Relazione Programmazione ad Oggetti

Alberto Di Girolamo, Davide Di Marco,
Nicolò Malucelli, Filippo Venturini

25 Aprile 2021

Indice

1 Analisi **2**
1.1 Requisiti 2
1.2 Analisi e modello del dominio 4

2 Design **6**
2.1 Architettura 6
2.2 Design dettagliato 8

3 Sviluppo **21**
3.1 Testing automatizzato 21
3.2 Metodologia di lavoro 23
3.3 Note di sviluppo 26

4 Commenti finali **27**
4.1 Autovalutazione e lavori futuri 27

A Guida utente **29**

B Esercitazioni di laboratorio **32**
B.0.1 Filippo Venturini 32
B.0.2 Alberto Di Girolamo 32

Capitolo 1

Analisi

1.1 Requisiti

L'applicazione MazeDungeon emula il videogioco roguelike "The Binding of Isaac" realizzandone le dinamiche principali. Il gioco è costituito da un labirinto di stanze in cui il personaggio principale si muove, affrontando i nemici con attacchi a distanza, raccogliendo le monete che essi rilasciano alla loro morte e acquistando degli oggetti che gli conferiscono dei bonus.

Requisiti funzionali

- Il videogioco si compone di un menù principale, in cui sarà possibile iniziare una nuova partita. All'avvio di essa verrà generato un labirinto di stanze casuale ed inizialmente il personaggio principale si troverà in una di esse, vuota. Ogni stanza sarà dotata di una o più porte (fino ad un massimo di 4) in base alla struttura della mappa, permettendo l'accesso ad altre aree del labirinto. Ogni volta che il personaggio principale attraversa una porta, spostandosi in un'altra stanza, verranno generati dei nemici di vario tipo, con cui dovrà combattere. Una volta sconfitto un nemico, esso rilascerà una moneta che potrà essere raccolta ed utilizzata in seguito. Sarà presente un menù di gioco, in cui sarà contenuto uno shop, grazie al quale si potranno acquistare degli item (oggetti) per potenziare le proprie statistiche (velocità di attacco, danno inflitto, velocità di movimento, ripristino salute). Durante il combattimento non sarà possibile attraversare alcuna porta o aprire il menù per l'acquisto degli item. In una stanza casuale ci si potrà imbattere nel Boss, un nemico più difficile da sconfiggere, avente più vita di un nemico semplice, una sua routine di movimenti e una sua serie di attacchi personalizzata.

- Ogni volta che verrà iniziata una nuova partita, sarà generata una mappa, la cui disposizione di stanze verrà assemblata casualmente. Ogni stanza al suo interno può contenere da 2 a 4 nemici la cui tipologia sarà determinata casualmente e una configurazione di ostacoli (anch'essa casuale), che permetteranno al personaggio principale di non essere colpito, ma renderanno anche più difficoltoso colpire i nemici.

- Il personaggio principale sarà dotato di una vita, che se terminata durante il combattimento con i nemici incontrati, porterà alla sconfitta.

- Una volta eliminati tutti i nemici e scoperte tutte le stanze, verrà generato un artefatto finale nella stanza di spawn (iniziale), che se raccolto identifica la fine del gioco con la vittoria da parte dell'utente.

- Sarà presente un HUD (heads-up display) ovvero un insieme di informazioni permanentemente presenti nella schermata di gioco, che mostrerà all'utente: la vita del personaggio, il numero di monete raccolte e gli item posseduti.

- Nello shop sarà presente un elenco di item acquistabili una sola volta, con il rispettivo prezzo e il bonus che essi conferiscono. Una volta acquistato, un item applica automaticamente ed istantaneamente il potenziamento delle statistiche al personaggio principale. Sarà possibile inoltre acquistare un item consumabile che ripristina parte della vita persa durante i combattimenti.

Requisiti non funzionali

- MazeDungeon dovrà risultare graficamente fluido, evitando rallentamenti nella presentazione delle immagini o addirittura blocchi, nonostante l'elevata presenza di entità grafiche.

