**Progetto per il corso di OOP**

1. **A 2019/2020**

**Relazione per**

**“Just Another Rogue-Like Game”**

**Lucia Fabbri**

**Gian Luca Nediani**

**Marco Ragazzini**

**Federico Pirazzoli**

**Indice**

**1, Analisi**

* 1. Requisiti
  2. Analisi e modello del dominio
  3. Estrazione dei concetti principali

**2, Design**

* 1. Architettura
  2. Design dettagliato
     1. Lucia Fabbri
     2. Marco Ragazzini
     3. Gian Luca Nediani
     4. Federico Pirazzoli

**3, Sviluppo**

* 1. Testing Automatizzato
  2. Metodologia di lavoro
     1. Lucia Fabbri
     2. Marco Ragazzini
     3. Gian Luca Nediani

**3.2.4** Federico Pirazzoli

* 1. Note di sviluppo
     1. Lucia Fabbri
     2. Marco Ragazzini
     3. Gian Luca Nediani
     4. Federico Pirazzoli

**4, Commenti Finali**

* 1. Autovalutazione e lavori futuri

**4.1.1** Lucia Fabbri

* + 1. Marco Ragazzini
    2. Gian Luca Nediani

**4.1.4**  Federico Pirazzoli

**Appendice**

**I** Guida utente

**Capitolo 1, Analisi**

* 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 interagisce.

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.
* Realizzare una piccola interfaccia grafica per il giocatore dove vengono descritte le vari 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 quell’attuale.

Ciascuna stanza è composta da nemici, ostacoli e 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:

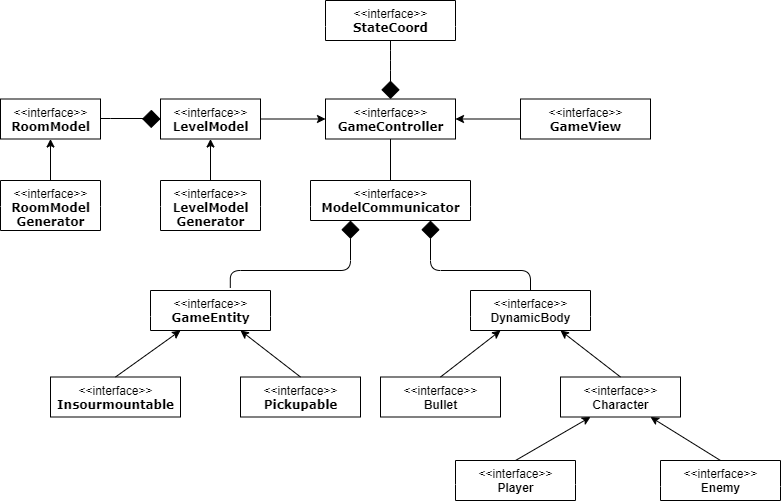


Figura 1.1: Schema UML dell’analisi dei requisiti indicante le principali entità e relazioni che comporranno l’applicazione

