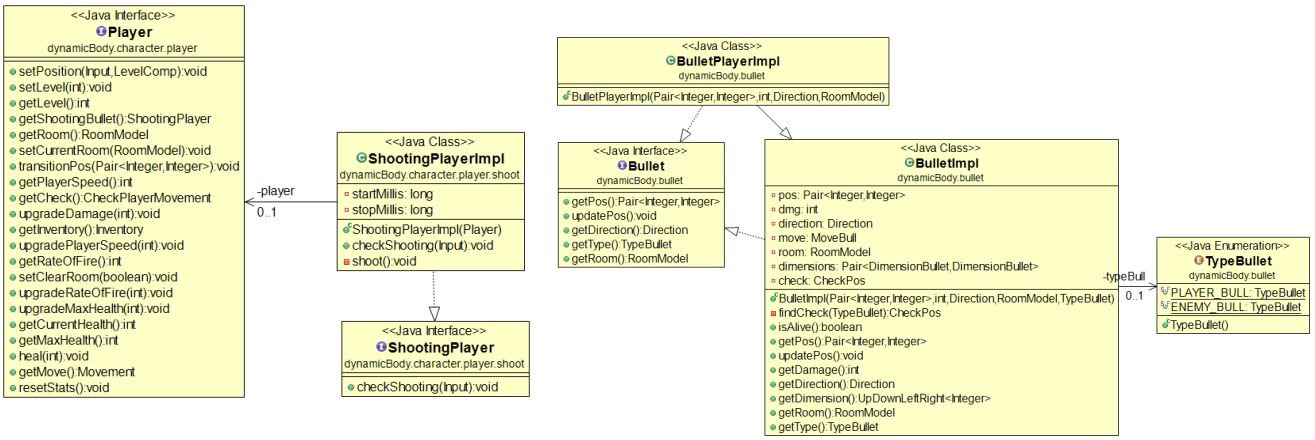



Figura 2.4: Struttura attacco del player

Figura 2.5: Struttura creazione proiettile del player

Infatti, per il controllo del movimento e della posizione del proiettile, ho pensato di riutilizzare la classe comune *CheckPos* per verificare che in caso di collisione contro il muro oppure contro i vari ostacoli presenti nelle stanze il proiettile si fermasse. Ho poi creato la classe *CheckPlayerBull* come estensione della precedente, introducendo anche il controllo sull'eventuale collisione con i nemici presenti all'interno della stanza corrente, infierendo loro del danno ogni qual volta vengano colpiti.
Per quanto riguarda il puro movimento del proiettile all'interno del mondo di gioco ho sfruttato la classe comune *MoveBullImpl* che implementa l'interfaccia *MoveBull*, dove per ogni posizione corrente del proiettile viene richiamato il controllo relativo alla sua tipologia e si verifica se la posizione successiva potrà essere occupata oppure no.

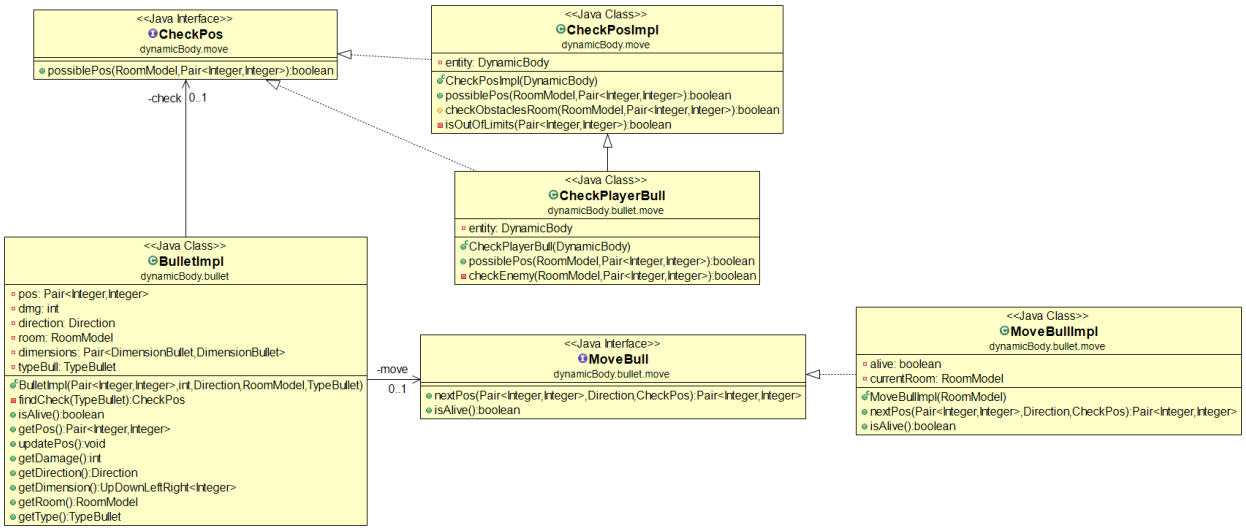


Figura 2.6: Struttura del movimento del proiettile del player

Per quanto riguarda la raffigurazione del proiettile nel mondo di gioco, abbiamo pensato di adottare la procedura analoga a quella precedentemente descritta per il player, creando in comune con il mio collega Marco Ragazzini la classe *BulletDimFactory* e l'enumerazione *DimensionBullet*, inserendo all'interno di quest'ultima le dimensioni relative ai proiettili.

### 2.2.2 Marco Ragazzini

La parte di progetto a me assegnata riguardava l'implementazione dei nemici e tutta l'interazione che questi hanno con l'ambiente di gioco.
Dentro l'interfaccia *Enemy* ho incluso le principali funzionalità dei nemici. Questa è implementata da *EnemyImpl*, utilizzata per la gestione dei nemici standard, e da *BossImpl*, utilizzata per la gestione del Boss.
Per minimizzare le ripetizioni di codice, in accordo con Lucia, abbiamo creato due interfacce: la prima di nome *DynamicBody* con i metodi comuni fra Enemy, Player e Bullet, mentre la seconda di nome *Character*, con i metodi comuni soltanto tra Enemy e Player.

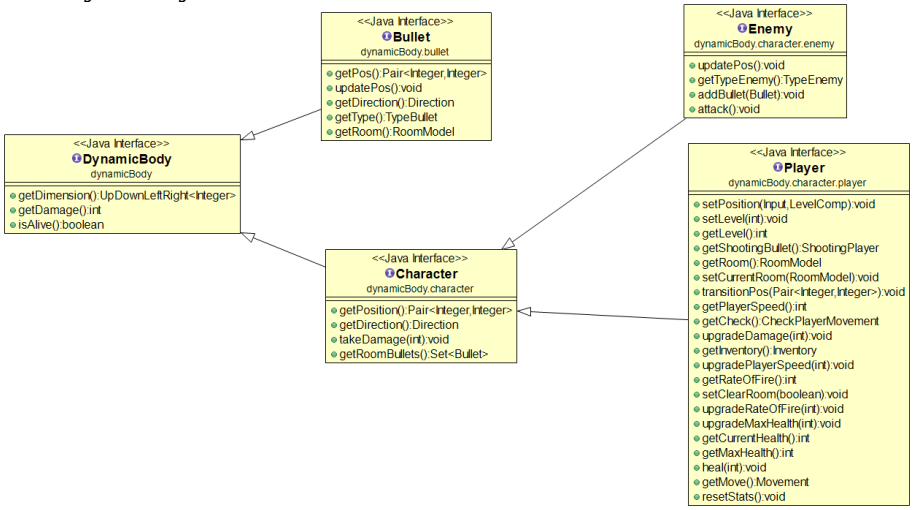


Figura 2.1: Struttura di interconnessione interfacce comuni

Ogni nemico doveva essere in grado di potersi muoversi in ogni posizione non occupata, per questo motivo ho optato per gestire il movimento tramite un *Pair* di Integer, in cui vengono salvate le coordinate del pixel in alto a sinistra del rettangolo che rappresenta il mostro.
La vita del mostro è gestita attraverso l'interfaccia *Health*, comune anche al Player, grazie ad un metodo, takeDamage, utilizzato per diminuire la vita ed a uno per calcolare se il personaggio sia ancora in vita.
Ogni nemico potrà avere differenti tipi di attacchi, movimenti e dimensioni, gestiti tramite delle Factory. La dimensione viene poi gestita grazie ad una classe Generica di nome *UpDownLeftRight*, la cui funzione è quella di salvare al suo interno quattro valori e, utilizzando gli appositi get, ritornarli quando necessario.