1.2 Analisi e modello del dominio

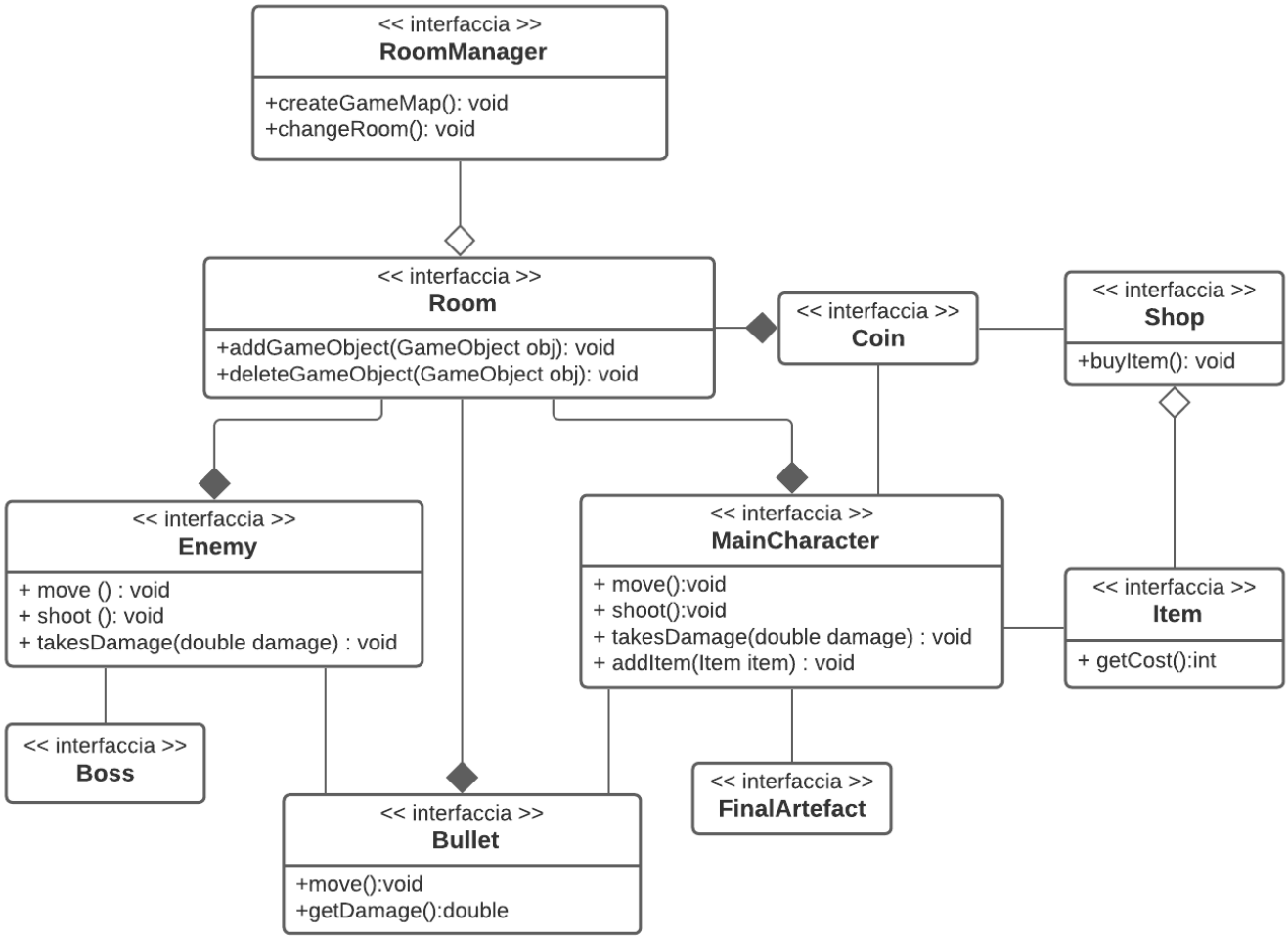


Figura 1.1: Schema UML che rappresenta il dominio applicativo

Il modello del dominio applicativo di MazeDungeon è strutturato come rappresentato in figura 1.1. Sarà presente un gestore di stanze, che si occuperà di creare il contesto di gioco, esso avrà il compito di creare le stanze del labirinto e i relativi collegamenti tra di esse. Ogni stanza conterrà al proprio interno gli elementi di gioco principali: personaggio principale, nemici, ostacoli, monete, proiettili.

All'interno della stanza corrente i nemici si muoveranno e, sparando proiettili, cercheranno di colpire il personaggio principale. Essi avranno una determinata velocità di movimento e una propria vita al termine della quale moriranno, rilasciando delle monete. Saranno presenti varie categorie di proiettili, che si differenzieranno per velocità e danno inflitto.