**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 e 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**

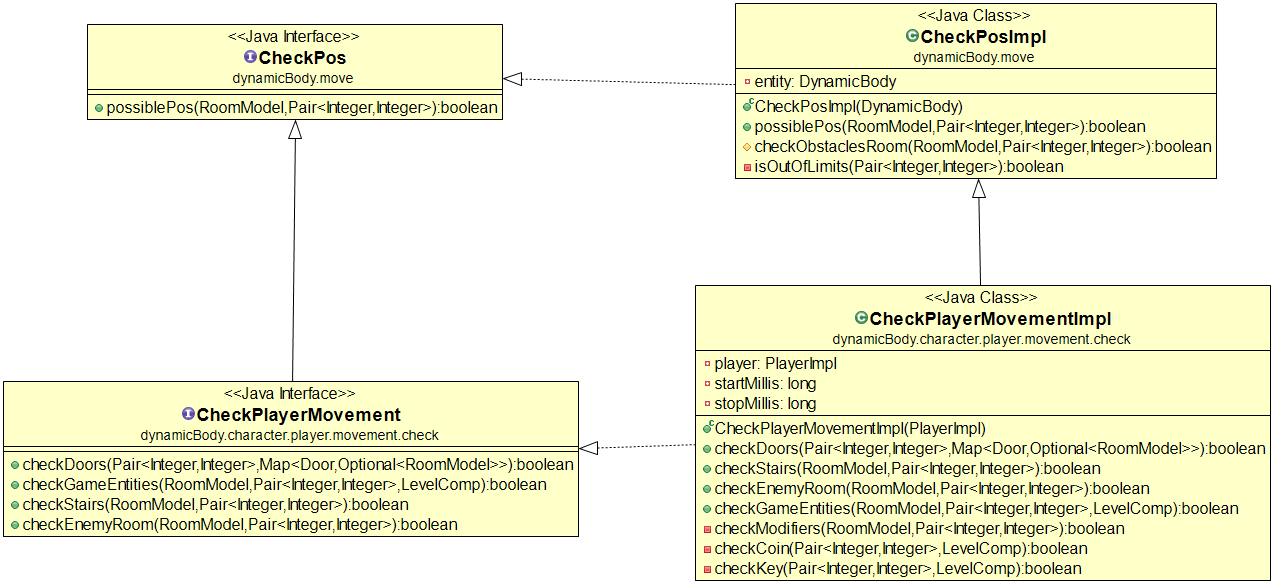
**2.2.1 Lucia Fabbri**

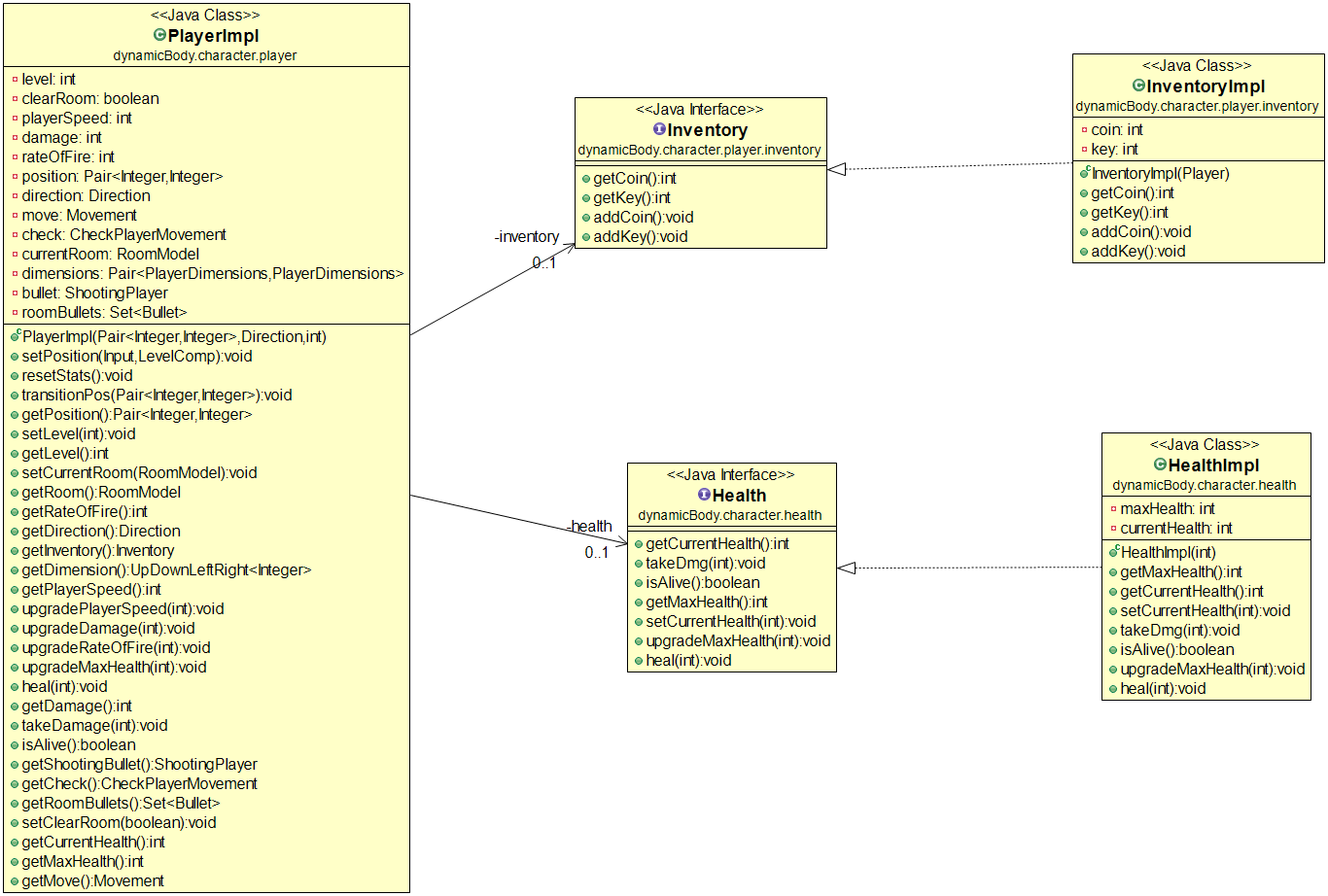
Il mio ruolo all’interno del progetto è stato quello di occuparmi della realizzazione del player e di tutto ciò che riguarda la sua interazione con l’ambiente di gioco.  
Ho racchiuso le sue principali funzionalità all’interno della classe *PlayerImpl* che implementa l’interfaccia *Player* ed altre proprietà 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.