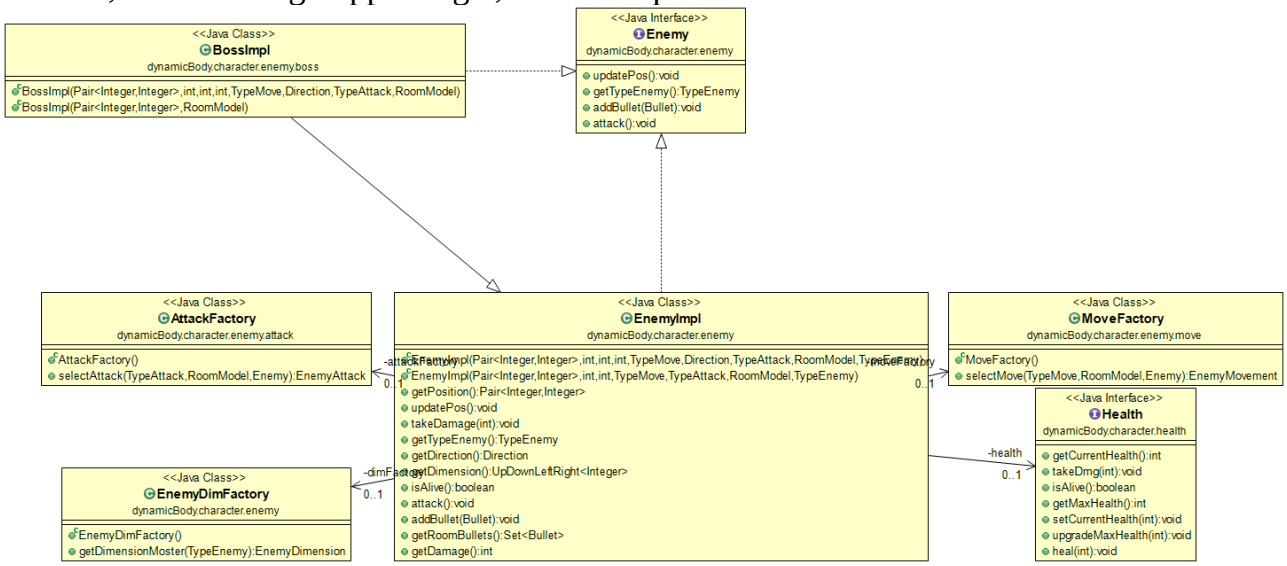


Figura 2.2: Struttura della Gestione dei vari nemici e delle Factory

L'interfaccia *EnemyMovement* ha lo scopo di gestire il movimento dei nemici. Al suo interno, infatti, troviamo il metodo per ricavare la prossima posizione del nemico, nextPos, ed un metodo per ricavare la nuova direzion. Questa interfaccia è implementata da cinque classi, ognuna delle quali corrisponde a un diverso tipo di movimento:

- *StraightMove*: permette al nemico di muoversi nella stessa direzione finché non si collide con un ostacolo o con un muro esterno della stanza. Una volta entrato a contato con uno di questi, cambierà direzione, non obliqua, e ricomincerà da capo.
- *TeleportMove*: permette, ogni cinque secondi, al nemico di teletrasportarsi in una posizione casuale nella stanza, purché questa non sia occupata da un ostacolo, per poi rimanere immobile per i secondi rimanenti.
- *RandomMove*: permette al nemico di muoversi casualmente. Dopo aver compiuto un numero casuale tra 250 e 380 o aver colliso prima con un muro o un ostacolo, di cambiare la direzione in una fra quelle possibili, scelta casualmente, e muoversi verso questa, finché non sarà necessario.
- *ImmobilizedMove*: semplicemente il nemico non potrà muoversi rimanedo fermo nella posizione iniziale.
- *ToPlayerMove*: permette al nemico di calcolare il percorso minimo che quest'ultimo deve svolgere per raggiungere il Player attraverso un algoritmo di Dijkstra. Questo viene raggiornato ogni volta che il personaggio, muovendosi, cambia il proprio Tile.

Per gestire il tipo di movimento da passare al nemico ho deciso di creare una Factory, *FactoryMove*, che, in base al valore dell'enumerazione *TypeMove* passatogli, ritornerà l'oggetto corretto.

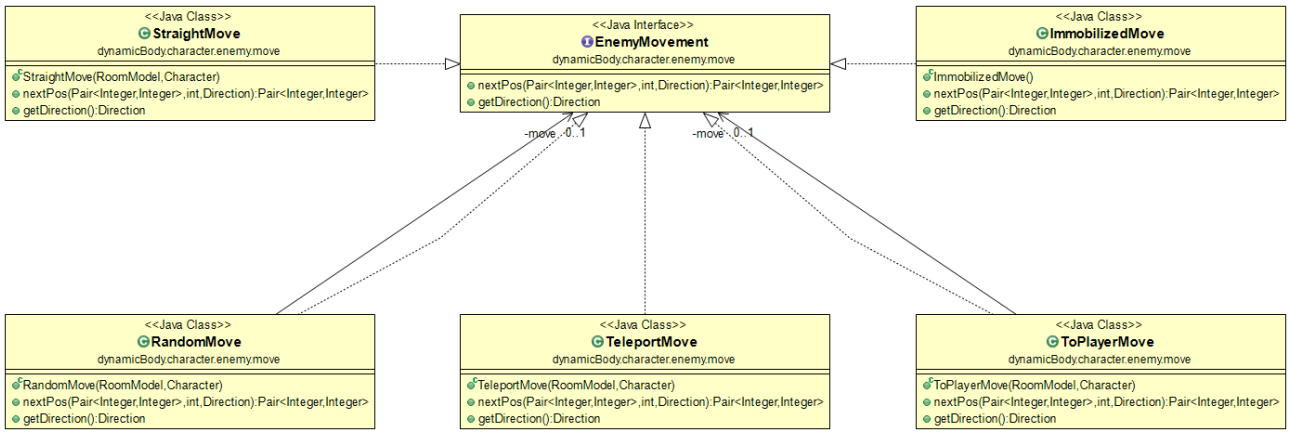


Figura 2.3: Struttura dei vari movimenti del Nemico

Per controllare le collisioni che i nemici possono avere con i muri e gli ostacoli ho implementato l'interfaccia *CheckEnemy* che estende *CheckPos* e presenta il metodo, changeDir, che, dopo la collisione del nemico con un muro o un ostacolo, cambia la direzione del nemico.

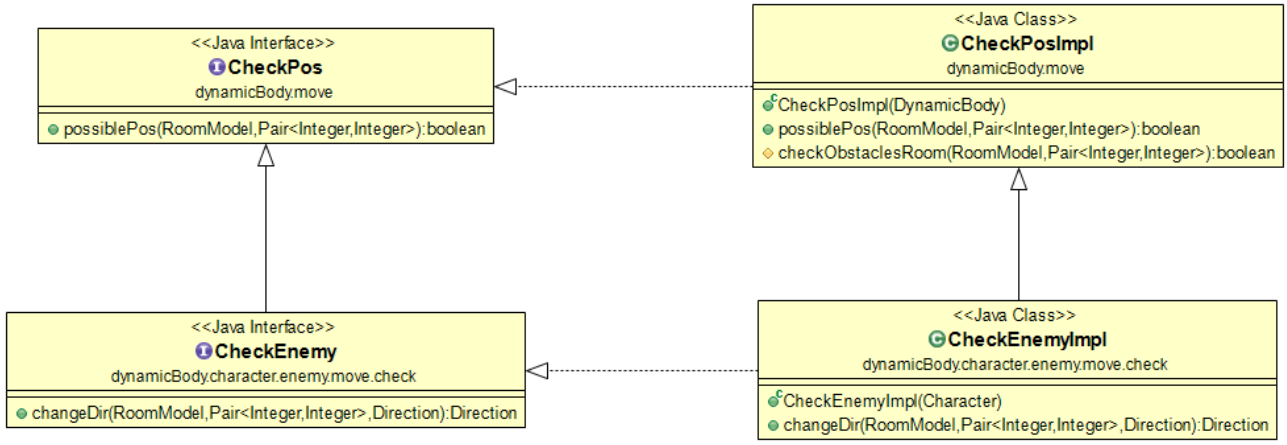


Figura 2.4: Struttura del controllo del nemico all'interno del mondo di gioco
L'interfaccia *EnemyAttack* ha lo scopo di gestire la creazione dei proiettili. Questa interfaccia è implementata a sua volta da quattro differenti classi:

- *OneSideAttack*: permette al nemico di creare un proiettile di fronte a lui che si muoverà nella sua stessa direzione finché non entrerà in contatto con il Player, un ostacolo o un muro.
- *TwoSideAttack*: permette al nemico di creare due proiettili, nelle due direzioni adiacenti rispetto a quella in cui sta guardando il nemico. Il movimento di questi due proiettili sarà diagonale. Sfruttando le due tipologie di attacco precedentemente descritte.
- *TripleAttack*: permette al nemico di creare tre proiettili, uno di fronte a lui, con la sua stessa direzione, e due nei lati adiacenti, con le loro direzioni di movimento diagonali.
- *FourSideAttack*: permette al nemico di creare quattro proiettili, uno per ogni direzione cardinale.

Anche per gestire il tipo di attacco da passare al nemico ho preferito creare una Factory, *FactoryAttack*, che, in base al valore dell'enumerazione *TypeAttack*, ritornerà l'oggetto corretto.