In una stanza casuale si troverà il Boss, all'interno di essa non saranno presenti ulteriori nemici ma nemmeno ostacoli, rendendo così il combattimento più impegnativo.

Dopo la scoperta di tutte le stanze presenti verrà generato un artefatto finale che se raccolto porterà alla conclusione della partita con la vittoria dell'utente. Il personaggio principale si muoverà comandato dall'utente, sarà in grado di sparare proiettili verso i nemici, raccogliere ed accumulare le monete rilasciate da essi, e come i nemici subirà danno se colpito dai proiettili. La morte del personaggio porterà alla conclusione della partita con la sconfitta dell'utente.

Sarà possibile accedere ad un negozio in cui, con le monete accumulate, si potranno acquistare oggetti che influiranno positivamente sulle statistiche del personaggio, come: velocità di attacco, velocità di movimento, danno inflitto, vita.
Una delle difficoltà principali sarà quella di gestire le collisioni tra le varie entità precedentemente descritte, esse saranno il fulcro centrale che permetterà l'evoluzione del gioco.

Capitolo 2

Design

2.1 Architettura

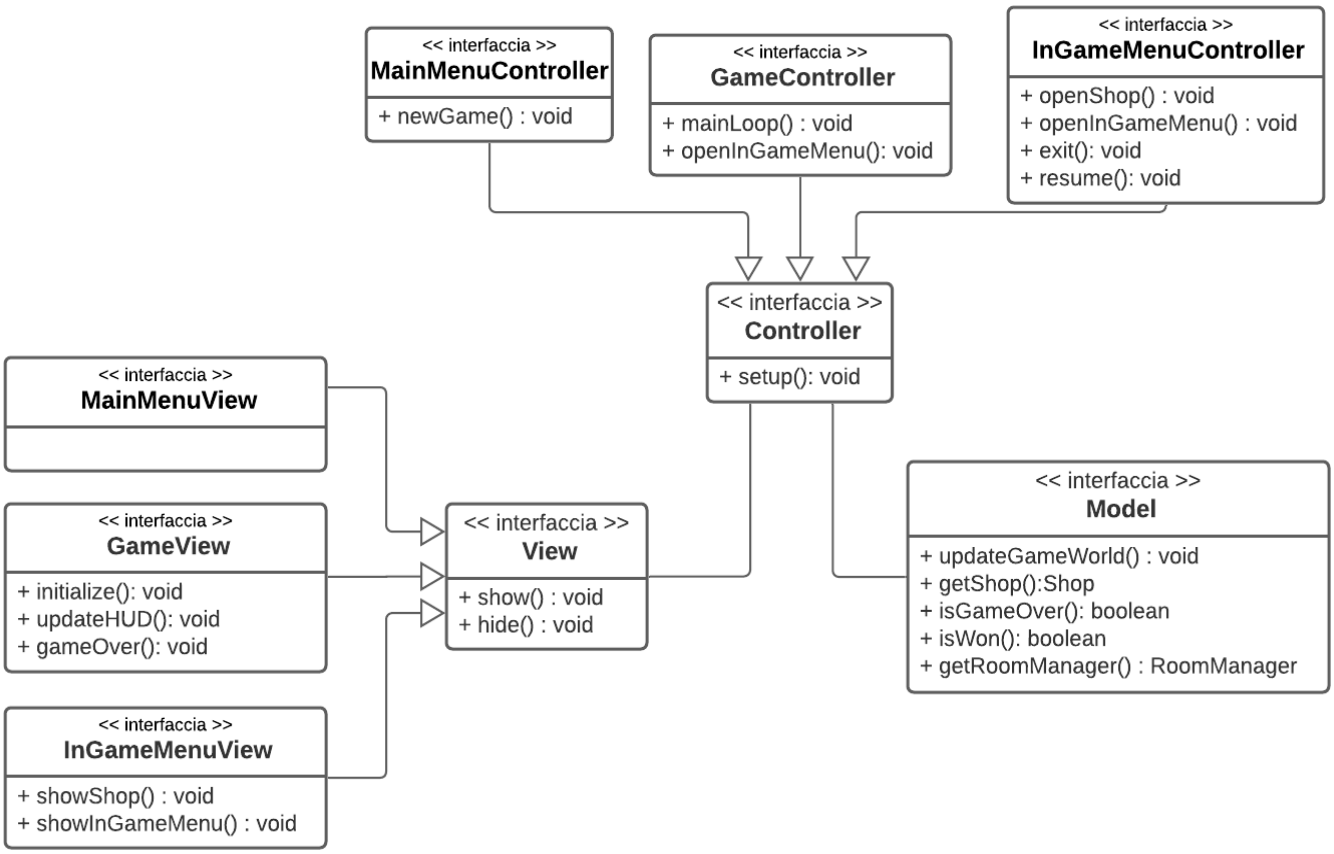


Figura 2.1: UML rappresentante l'architettura dell'applicazione

MazeDungeon segue il pattern architetturale MVC (figura 2.1). In particolare è composto da tre `View`: di gioco, del menu principale e del menu in-game. Ogni `View` rappresenta graficamente uno stato possibile dell'applicazione, ad ognuna di esse è associato un `Controller` che coordinerà adeguatamente l'interazione con il `Model`.