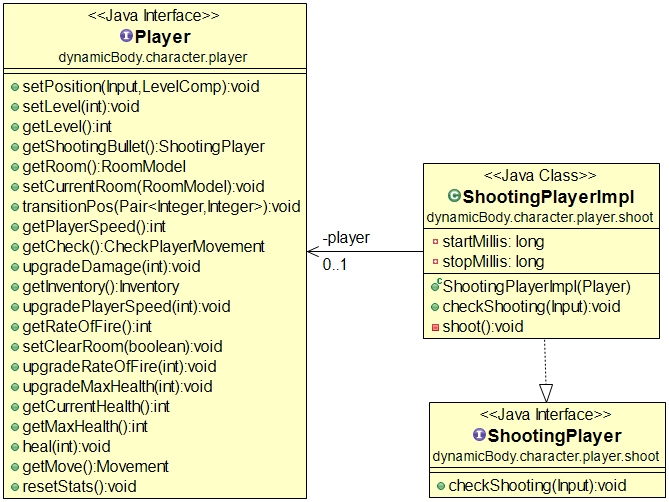
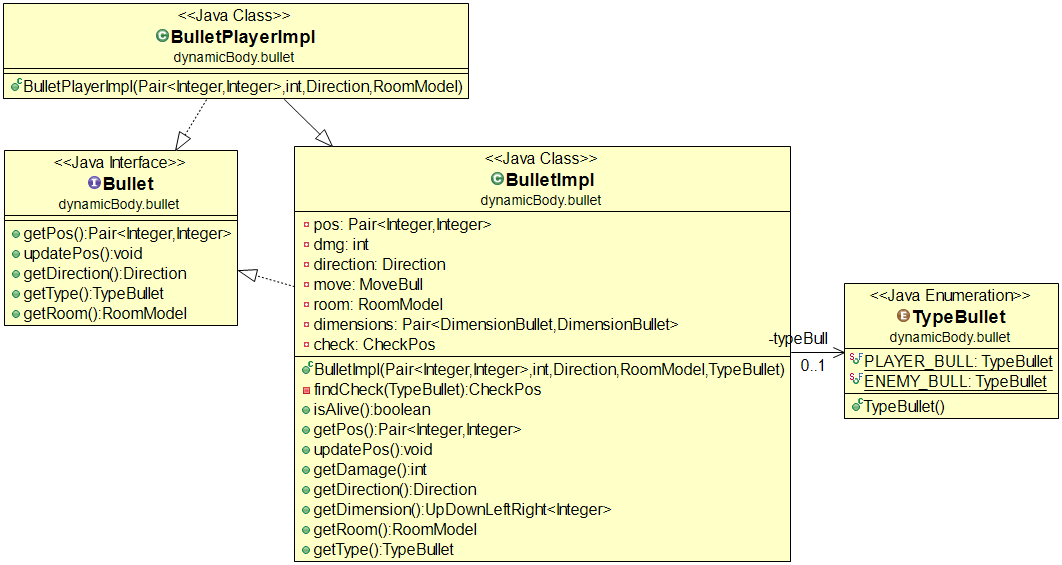
  
Figura 2.1: Struttura di interconnessione interfacce comuni

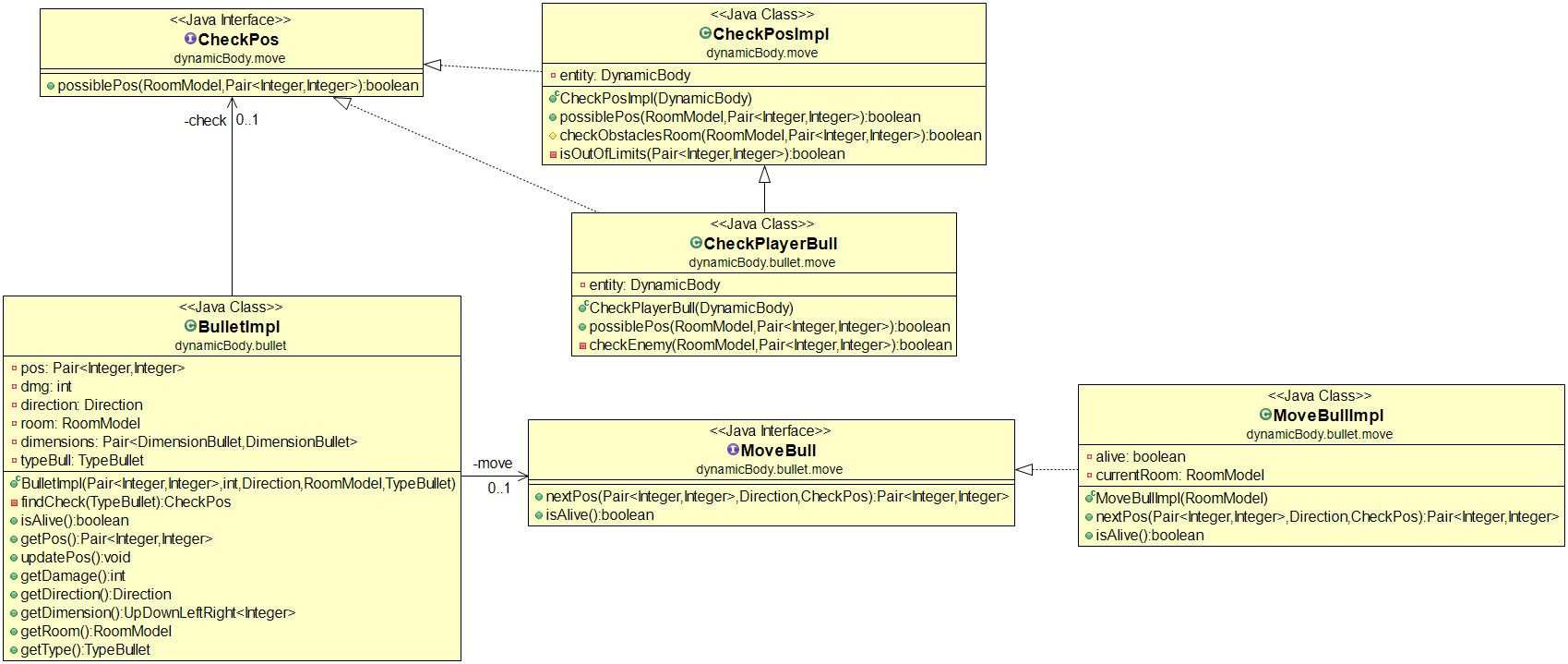
Per quanto riguarda il player, la sua funzionalità principale è quella di muoversi liberamente all’interno della finestra di gioco. Utilizzando la libreria esterna Slick2D(<http://slick.ninjacave.com/javadoc/>) per acquisire l’input da tastiera, ogni volta che viene premuto un pulsante ne effettuo il 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 Figura 2.5: Struttura creazione proiettile del player  
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.