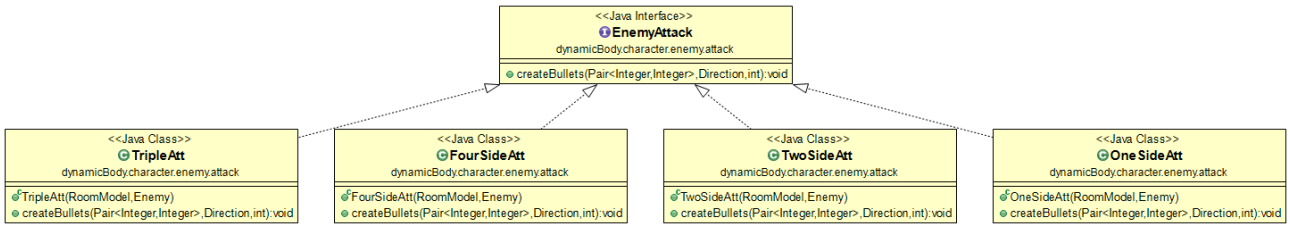


Figura 2.5: Struttura dei vari tipi di attacco del nemico

L'interfaccia *Bullet* racchiude tutte le funzionalità dei proiettili. Tale interfaccia è implementata dalla classe *BulletImpl*. *Bullet*, estendendo *DynamicBody*, in modo tale da avere metodi per gestire le dimensioni, i danni e il controllo per sapere se l'oggetto sia ancora in vita, verificando se non sia entrato in contatto con altre entità di gioco. Attraverso l'enumerazione *TypeBullet*, vi è la possibilità di sapere se il proiettile è stato lanciato dal personaggio principale o da un nemico. La differenza principale fra questi due proiettili è che il primo deve colpire i nemici ed applicare loro i danni, mentre il secondo colpirà soltanto il Personaggio principale, ma non i nemici. Le enumerazioni *BulletDimension* e *BulletDefault* forniscono i valori standard del proiettile. Anche la gestione delle Dimensione è basata su una factory. Per evitare che i proiettili venissero creati sovrapposti al personaggio, ho creato una classe statica, con costruttore privato, così che non potesse essere inizializzata, questa in base alla direzione, calcola le coordinate per la creazione del proiettile. Per semplificare la creazione dell'oggetto ho deciso di creare una classe *BulletEnemy*, ed richiamando il costruttore della superclasse, ho evitato la ripetizione relativa al tipo di proiettile del Nemico, cercando così di evitare errori relativi al tipo dei proiettili. L'interfaccia *BulletMove*, implementata da *BulletMoveImpl*, ha lo scopo di ritornare la posizione successiva del proiettile, in base alla sua direzione. Il *CheckEnemyBull*, che estende *CheckPosImpl* ed implementa *CheckPos*, oltre a controllare se il proiettile entra in contatto con il bordo della mappa o con un ostacolo, controlla anche se colpisce il personaggio.

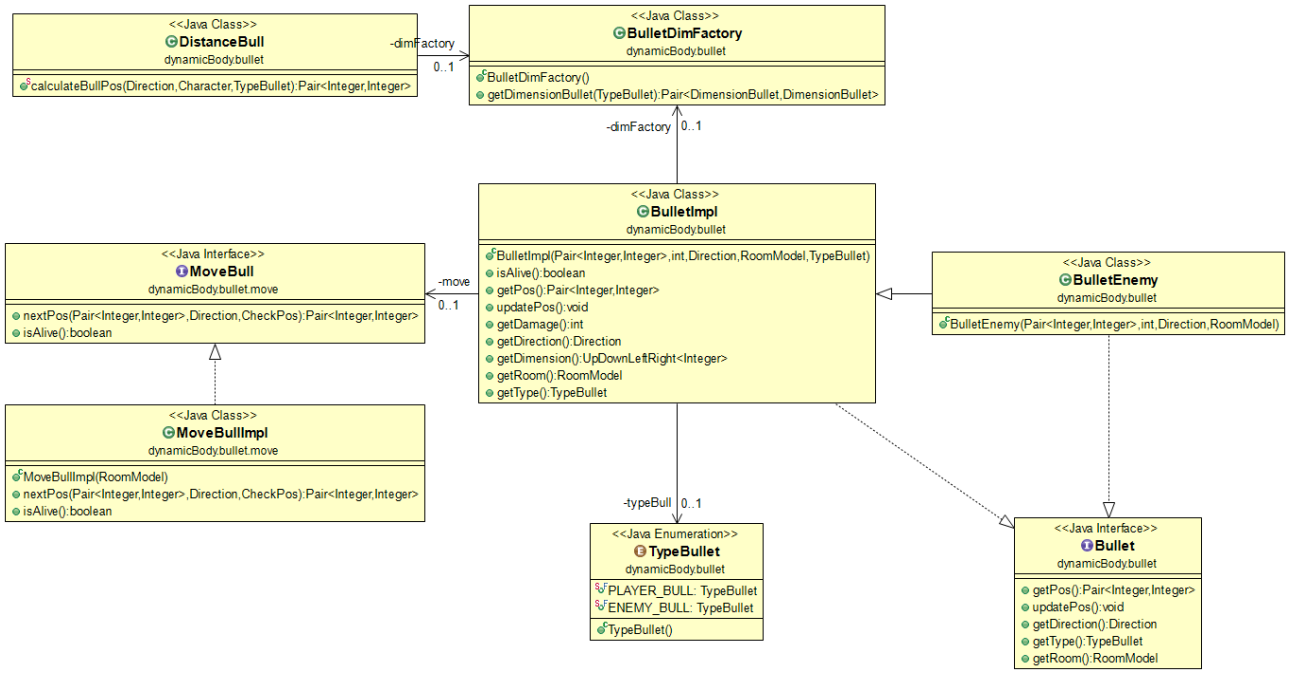
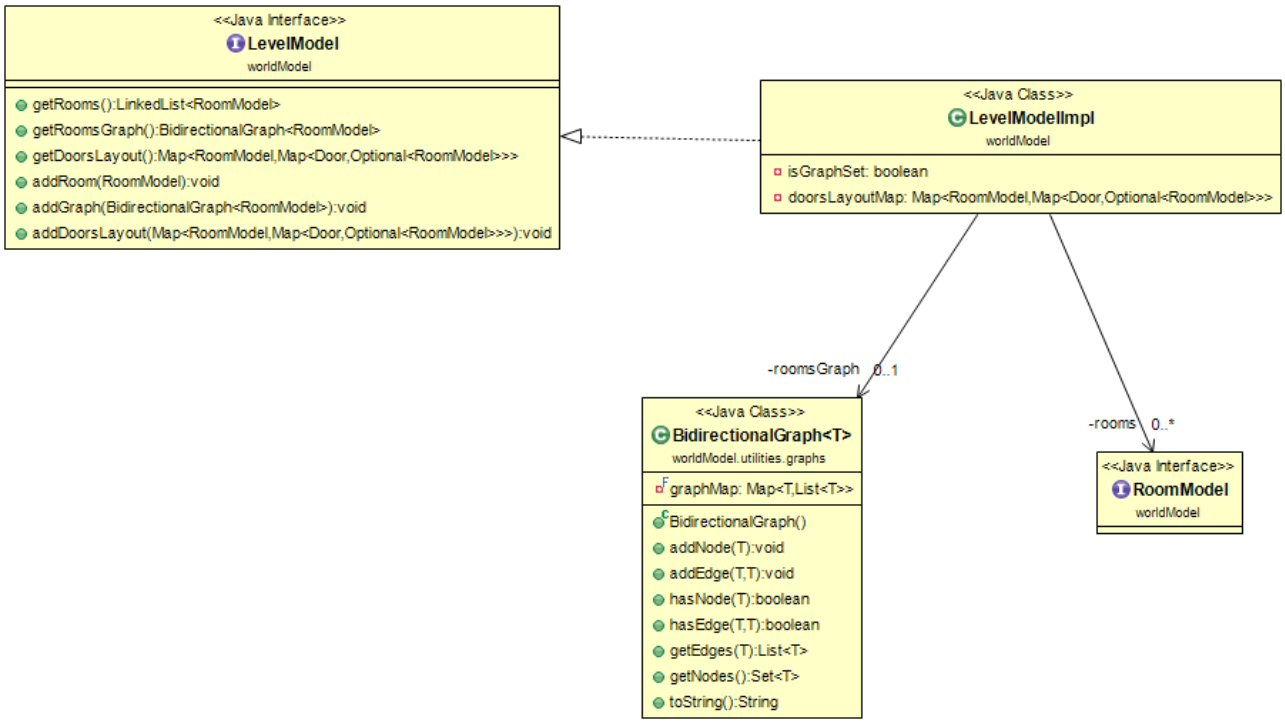



Figura 2.6: Struttura dei proiettili

Infine, per semplificare la creazione dei nemici ho creato l'interfaccia *EnemyCreator* che fornisce la possibilità di creare un nemico differente in base a valore dell'enumerazione *TypeEnemy* che gli si passa, e di generare il Boss del gioco.