All'avvio dell'applicazione il controller attivo sarà il `MainMenuController`, il quale si occuperà di far visualizzare la `MainMenuView` a schermo e permetterà di iniziare una nuova partita.

All'inizio di essa il controllo dell'applicazione verrà passato al `GameController` che mostrerà di conseguenza la `GameView`.
Il `GameController` rappresenta il fulcro dell'applicazione, in quanto contiene al proprio interno il ciclo principale che identifica l'avanzamento del gioco (Game Loop).

Durante il gioco, sarà possibile accedere al menu in-game e quindi allo shop. In tal caso il controllo verrà assunto dal `InGameMenuController`, che come i due precedenti mostrerà la relativa `InGameMenuView`.

L'evoluzione logica del mondo di gioco, ovvero quella che contiene le entità descritte in fase di analisi, è parte del `Model`.
Essendoci più `Controller` che si occuperanno di coordinare `View` differenti e che faranno riferimento a stati dell'applicazione diversi, il `Model` fornirà tutte le informazioni logiche necessarie per permettere l'aggiornamento delle interfacce grafiche.

Con la progettazione architetturale precedentemente descritta, la tipologia di tecnologia scelta per l'implementazione delle varie `View`, non impatta in alcun modo sui vari `Controller` ne tantomeno sul `Model`. Sarebbe infatti possibile realizzare ogni `View` con una tecnologia differente, senza modificare il funzionamento della parte logica.

2.2 Design dettagliato

In questa sezione sono illustrati alcuni elementi di design con maggior dettaglio. Ogni membro del gruppo presenta i componenti più significativi da lui concepiti, al fine di realizzare l'applicazione, predisponendola a future espansioni.