* + 1. **Marco Ragazzini**

La parte di progetto a me assegnata riguarda l’implementazione dei nemici e tutto l’interazione che questi hanno con l’ambiente di gioco.

Dentro l’interfaccia *Enemy* ho racchiuso le principali funzionalità dei nemici. Tale interfaccia è implementata da *EnemyImpl*, utilizzata per la gestione dei nemici base, e da *BossImpl*, utilizzata per la gestione del Boss.

In collaborazione con la mia collega Lucia abbiamo deciso di cercare di minimizzare il più possibile le ripetizioni di codice, creando 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 Enemy dovrà essere in grado di potersi muoversi in ogni posizione viabile. Per gestire il movimento ho deciso di utilizzare un *Pair* di Integer, dove vengono salvate le coordinate del pixel in alto a sinistra del nemico.

La vita del mostro è gestita attraverso l’interfaccia *Health*, comune con il Player, con all’interno un metodo per diminuire la vita ed un metodo per calcolare se il personaggio sia ancora in vita o meno.

Ogni Enemy potrà avere differenti tipi di attacchi, movimenti e dimensioni gestiti con delle Factory. Quest’ultima è gestita grazie ad una classe Generica di nome *UpDownLeftRight*, il cui compito è quello di salvare al suo interno quattro valori e, utilizzando gli appositi get, ritornarli quando necessario.