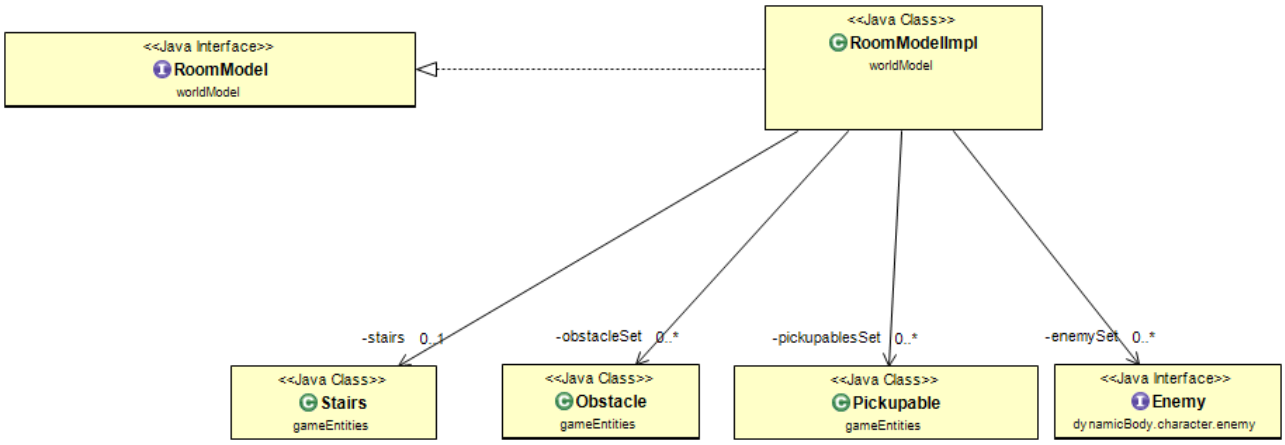
### 2.2.3 Gian Luca Nediani

Il mio compito principale nel progetto è stato quello di modellare il mondo di gioco, e dunque i livelli e le stanze che lo compongono.
Il seguente diagramma UML definisce la struttura di un livello:



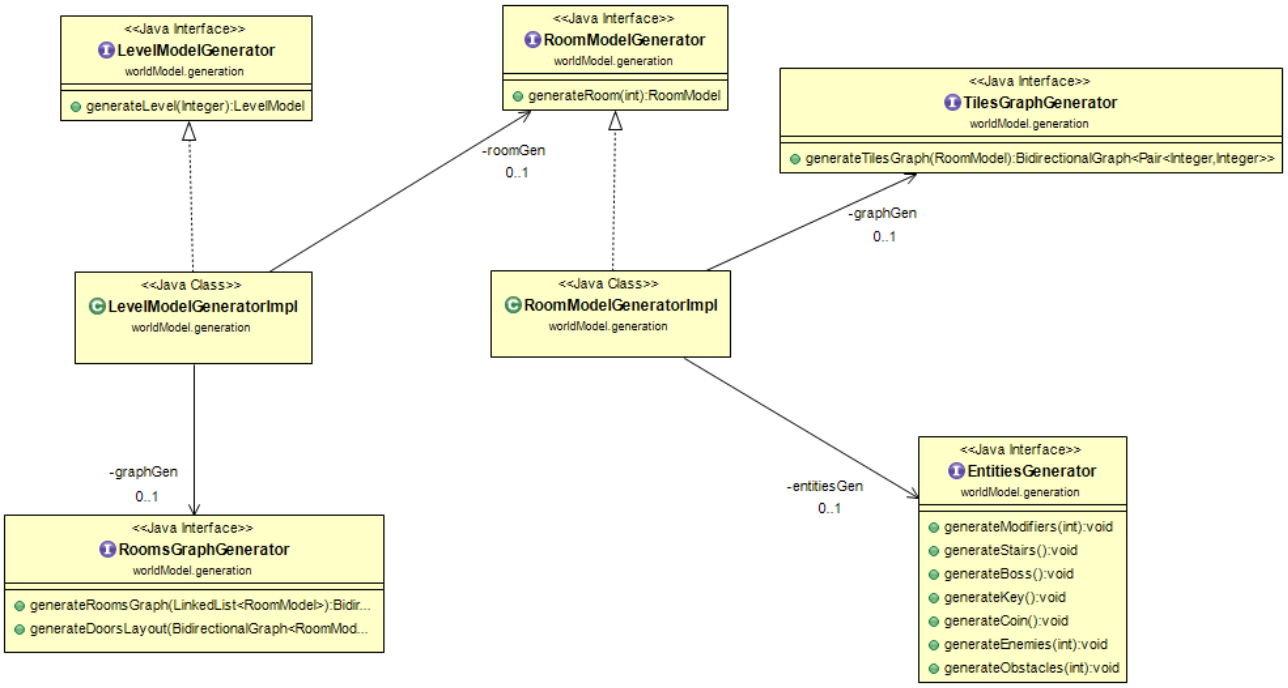

L'interfaccia *LevelModel* definisce le operazioni principali su un livello di gioco, mentre la sua implementazione contiene le strutture dati necessarie alla modellazione: un livello è composto di una lista di stanze e delle strutture dati per definire le connessioni fra le stanze.
Questa implementazione dell'interfaccia potrebbe in futuro essere scambiata ad un'altra garantendo così la possibilità di evolvere la struttura del mondo di gioco in futuro senza stravolgere l'assetto utilizzato.

Il prossimo diagramma mostra invece la progettazione delle stanze:




Le stanze di gioco non sono altro che una composizione degli oggetti che la popolano: mostri, oggetti, ostacoli e così via. Qualora si volesse ampliare il gioco aggiungendo nuove entità, basterebbe sostituire all'attuale implementazione dell'interfaccia *RoomModel*, una versione attrezzata a gestire le nuove feature.

Dopo essermi confrontato con i miei partner di progetto, ho optato per la generazione di livelli di gioco "casuali", istanziati da generatori di livelli, definiti nell'interfaccia *LevelModelGenerator*; l'utilizzo di livelli casualmente generati, non solo aderisce al genere Rogue-like che ci eravamo prefissati di emulare, ma garantisce unicità a ogni livello e ad ogni partita, obiettivo minimo del nostro progetto.
La casualità dei livelli è garantita da file di configurazione per ciascuno dei quattro livelli, che indicano i limiti inferiori e superiori per la generazione di nemici, modificatori, stanze e di tutti gli altri elementi. Tali file contengono inoltre le statistiche (tipo, danno, salute) dei nemici per ogni livello, a salire di difficoltà a mano a mano che si avanza. Anche il grafo che collega le stanze di un livello è generato casualmente per ottenere livelli sempre diversi fra loro a ogni partita. I grafi dei livelli sono bidirezionali e coerenti col mondo di gioco, questo vuol dire che se la stanza A si collega con la stanza B tramite la porta NORD, allora la stanza B si collega a B tramite la stanza SUD.




I generatori di livelli si compongono di generatori di stanze e generatori di grafi per i collegamenti fra le stanze; i generatori di stanze si compongono a loro volta di generatori di entità, ovvero gli elementi che popolano le stanze.
L'infrastruttura di generazione si avvale del pattern *Factory Methods*, i generatori sopra elencati sono in grado di "fabbricare" tutti gli oggetti che formano un livello, alla richiesta della creazione di quest'ultimo.

Nel generatore di livelli *LevelModelGenerator* è invece utilizzato il pattern *Strategy*: il generatore è infatti in grado di produrre qualsiasi dei quattro diversi livelli di gioco e la scelta può essere fatta a run-time; l'algoritmo a quel punto caricherà dinamicamente la configurazione necessaria alla creazione casuale del livello scelto.

### 2.2.4 Federico Pirazzoli

Il mio ruolo all'interno del progetto riguardava principalmente l'implementazione di elementi che permettessero al mondo di gioco di interagire con tutte le entità presenti in esso. Nel fare questo, mi sono avvalso della libreria esterna Slick2D.
Più nello specifico, le principali mansioni che ho svolto sono:

- Modellazione ed implementazione dei diversi aspetti funzionali di base, come la rappresentazione (rendering) degli elementi grafici del gioco e degli aspetti logici (logic).
- Semplice implementazione per modellare e richiamare i componenti dello stesso mondo di gioco.
- Modellazione di un sistema di conservazione dei vari elementi presenti all'interno di ogni stanza di ogni livello, in modo tale semplificare l'esecuzione della logica e la renderizzazione della parte grafica.

- Modellazione e disegno del menù e dei vari elementi grafici che compongono la GUI, così da ottenere una migliore esperienza di gioco.
- Stabilire un'implementazione per permettere al gioco di cambiare livello (o “state”, come sono implementate all'interno del progetto) senza problemi o rallentamenti, conservando determinati elementi da un livello ad un altro (principalmente legati al personaggio principale).
- Permettere al software di essere eseguito anche su sistemi Linux e MacIOS e non solamente su sistemi Windows.
- Implementazione di suoni, musiche e grafica creati principalmente da me, per rendere il progetto in sé più vivo e realistico nonostante non fossero mansioni fondamentali al compimento del progetto.