Filippo Venturini

- **Gerarchia GameObject**

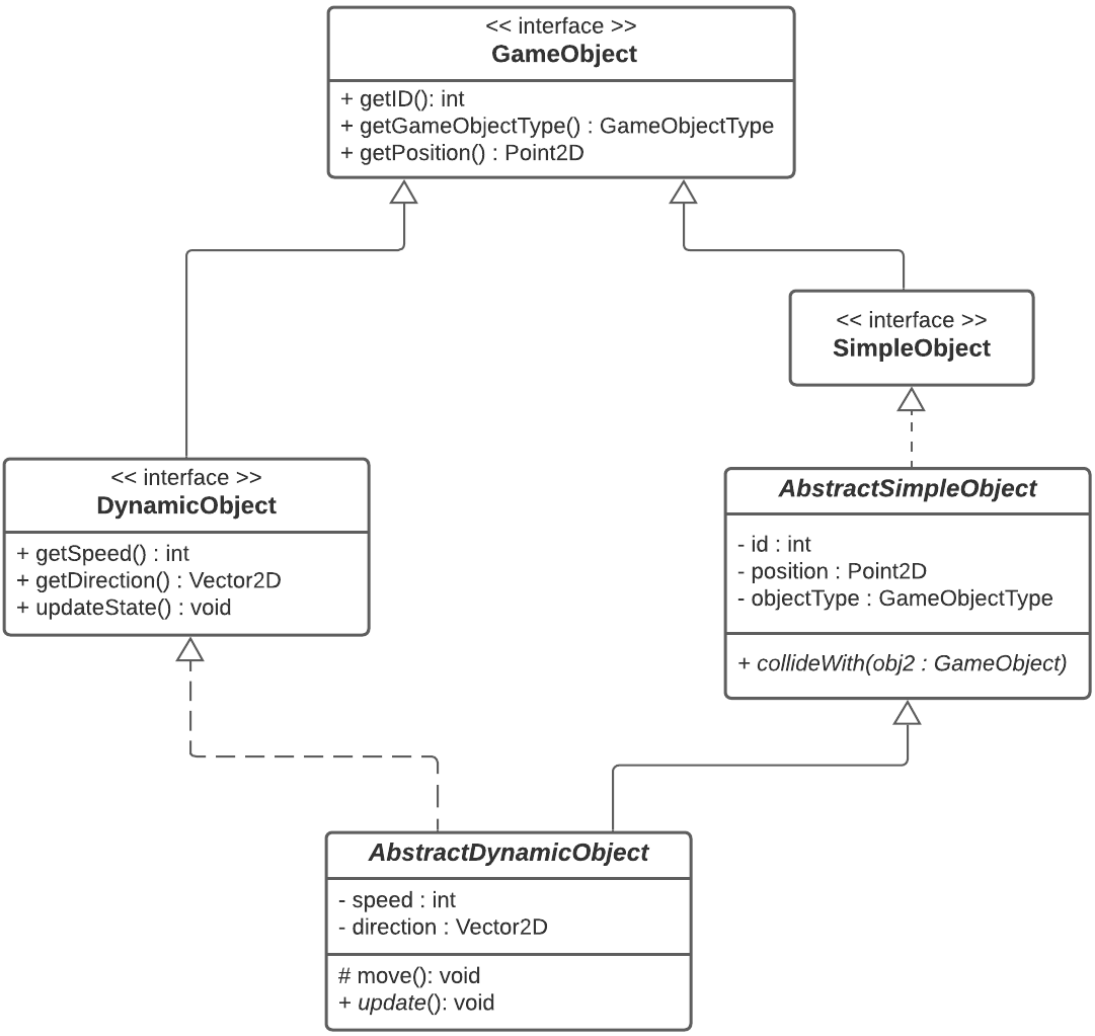


Figura 2.2: Schema UML rappresentante la gerarchia dei GameObject

Tutti gli oggetti che si trovano all'interno di una `Room` sono definiti `GameObject`, essi hanno un ID che li identifica univocamente e una posizione nell'ambiente di gioco.
Scendendo con un approccio top-down nello schema UML in figura 2.2, vi è una ramificazione che distingue i `SimpleObject` dai `DynamicObject`, i primi sono gli oggetti semplici (o statici) che non effettuano alcun tipo di movimento o azione dinamica (monete, sassi, muri, porte), gli altri invece sono tutti gli oggetti che con l'avanzare del gioco si spostano con una direzione e ad una certa velocità, modificando il loro stato (personaggio principale, nemici, proiettili).
Concettualmente gli oggetti dinamici sono degli oggetti semplici con caratteristiche aggiuntive (definite dall' interfaccia `DynamicObject`), che garantiscono la loro dinamicità.
Di conseguenza `AbstractDynamicObject` estende `AbstractSimple-Object` e implementa l'interfaccia `DynamicObject`.
Entrambe le implementazioni sono realizzate con una classe astratta per permettere una gestione delle collisioni personalizzata, e nel caso dei `DynamicObject` un aggiornamento ad-hoc delle azioni compiute da essi (movimento, sparo ecc.), ciò grazie all'utilizzo dei metodi astratti `collideWith()` e `update()`.
Questa gerarchia permette facilmente di creare nuove entità di gioco, siano esse statiche o dinamiche e di definire il loro comportamento a seconda delle necessità, favorisce quindi l'eventuale espansione futura del mondo di gioco.