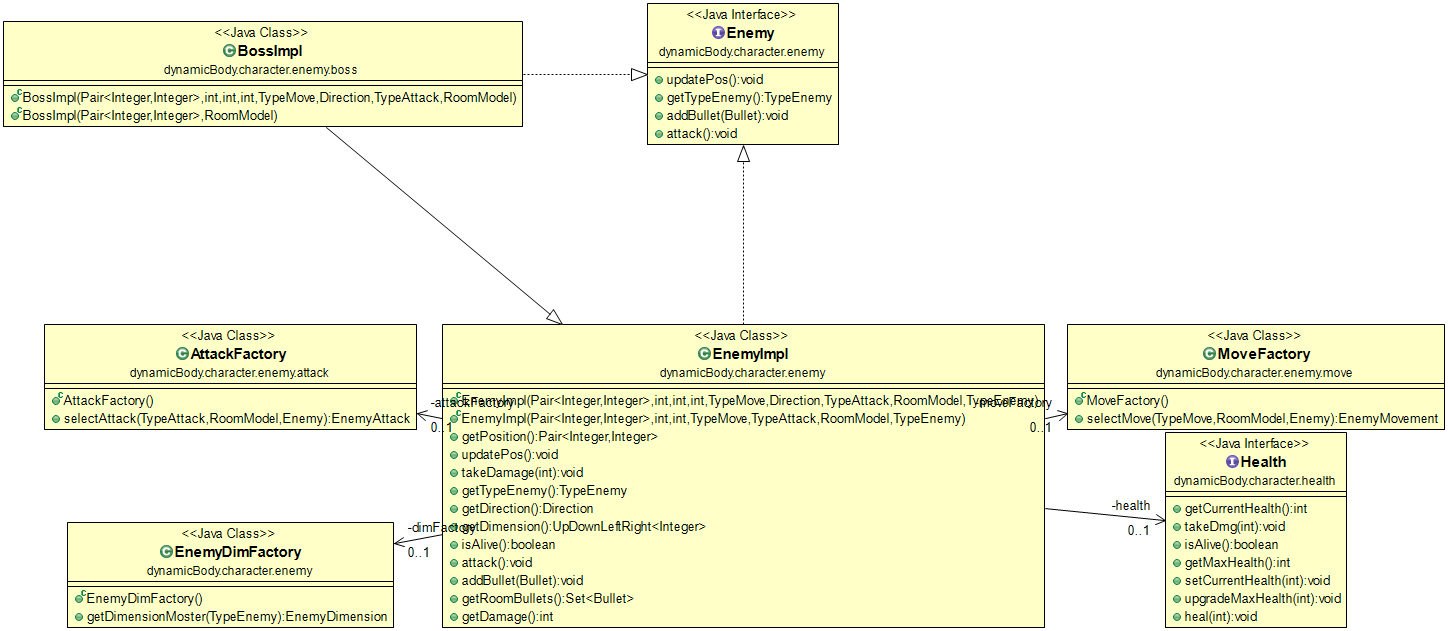


Figura 2.2: Struttura della Gestione dei vari nemici e delle Factory

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, ed un metodo per ricavare la nuova direzioni. Questa interfaccia è implementata da cinque classi, una per ogni 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 e ricomincerà da capo.
* *TeleportMove*: permette al nemico, ogni cinque secondi, di teletrasportarsi in una posizione casuale nella stanza, purché non sia occupata da un ostacolo, per poi rimanere immobile per i secondi rimanenti.
* *RandomMove*: permette al nemico di muoversi casualmente. Dopo aver compiuto 250 più un numero casuale fra 0 e 130 di passi o aver colliso con un muro od ostacolo, di cambiare la direzione in una casuale fra quelle possibili, e muoversi diritto finché non bisogni ripetere il cambiamento direzioni.
* *ImmobilizedMove*: non permette al nemico di muoversi, ma soltanto di rimanere immobile 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 percorso 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 tipo di valore dell’enumerazione *TypeMove*, ritornerà l’oggetto corretto.

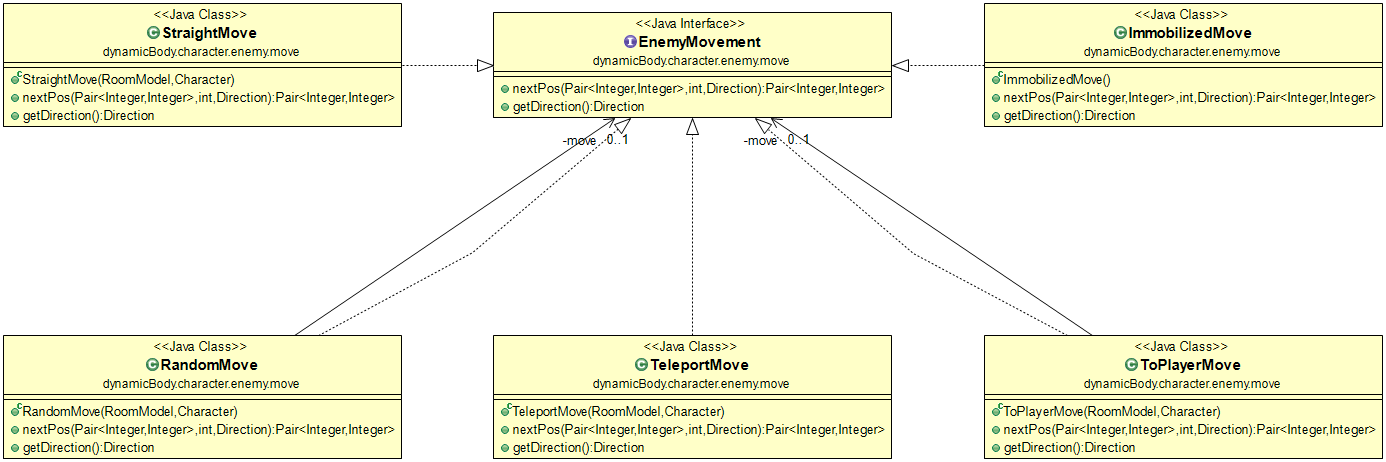


Figura 2.3: Struttura dei vari movimenti del Nemico

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 che, dopo la collisione del nemico con un muro o un ostacolo, cambia la direzione del nemico. Dentro *CheckPos* vi è il metodo *possiblePos*, il cui scopo è quello di ritornare un boolean true, se la posizione è possibile, false altrimenti. Tale controllo si articola in due funzioni, la prima verifica che il personaggio non superi i limiti, mentre la seconda che non coincida con un ostacolo.

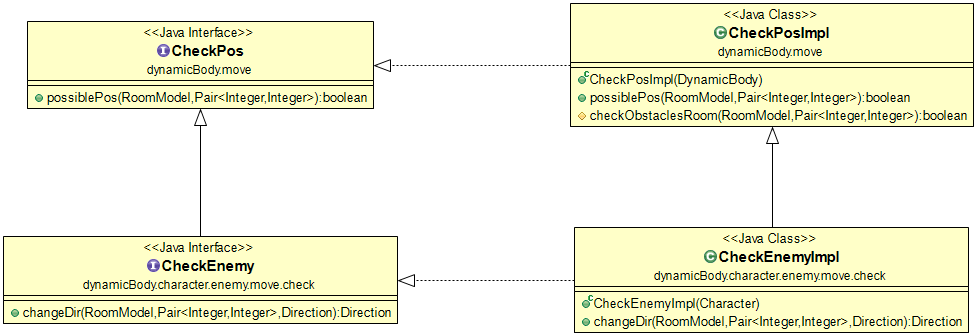


Figura 2.4: Struttura del controllo del nemico all’interno del mondo di gioco

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, nei lati adiacenti a quello in cui sta guardando il nemico. Il movimento di questi due proiettili sarà diagonale.
* *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 un proiettile per ogni lato.