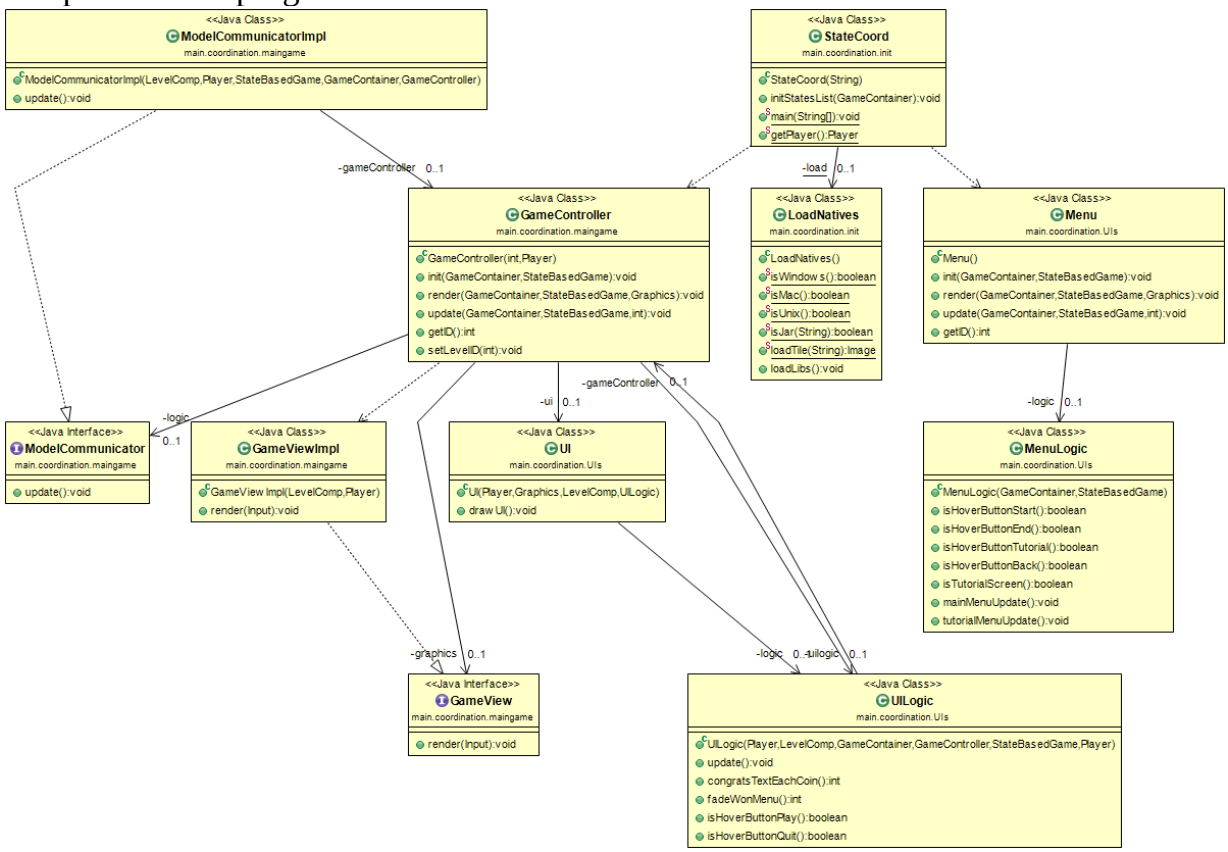



Figura 2.1: Processo di inizializzazione

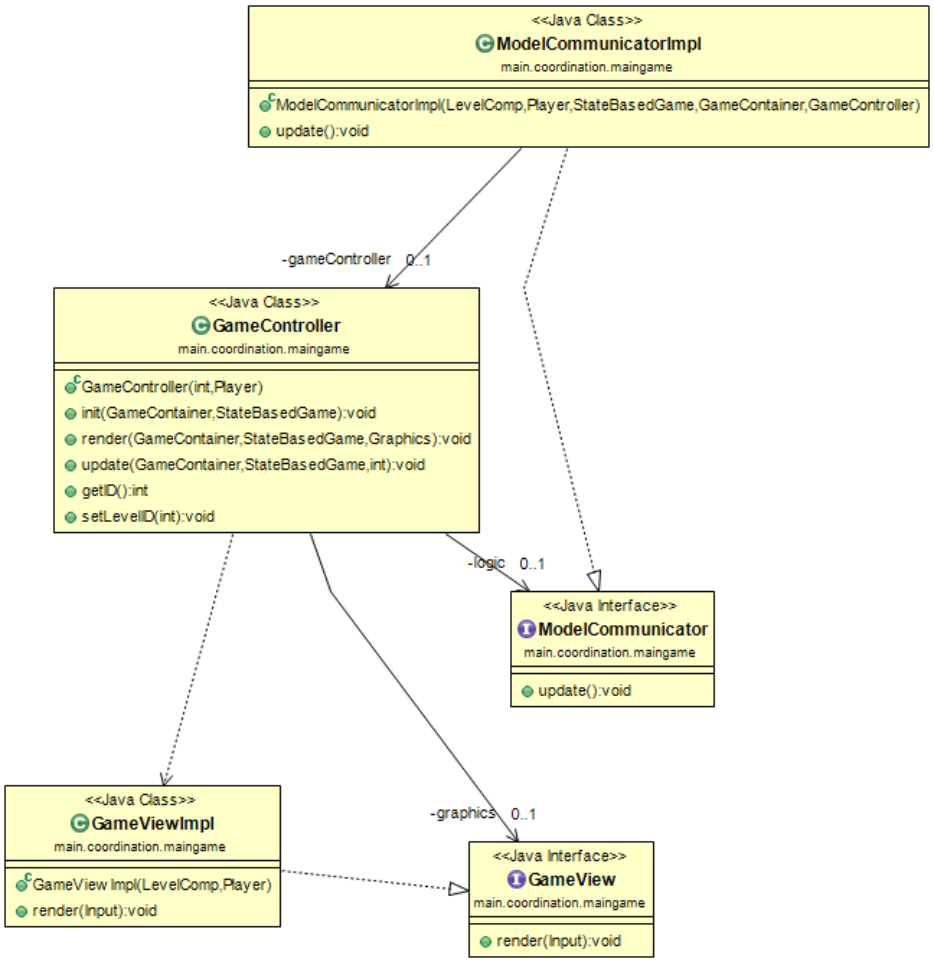



Figura 2.2: Controllo del Gioco

Nella prima fase di sviluppo, il mio compito principale è stato quello di testare e capire come la libreria Slick2D potesse essere utilizzata nel modo più efficiente e sensato per soddisfare i nostri bisogni.
Il suo utilizzo ci ha permesso di creare un sistema a “state” (o stati) per identificare i vari componenti del gioco, come il menù e la classe che gestisce i livelli all’interno della quale è presente il “cuore” del gioco.
Abbiamo quindi la classe *GameController* che estende il *BasicGameState* da Slick2D e contiene tre metodi obbligatori:
- Init(), utilizzato prima di qualsiasi altra azione per caricare le risorse necessarie al programma per poter esser eseguito
- Render(), contiene tutti i riferimenti agli elementi utilizzati nella renderizzazione di poligoni, immagini o elementi grafici all'interno del programma, oltre ad avere riferimenti alla classe grafica del contesto OpenGL che si crea e quindi permettere di reindirizzare elementi grafici nella finestra
- Update(), similmente a Render, deve contenere tutti i riferimenti agli update o alle modifiche degli elementi e delle entità presenti nel mondo di gioco.
La classe *GameController* è chiamata dalla classe *StateCoord*, la quale si occupa a sua volta di chiamare la classe LoadNatives per iniziare il caricamento delle librerie necessarie ad eseguire il programma, inizializzare l'entità Player ed infine passare il

controllo alla classe Menu. Così come *GameController*, anche quest'ultima classe estende *BasicGameState* e permette di creare la videata iniziale dove poter scegliere di iniziare il gioco, uscire dall'applicazione oppure leggere un breve tutorial. L'aspetto grafico si trova nella classe Menu mentre la logica dei bottoni si trova nella classe *MenuLogic*.

L'adozione di questo tipo di modello mi ha permesso di dividere in maniera pulita ed efficiente le varie attività compiute dalle entità e dal mondo di gioco, separando ulteriormente quest'ultime in altre due interfacce: *GameView* e *ModelCommunicator*. Il loro ruolo è quello di distinguere chiaramente quali elementi compiono una determinata azione e sono tutti contenuti nel package *Coordination*, il quale racchiude al suo interno tutti gli aspetti di coordinazione del gioco.

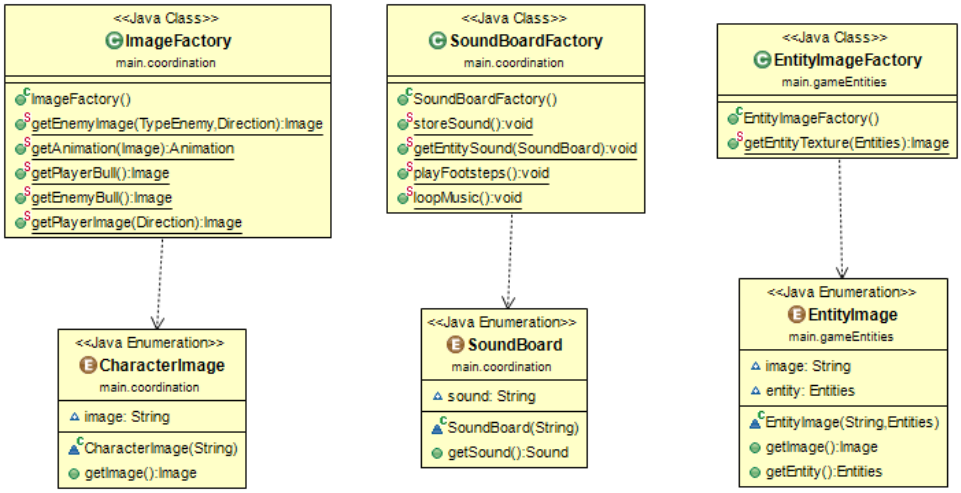



Figura 2.3: Factories delle risorse

Abbiamo inoltre deciso di immagazzinare gli elementi grafici e sonori del progetto dentro delle Enums, rispettivamente:

- *CharacterImage* e *ImageFactory*, per conservare le immagini delle entità del gioco, come i nemici ed il Player.
- *EntityImage* e *EntityImageFactory*, per conservare le texture degli oggetti e dei potenziamenti del gioco, come gli items ed i modifiers.
- *AnimatedTile*, *AnimatedTileImage*, *Tile* e *TileImage*, tutte dedicate alla creazione e conservazione dei tiles delle stanze.
- *SoundBoard* e *SoundBoardFactory*, per conservare gli elementi audio di tutti gli eventi che ne richiedono l'utilizzo, come l'essere colpiti da un proiettile nemico o l'apertura delle porte.

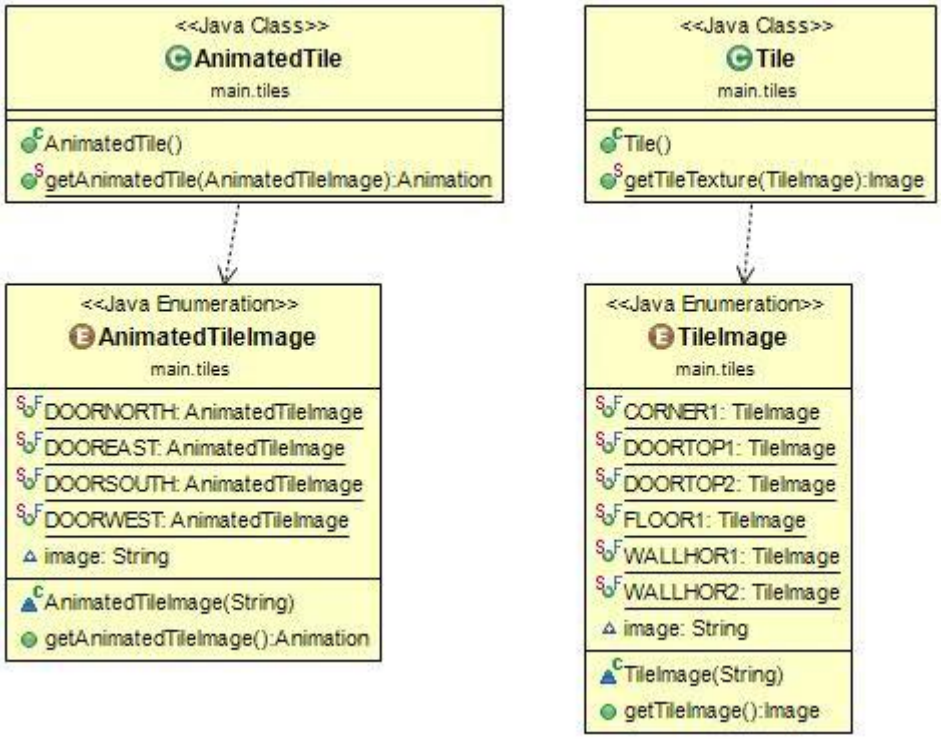



Figura 2.4: Tiles animati e normali

Uno dei problemi iniziali è stato quello di capire esattamente come la libreria interpretasse l'asse cartesiano della finestra, che ho poi scoperto avere origine nell'angolo in alto a sinistra e le coordinate che rappresentano la lunghezza e altezza della finestra totali nell'angolo in basso a destra.
La facilità di rappresentare gli elementi grafici ha influito nella scelta della libreria esterna Slick. Prendendo spunto tra i vari Game Engines, ho pensato di utilizzare i Tiles, ovvero "tasselli" di grandezza predefinita (nel nostro caso, 64x64 pixels), che vengono utilizzati per disegnare tutto ciò di cui si ha bisogno sullo schermo in modo tale da mantenere tutte le proporzioni senza rischiare di avere textures più allargate o ristrette del dovuto. Attraverso l'utilizzo dei Tiles a 64 pixels, siamo inoltre riusciti a "modellare" la griglia di gioco delle entità, utilizzando suoi multipli per costruire la base del mondo di gioco (pavimento, muri, porte) senza troppe difficoltà.

Ho poi immagazzinato gli elementi grafici delle stanze (pavimento, muri, porte) all'interno del package main.tiles per permettere a questi elementi di rimanere caricati in memoria e di conseguenza alleviare il lavoro del programma, in quanto l'idea era quella di riutilizzare le texture dei muri, delle porte e del pavimento per formare nuove stanze.
Tutto questo è stato realizzato per poter permettere ai miei colleghi di scrivere liberamente del codice ed essere privi dalle dipendenze di Slick, se non per il lancio delle eccezioni.


Figura 2.5: Implementazione delle stanze e dei livelli

L'idea che ho avuto per contenere tutte le informazioni e dati di ogni stanza dei livelli è stata quello di creare l'interfaccia *LevelComp*. Essa permette la creazione di una lista contenente un altro oggetto di tipo Room, la quale contiene tutto ciò che è necessario al Player per continuare a giocare: dalla texture dei muri, delle porte e del pavimento ai nemici, ostacoli ed oggetti presenti nel mondo di gioco.
La costruzione di tutto questo è stata possibile grazie all'utilizzo di un metodo che crea in maniera "randomica" dei livelli contenenti delle stanze rappresentate da grafi, realizzato dal mio collega Nediani.
Questo meccanismo permette alla classe *ModelCommunicator* di richiamare le API delle entità create da Fabbri e Ragazzini molto facilmente e velocemente.

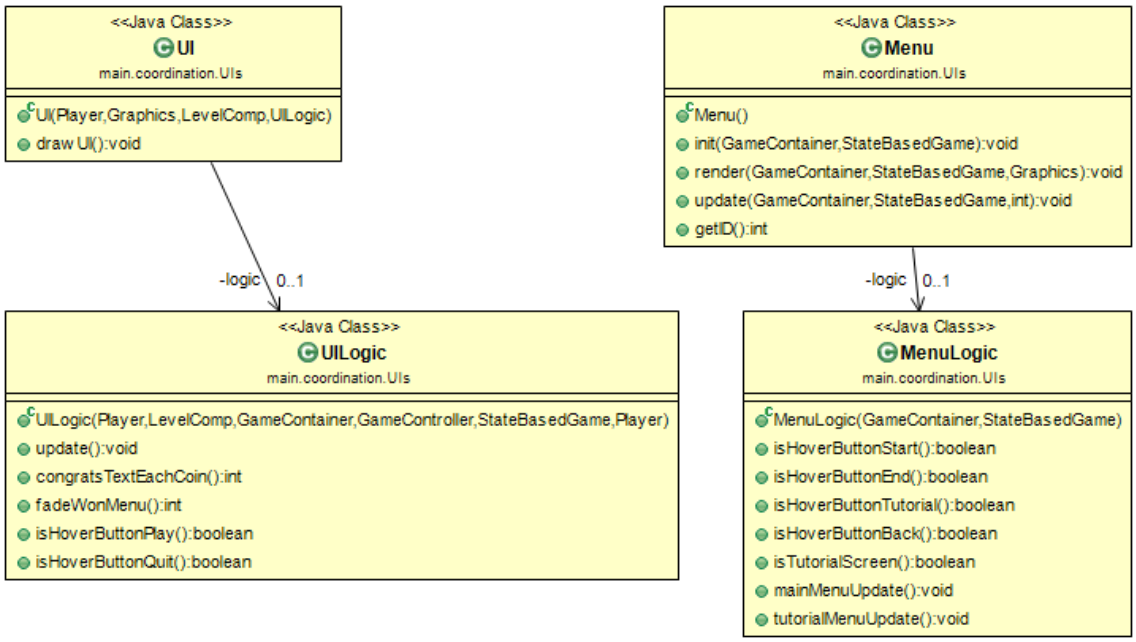



Figura 2.6: Implementazione UI e menu

La GUI ed il menù sono invece semplicemente creati utilizzando il contesto Graphics offerto dalla funzione render, come già menzionato precedentemente, vengono implementati all'interno della classe *UI* e traggono il necessario per disegnare elementi dinamici da altri oggetti già stanziati come il livello o il Player, lasciando la logica di bottoni e simili alla classe *UILogic*.