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

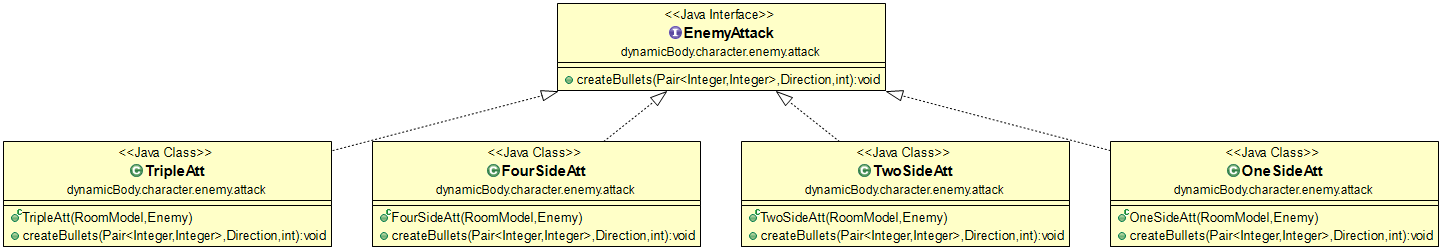


Figura 2.5: Struttura dei vari tipi di attacco del nemico

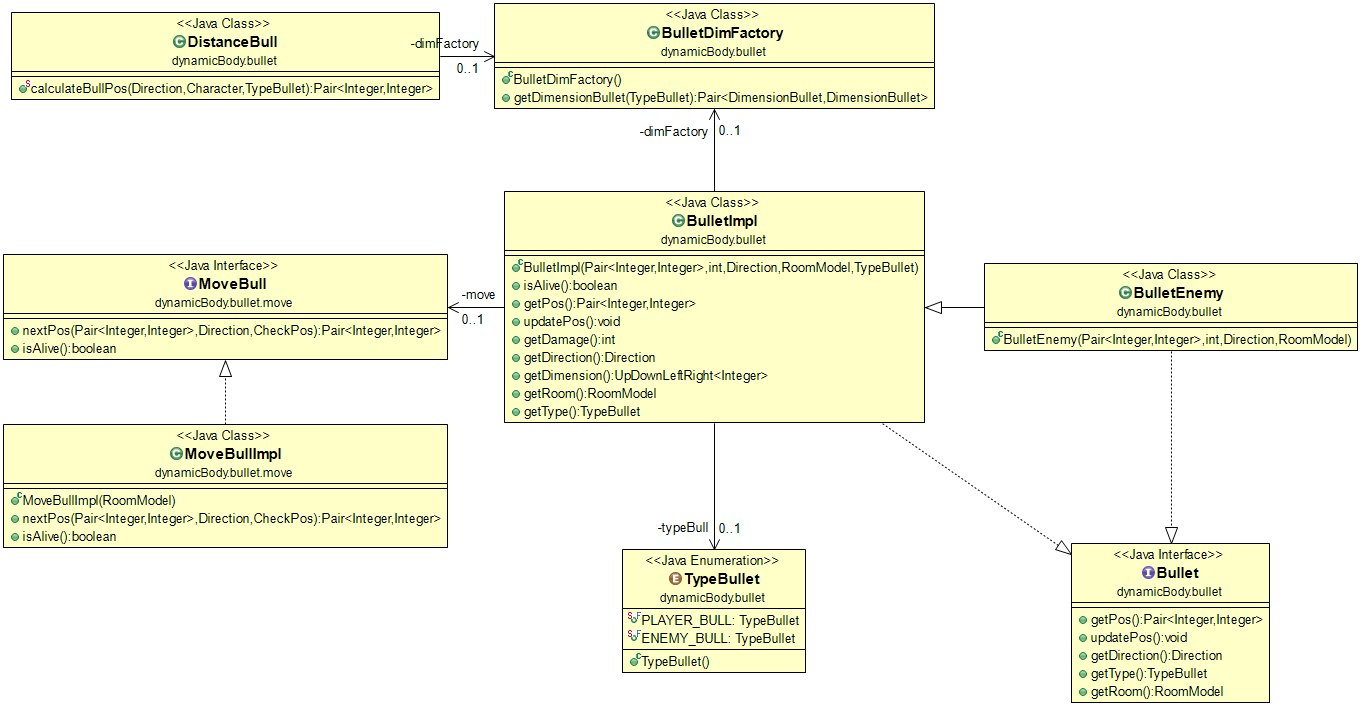
*Bullet* è l’interfaccia che racchiude tutte le funzionalità dei proiettili. Tale interfaccia è implementata dalla classe *BulletImpl*. *Bullet*, estendendo *DynamicBody*, avrà i metodi per gestire le dimensioni, i danni e il controllo se l’oggetto sia ancora in vita, ovvero se non sia entrato in contatto con altre entità di gioco. Per il movimento vi è l’apposito metodo, *updatePos*, che ritorna le coordinate successive del proiettile. Attraverso l’enumerazione *TypeBullet*, vi è la possibilità di sapere se il proiettile è stato lanciato da un 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 valori standard del proiettile. Per evitare che i proiettili venissero creati sovrapposti al personaggio, ho creato una classe statica, con costruttore privato, così che non potesse essere inizializzata, che, in base alla direzione, calcolasse le coordinate per la creazione del proiettile. Per semplificare la creazione dell’oggetto ho deciso di creare una classe *BulletEnemy*, ed attraverso un super costruttore non fosse necessario specificare che il tipo di proiettile fosse del Nemico, cercando anche di evitare errori con il 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 entra in contatto con il bordo della mappa o con un ostacolo, controlla anche se colpisce il personaggio. 