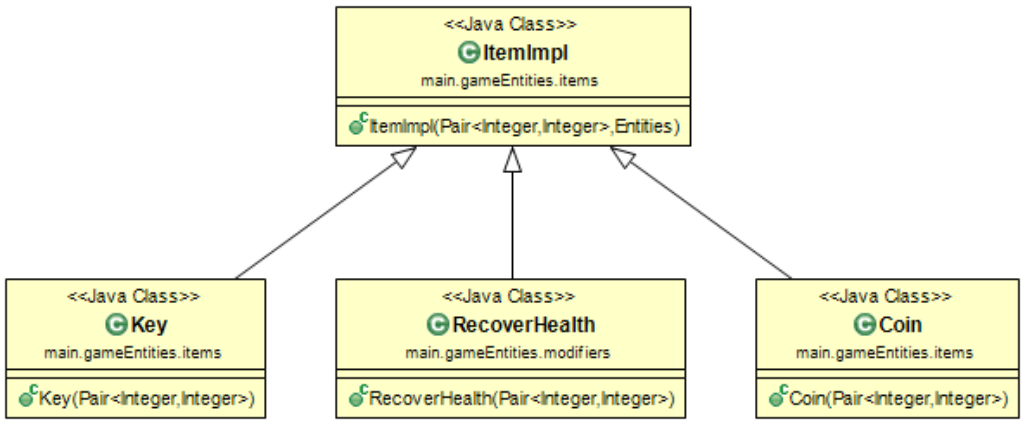



Figura 2.7: Implementazione degli oggetti

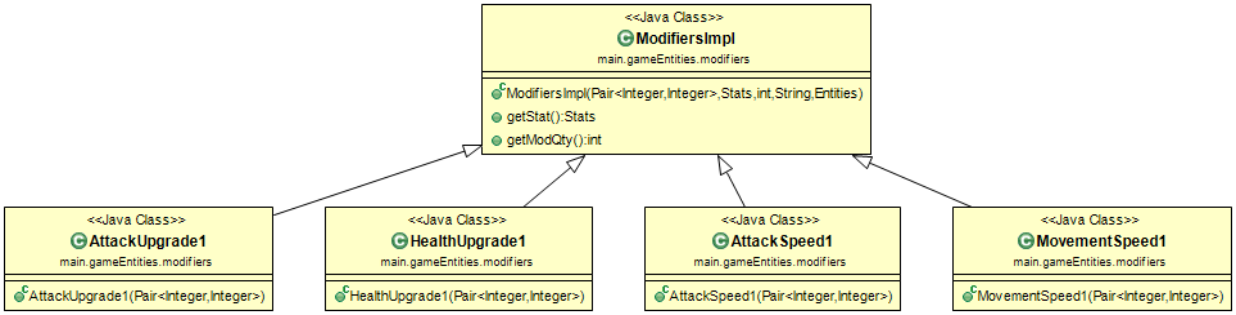



Figura 2.8: Implementazione dei modificatori

I vari oggetti e potenziamenti estendono la classe *Pickupable* realizzata del mio collega Nediani. Come la parola suggerisce, al suo interno troviamo tutte entità che il Player può raccogliere dal mondo di gioco per proseguire la sua esplorazione, migliorare le proprie abilità o solamente aggiungere alla sua collezione.
La loro posizione ed il loro tipo sono sempre creati in maniera randomica e variano da stanza a stanza.

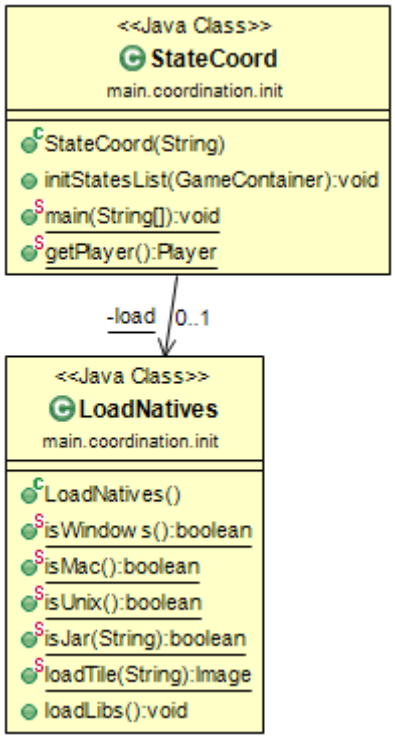



Figura 2.9: Inizializzazione delle librerie

Per quanto riguarda la compatibilità con altri sistemi operativi, la maggior parte del lavoro si è basata sull'assicurare che elementi quali i settings di Nediani per la generazione dei livelli, le texture e gli elementi audio all'interno del progetto venissero selezionati utilizzando i separatori adeguati, caricando le risorse dal *Classpath* in modo tale che, indipendentemente dal contesto di esecuzione, il file *jar* possa essere eseguito prima estraendo determinate risorse all'interno della cartella *temp* del sistema operativo, poi caricandole per permettere l'esecuzione.

# Capitolo 3, Sviluppo

## 3.1 Testing automatizzato

Per molteplici parti del codice sono stati realizzati dei test tramite il framework *JUnit*, per assicuraci del loro corretto funzionamento anche a fronte di modifiche al codice. In particolare, sono presenti test automatizzati per i seguenti aspetti del software:

- Creazione nemici
- Movimento nemici
- Attacchi nemici
- Funzionalità player
- Creazione proiettili player
- Input da tastiera
- Algoritmo di Dijkstra
- Generazione livelli di gioco
- Grafi ed esplorazione di essi

## 3.2 Metodologia di lavoro

Per collaborare in remoto alla realizzazione del progetto ci siamo serviti del DVCS Git con una repository condivisa; ogni membro disponeva di un branch personale dove testare le feature da lui sviluppate, prima di farle confluire nel branch *development*. Il branch *master* ospita invece la versione stabile e completa del software.

La suddivisione dei compiti concordata inizialmente è stata rispettata; di seguito ogni membro elenca nello specifico le componenti su cui ha lavorato.

**Lucia Fabbri**: mi sono occupata dal movimento del personaggio nel mondo di gioco, della sua interazione con i vari oggetti ed ostacoli e della capacità di attacco.

**Marco Ragazzini**: il mio compito è stato quello della progettazione e implementazione dei nemici e tutto ciò che ne compete (movimenti, attacchi…) e della gestione dei proiettili.

**Gian Luca Nediani**: la macroarea di cui mi sono occupato individualmente è quella della modellazione del mondo di gioco, vale a dire il contenuto del package *worldmodel*. Tale package comprende la definizione e implementazione di:

- Livelli e stanze di gioco (package *worldmodel*).

- Generatori di livelli, grafi dei livelli, stanze e entità di gioco, nonché di grafi per la gestione di svariati aspetti logici del gioco (sotto-package *worldmodel.generation*).
- Utility per lo sviluppo (sotto-package *worldmodel.utilities*).
- Enumerazioni per insiemi di valori costanti (sotto-package *design.utilities.enums*).
- Grafi e classi per la esplorazione di tali grafi (sotto-package *design.utilities.graphs*).

**Federico Pirazzoli**: i miei compiti riguardavano la gestione della parte grafica del progetto attraverso l'utilizzo della libreria Slick2D, la coordinazione tra la parte grafica e quella logica del programma, la creazione di oggetti e potenziamenti ed infine il caricamento di risorse per il file JAR.

## 3.3 Note di sviluppo

### 3.3.1 Lucia Fabbri

Nello sviluppare il progetto ho sempre cercato di:

- Rispettare le convenzioni del linguaggio Java ed utilizzare nomi esplicativi in modo da avere uno stile di codice corretto e facilmente leggibile.

- Spiegare e rendere il più comprensibili possibili punti critici e scelte particolari all'interno del codice, introducendo commenti aggiuntivi oltre alle annotazioni ed ai tag standard richiesti per la documentazione.

- Ho fatto uso della libreria esterna "Slick2D" http://slick.ninjacave.com/javadoc/ per l'acquisizione dell'input da tastiera, ovvero per permettere il movimento del player e la partenza del relativo proiettile.

- Per quanto riguarda il controllo di collisione del player contro i nemici e la possibilità di sparare proiettili ripetutamente tenendo premuta la barra spaziatrice, non ho potuto utilizzare il metodo Thread.sleep() della classe Thread in quanto avrebbe fermato l'intera esecuzione dell'applicazione.

  Ho quindi pensato di inserire due variabili (startMillis e stopMillis) che controllassero il passare del tempo ed effettuando la loro successiva differenza sono riuscita a fronteggiare questo problema.

### 3.3.2 Marco Ragazzini

Per l'implementazione della porzione del mio codice ho utilizzato meccanicismi di:

_ **Stream** anche se in minima parte.

_ **Factory Pattern** per la gestione del movimento, attacco e dimensioni.

_ **Generici** nella classe UpDownLeftRight <X>, in cui X specifica il tipo di ritorno di value, e nella classe CircularList<E>, in cui E rappresenta il tipo di elemento.

_ Utilizzo dell'**Algoritmo di Dijkstra** preso da **https://www.baeldung.com/java-dijkstra** e reso generico per essere utilizzato con la classe Graph creata da Nediani.

_ Utilizzo di **System.currentTimeMillis()** per la gestione del tempo di attesa fra un attacco e l'altro.

### 3.3.3 Gian Luca Nediani

Aspetti avanzati del linguaggio Java di cui mi sono avvalso sono:

- *Optional*, per modellare elementi che possono essere presenti o meno all'interno delle stanze del gioco, come la moneta collezionabile o alcune delle porte.
- *Stream*, Lambda expressions e classi anonime per rendere, dove possibile, il codice più conciso e leggibile.
- *Generici*, per rendere i grafi e la loro esplorazione riutilizzabili per diversi aspetti del gioco. Tale scelta si è rivelata vincente in quanto i grafi, inizialmente concepiti solamente per modellare il layout dei livelli, sono stati invece utilizzati anche per lo sviluppo delle stanze e della logica dei nemici.