Figura 2.6: Struttura dei proiettili

Dentro l’interfaccia *Enemy* possiamo trovare due metodi che lavorano più o meno nello stesso modo, ma con funzione molto diverse. Uno, chiamandolo, permette al nemico di aggiornare la propria posizione con quella successiva, simulando un movimento, l’altro permette di attaccare, ovvero di creare i 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.

* + 1. **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:

Immagine che contiene screenshot, mappa

Descrizione generata automaticamente

L’interfaccia *LevelModel* definisce le operazioni principali su un livello di gioco, mentre la sua implementazione contiene le strutture dati necessarie alla modellazione: un è composto da 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:

Immagine che contiene mappa, screenshot

Descrizione generata automaticamente

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 4 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 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 ad B tramite la stanza SUD.

Immagine che contiene mappa, testo

Descrizione generata automaticamente

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 4 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.

* + 1. **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 nel mondo stesso, modellate e implementate dai miei colleghi, il quale si è provata una mansione non facile in quanto la decisione di adottare una libreria esterna (Slick2D) completamente nuova e dovendo studiare la sua documentazione o reperire informazioni per capire il suo preciso funzionamento non è stata una passeggiata, tutto qusto per avere già una base concreta su cui lavorare e un modello designato apposta per creare giochi; altri ruoli secondari sono stati la creazioni di oggetti e modificatori, e permetter al file progetto di essere eseguibile su più sistema operativi senza restrizioni, permettendo una standardizzazione del file jar consegnato.

Andando più nello specifico le mansioni principali che ho svolto sono state:

* Riuscire a modellare e implementare diversi aspetti funzionali di base come la rappresentazione (o rendering) degli elementi grafici del gioco, e gli aspetti riguardanti la logica (o appunto logic);
* Implementare un modo semplice per modellare e richiamare i componenti del mondo di gioco stesso;
* Modellare un sistema di conservazione dei vari elementi presenti all'interno di ogni stanza di ogni livello, in modo tale da rendere l'esecuzione della logica e la renderizzazione della grafica più semplice;
* Modellare e disegnare un sistema di menù e elementi grafici, ovvero una GUI, per rendere l'esperienza di gioco migliore;
* Stabilire un'implementazione per permettere al gioco di passare da un livello (o “state” come sono implementate all'interno del progetto) senza problemi o rallentamenti, conservando determinati elementi da livello a livello (principalmente legati al personaggio principale);
* Permettere al software di essere eseguito su sistemi Linux e MacIOS, e non solo Windows
* Opzionalmente per decisione volontaria ho voluto anche implementare suoni, musiche e grafica creata principalmente da me per rendere il progetto in sé più vivo e realistico, ovviamente mansioni non fondamentali al compimento del progetto;

La prima fase di sviluppo, dopo esserci divisi i vari compiti, nel mio caso è stata principalmente sul fare test per capire come la libreria Slick2D potesse essere utilizzata nel modo più efficiente e sensato per soddisfare i nostri bisogni, la quale ci ha permesso di utilizzare un sistema a “state” o stati per identificare vari componenti del gioco, come il menù e la classe che gestisce i livelli, dove è presente il “cuore” del gioco; abbiamo una classe (GameController) che estende il BasicGameState da Slick, la quale ha 3 metodi obbligatori: Init, utilizzata per caricare, prima di qualsiasi altra azione, le risorse necessarie al programma per esser eseguito; Render, che oltre ad avere riferimenti alla classe grafica del contesto OpenGL che si crea e quindi permettere di reindirizzare elementi grafici nella finestra, contiene tutti i riferimenti a elementi utilizzati nella reindirizzazione di poligoni, immagini o elementi grafici all'interno del programma; e infine Update, che similmente a Render, deve contenere tutti i riferimenti agli update, o modifiche, degli elementi del gioco, dal mondo stesso alle entità che sono presenti; il GameController in sé è chiamata dalla classe StateCoord, che si occupa inizialmente di chiamare LoadNatives per caricare le librerie necessarie ad eseguire il programma, per poi inizializzare l'entità Player e passare il controllo alla classe Menu. Menu estende, come GameController, BasicGameState per creare una videata iniziale dove poter scegliere di iniziare il gioco, uscire dall'applicazione oppure leggere un breve tutorial, dividendo l'aspetto grafico in Menu, e la logica dei bottoni dentro MenuLogic.

Ciò che rende interessante questo tipo tipo di modello mi ha permesso di dividere in maniera pulita ed efficiente le varie attività compiute dalle entità e dal mondo di gioco, dividendo ulteriormente quest'ultimi in altre due interfacce GameView e ModelCommunicator per permettere di avere una chiara distinzioni di quali elementi compiessero cosa, tutti contenuti nel package Coordination perché appunto tutti gli aspetti del gioco vengono coordinati dentro queste classi.

Abbiamo oltretutto 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, il Player;
* EntityImage e EntityImageFactory, per conservare le texture degli oggetti e potenziamenti del gioco, come gli items e modifiers;
* Infine SoundBoard e SoundBoardFactory, per conservare gli elementi audio di tutti gli eventi che ne richiedeno, come l'essere colpiti da un proiettile nemico, o l'apertura delle porte.

Ho anche 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 di poter riutilizzare le texture dei muri, porte e pavimento per formare nuove stanze.

Tutto questo per permettere al codice dei miei colleghi di essere liberi dalle dipendenze di slick se non per il lancio di eccezioni,

Inizialmente ho avuto dei problemi a capire come esattamente la libreria interpretava l'asse cartesiano della finestra, dove l'origine è nell angolo in alto a sinistra, e le coordinate che rappresentano la lunghezza e altezza della finestra totali è nell angolo in basso a destra. Un altro dei motivi principali nella scelta di Slick era anche nella facilità di rappresentare elementi grafici da cui ho preso spunto da altri Game Engines, chiamati Tiles, ovvero generalmente si parla di “tasselli” di grandezza predefinita (nel nostro caso 64 x 64 pixels), che vengono utilizzate per disegnare tutto ciò di cui si ha bisogno sullo schermo, in modo tale da tenere tutto proporzionate senza rischio di textures più allargate o ristrette del dovuto, difatti attraverso l'utilizzo dei Tiles a 64 pixels siamo riusciti a anche a “modellare” la griglia di gioco delle entità, utilizzando multipli di 64 per costruire la base del mondo di gioco (pavimento, muri, porte) senza troppe difficoltà.

L'idea che ho avuto per contenere tutte le informazioni e dati di ogni stanza dei livelli è stata attraverso la creazione di un'interfaccia LevelComp, la quale permette di creare 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, porte e pavimento, ai nemici, ostacoli, e oggetti presenti; il tutto costruito attraverso l'utilizzo di un metodo creato dal mio collega Nediani che crea in maniera “randomica” dei livelli contenenti delle stanze rappresentate da grafi; tutto questo permette alla classe ModelCommunicator di richiamare le API delle entità create da Fabbri e Ragazzini molto facilmente e velocemente.

La GUI e menù invece sono semplicemente creati utilizzando, come menzionato prima, il contesto Graphics offerto dalla funzione render, e implementata nella classe UI , traendo dagli altri oggetti stanziati come il livello o il Player gli elementi necessari per disegnare elementi dinamici, lasciando la logica di bottoni o simili alla classe UILogic.

Gli oggetti e potenziamenti sono caratterizzati dal fatto che estendono una classe del mio collega Nediani: Pickupable; come la parola suggerisce, sono tutte entità che il Player può raccogliere per andare avanti nel gioco, migliorare le proprie abilità o per collezione. La loro posizione, e il loro tipo, è sempre randomico stanza per stanza.

Per quanto riguarda la compatibilità con altri sistemi operativi invece, 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 in modo dal Classpath in modo tale che, indipendentemente dal contesto, 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 perlopiù 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.

**3.3 Note di sviluppo**

**3.3.1 Lucia Fabbri**

Nello sviluppare il progetto ho sempre cercato di:

- rispettare le convenzioni del linguaggio Java e 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 il movimento del player e per 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**](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, 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 detto precedentemente nella sezione, si è fatto uso della libreria Slick2D (Javadoc: <https://slick.ninjacave.com/javadoc/> Wiki: <http://slick.ninjacave.com/wiki/index.php?title=Main_Page>) altre fonti che ho usato per capire meglio il suo utilizzo è stata questa serie di video: <https://www.youtube.com/watch?v=9dzhgsVaiSo> la quale spiega svariati utilizzi della libreria. Ci sono sicuramente librerie più “avanzate” per il tipo di lavoro che ci siamo posti, ma Slick era quella che richiedeva meno conoscenza di altri librerie, come OpenGL, oltre che per il fatto che permette uno sviluppo di videogiochi un po' più “semplice”.

Le “feature avanzate” di cui ho fatto utilizzo sono state principalmente:

**Lambda expressions**, all'interno delle classi GameViewImpl e ModelCommunicatorImpl, dove nel secondo caso mi sono state utili nel richiamare i metodi di tutte le entità presenti all'interno delle stanze in maniera pulita e veloce, e nel primo per richiamare la texture e coordinate delle entità da disegnare;

**Streams**, sempre all'interno delle classi GameViewImpl e ModelCommunicatorImpl, nel primo caso nel metodo drawMod() per selezionare tutte le entità da disegnare, ovvero i modificatori, nel secondo caso invece nei metodi checkEmpty() e getRoomID per, rispettivamente, capire se una stanza è collegata ad altre stanze e di conseguenza verificare se è presenta una porta, e per ottenere il roomID corrispondente alla stanza collegata da una determinata porta.

**Capitolo 4, Commenti finali**

**4.1 Autovalutazione e lavori futuri**

**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.