Ho inoltre implementato, previa consultazione di documentazione a riguardo(https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/), un algoritmo di esplorazione di gradi di tipo *Breadth First Search*. Questo algoritmo è stato utilizzato dal sottoscritto per assicurarmi che elementi fondamentali all'interno del gioco - come porte, stanze, chiavi - fossero sempre raggiungibili da ogni posizione che il giocatore può occupare.

### 3.3.4 Federico Pirazzoli

Come già descritto nella sezione precedente, ho fatto uso della libreria Slick2D (Javadoc: https://slick.ninjacave.com/javadoc/ Wiki: http://slick.ninjacave.com/wiki/index.php?title=Main_Page) e per capire al meglio il suo utilizzo ho sfruttato anche la serie di video indicata a seguito: https://www.youtube.com/watch?v=9dzhgsVaiSo.
Sono consapevole della disponibilità di librerie più avanzate per il tipo di lavoro che ci siamo posti, ma la libreria Slick è risultata essere quella che permetteva lo sviluppo del videogioco in maniera più semplice oltre a richiedere conoscenze pregresse minori rispetto ad altre come OpenGL.

Per sviluppare la maggior parte della classe LoadNatives ho cercato del codice che mi poteva essere utile per capire come coprire il warning descritto nel file README (https://stackoverflow.com/questions/46454995/how-to-hide-warning-illegal-reflective-access-in-java-9-without-jvm-argument), per auto-estrarre determinati file dal jar (https://stackoverflow.com/questions/1529611/how-to-write-a-java-program-which-can-extract-a-jar-file-and-store-its-data-in-s) e determinare il sistema operativo in uso (https://mkyong.com/java/how-to-detect-os-in-java-systemgetpropertyosname/).
Le "feature avanzate" di cui mi sono avvalso sono state principalmente:

- **Lambda expressions**, all'interno delle classi GameViewImpl e ModelCommunicatorImpl.
  In quest'ultimo caso, mi sono state utili nel richiamare i metodi di tutte le entità presenti all'interno delle stanze in maniera pulita e veloce mentre in quello precedente le ho utilizzate per richiamare la texture e coordinate delle entità da disegnare;

- **Streams**, all'interno delle classi GameViewImpl e ModelCommunicatorImpl.
  Nel primo caso, sono stati utilizzati nel metodo drawMod() per selezionare tutti i modificatori da disegnare mentre nel secondo caso sono presenti nei metodi checkEmpty() e getRoomID(), i quali vengono utilizzati rispettivamente per capire se una stanza sia collegata ad altre e verificare la presenza di una porta all'interno di essa e per ottenere il roomID corrispondente alla stanza collegata da una determinata porta.

# Capitolo 4, Commenti finali

## 4.1 Autovalutazione e lavori futuri

### 4.1.1 Lucia Fabbri

Non avendo mai avuto precedenti esperienze riguardanti il linguaggio Java e la progettazione in gruppo, devo dire che la collaborazione ed il confronto con i miei colleghi è stata di fondamentale importanza per affrontare problemi sia di natura progettuale ma anche motivazionale.
Non posso infatti nascondere che l'inizio è stato molto difficile. La poca esperienza alle spalle mi ha portato ad avere vari momenti di tensione e di frustrazione, aggravati dal pensiero di aver altri compiti da portare a termine in aggiunta alla singola implementazione del codice. L'indecisione su come partire e come gestire determinate situazioni mi ha certamente rallentata.
Nonostante queste prime difficoltà, lavorare ad un progetto di questa importanza mi ha permesso una crescita dal punto di vista personale ma soprattutto da un punto di vista lavorativo, simulando su piccola scala ciò che immagino si possa verificare in

un ambiente di lavoro.
Grazie alla scelta di lavorare su questa tipologia di progetto, ho avuto modo di conoscere sotto un nuovo punto di vista tutto ciò che sono le fondamenta per la realizzazione di videogiochi, sperimentando concretamente il lato progettuale e non fruirne solamente.
Per questo motivo è stato particolarmente entusiasmante vedere come, pezzo dopo pezzo, la nostra applicazione prendeva forma e andava sviluppandosi.
Inoltre, ho avuto modo di sperimentare ed utilizzare in maniera sempre più consapevole il linguaggio Java, la cui conoscenza e padronanza offre grandi potenzialità ed utilità in molti ambiti.

### 4.1.2 Marco Ragazzini

Sono complessivamente soddisfatto del progetto ultimato. L'obiettivo principale era quello di creare diversi tipi di nemici con funzionalità differenti per rendere l'esperienza di gioco più varia. A livello implementativo, la difficoltà maggiore è stata quella di realizzare le molte proposte che avevo esposto, cercando comunque di mantenere il codice il più pulito possibile A posteriori credo che sarebbe stato più pulito utilizzare una libreria esterna per la gestione dell'algoritmo di Dijkstra, per poi modificarlo e renderlo più generico per adattarlo al meglio alle esigenze del nostro programma. A prescindere dalle difficoltà riscontrate, in generale reputo questa esperienza estremamente positiva a prescindere dall'implementazione e dell'ottimizzazione del codice, in quanto mi ha dato la possibilità di lavorare in team, cosa che non era mai stata possibile e mi ha dato l'occasione di mettermi alla prova in nuovi ambiti che difficilmente si riescono ad affrontare con brevi esercitazioni. Inoltre, la fase implementativa, ha portato una maturazione delle mie competenze, che si è rispecchiata nella struttura del codice che ha subito diverse refactory per rendere il codice flessibile ed estendibile.

### 4.1.3 Gian Luca Nediani

Nel complesso mi ritengo soddisfatto della riuscita del progetto. Come gruppo, siamo riusciti ad implementare tutte le funzionalità minime concordate, ed entro le 80 ore di lavoro siamo riusciti anche a implementare parte delle funzionalità opzionali. La suddivisione dei compiti è stata rispettata e la comunicazione fra i membri, nonostante l'impossibilità di incontrarsi di persona a causa della attuale emergenza sanitaria, è stata buona, sia nel concepimento del progetto che nel suo sviluppo.

Anche per quanto riguarda il lavoro individuale, ritengo di aver soddisfatto i requisiti minimi decisi ad inizio progetto ed anche quello opzionali di generare i livelli in maniera casuale. Nonostante io sia riuscito sviluppare tutti gli aspetti richiesti, ritengo a posteriori di non aver fatto pieno uso di tutte le nozioni di programmazione a oggetti apprese a lezione e delle intere potenzialità del linguaggio Java. In particolare, riguardando il progetto ora che è completo, mi sono reso conto di non aver dato

sufficiente importanza ai pattern di programmazione visti in aula.

### 4.1.4 Federico Pirazzoli

Complessivamente sono soddisfatto del risultato finale che abbiamo ottenuto, considerando che l'unica esperienza precedente che ho avuto con il linguaggio Java è stata alle superiori.
La parte più difficile che ho riscontrato durante il lavoro è stata sicuramente capire il funzionamento della libreria esterna Slick. Nonostante sia stata generalmente molto utile, la documentazione, a mio avviso, è stata molto carente di informazioni e vaga in certi punti e non descriveva con rigore i vari metodi di cui disponeva.
Per questo motivo ho incontrato altre difficoltà nella realizzazione della parte grafica del progetto, in quanto ho dovuto analizzare e capire attentamente come la libreria reindirizzasse i vari componenti nella finestra, dove fosse il punto di origine degli elementi reindirizzati e altre situazioni simili. Nonostante tutto, sono riuscito nel mio intento di conferire una “forma” alle entità disegnate superando la semplice realizzazione tramite coordinate 64x64 in modo tale da poter effettuare tutto ciò che riguarda le interazioni con il resto del mondo e le altre entità.
Un ultimo aspetto che non avevo mai avuto l’occasione di sperimentare è stato l'esportazione del file jar eseguibile e, considerando la presenza di risorse e librerie esterne, si è rivelata essere un’operazione un po' più difficile del previsto.
Questo progetto mi ha dato la possibilità acquisire nuove conoscenze ed informazioni ma soprattutto di sperimentare in prima persona i vari processo necessari allo sviluppo di un progetto, fornendomi una minima familiarità utile qualora in futuro mi trovassi in situazioni simili.

# Appendice

## I Guida utente

All’avvio dell’applicazione viene mostrato il seguente menù principale


Le opzioni selezionabili sono le seguenti:
- “Start Game” per iniziare l’esplorazione
- “Quit Game” per abbandonare il gioco
- “How to Play” per conoscere tutte le regole di gioco

Per muovere il personaggio all’interno del mondo di gioco, si utilizzino i pulsanti ‘W’, ’A’, ’S’, ’D’ corrispondenti rispettivamente alle quattro direzioni alto, sinistra, basso e destra.

Per permettere al personaggio di compiere la sua azione di attacco utilizzare invece la barra spaziatrice.

Dopo l’avvio, è possibile premere il tasto “esc” in qualsiasi momento per mettere in pausa il gioco e riprendere l’esplorazione successivamente.

La seguente figura mostra una schermata di gioco della nostra applicazione: il nostro player si trova in alto al centro mentre i due nemici sono in movimento e stanno sparando lungo la loro traiettoria.
In alto a sinistra possiamo vedere la barra di vita del player e l'inventario con le monete collezionate, in basso a sinistra la chiave raffigurata servirà per aprire le porte della stanza mentre vari ostacoli sono presenti in maniera randomica all'interno della stanza.