- **Template Method in AbstractEnemy**

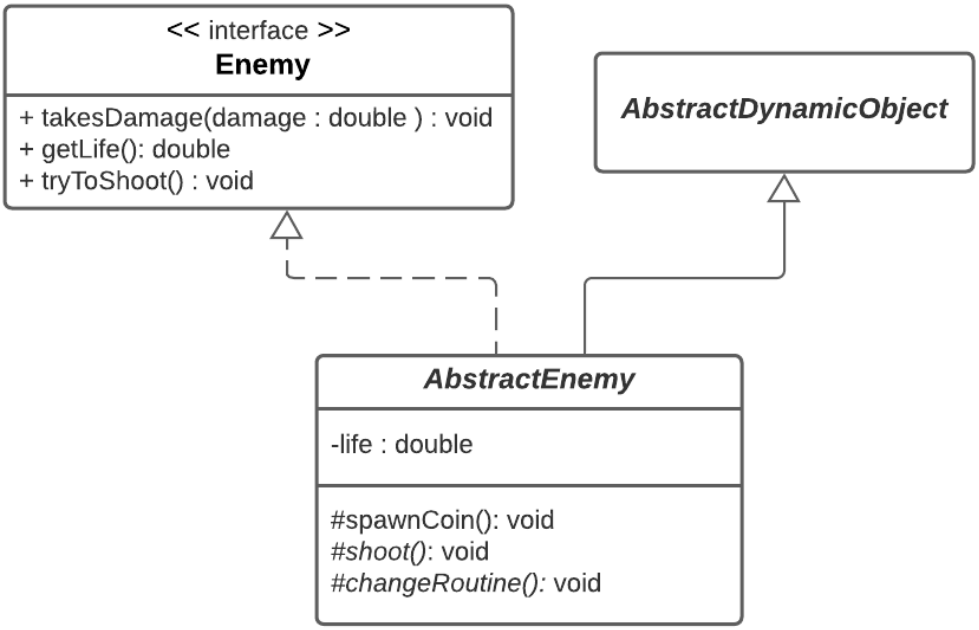


Figura 2.3: Schema UML che rappresenta la gerarchia dei nemici

I nemici sono una delle principali concretizzazioni della gerarchia descritta in precedenza, essi estendono la classe `AbstractDynamicObject` essendo degli oggetti dinamici e implementano l'interfaccia `Enemy` che li identifica come nemici.
La classe `AbstractEnemy` implementa le funzionalità comuni a tutti i nemici, come la gestione della vita, lo spawn di una moneta alla morte e la collisione con gli altri `GameObject`. Contiene un metodo template `tryToShoot()` che invoca al proprio interno il metodo astratto `shoot()`, che verrà successivamente implementato da ogni nemico a seconda della caratterizzazione dello stesso.
Vi è inoltre un metodo astratto per il cambio della routine di movimento (`changeRoutine()`), che sarà implementato in maniera differente per ogni tipologia di nemico.
Questo design permette senza particolari problemi l'aggiunta di nuove tipologie di nemici che possiedono proprie routine di movimento e di sparo.

- **Factory di Enemy**

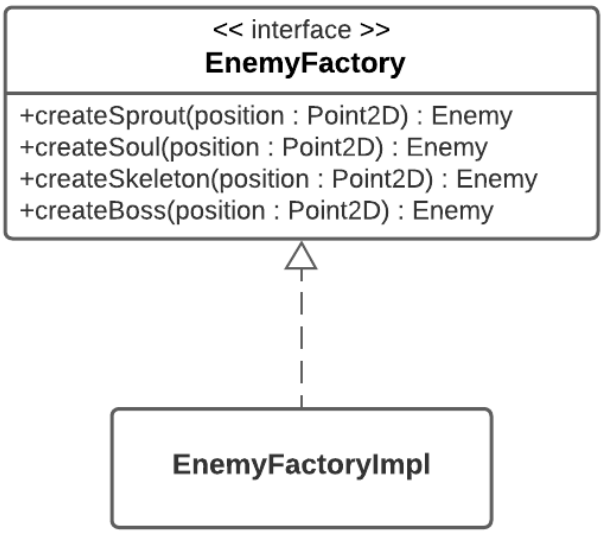


Figura 2.4: Schema UML che rappresenta la EnemyFactory

La factory di nemici presenta al proprio interno un metodo per la creazione di ogni tipologia di nemico(`Sprout`, `Soul`, `Skeleton` e `Boss`) . Le implementazioni dei nemici sono state realizzate con l'utilizzo di classi anonime che sfruttano la classe `AbstractEnemy` e ne implementano i metodi astratti per poter realizzare il nemico desiderato.
L'utilizzo di una factory, abbinato alla scelta di utilizzare una classe astratta per rappresentare un nemico generico, permette da un lato come già evidenziato in precedenza la facile aggiunta di nuove tipologie di nemici, dall'altro di ottenere rapidamente delle istanze di essi.

- **MainMenu**
  Il menù principale è composto da una view che ne individua l'intefaccia grafica e un controller che si occupa della coordinazione della GUI con il resto dell'applicazione, in particolare della creazione di una nuova partita.
  Questa semplice divisione tra controller e view permette l'indipendenza dell'applicazione dalla tecnologia utilizzata per la parte grafica del Main Menu, e contribuisce alla separazione delle responsabilità dividendo quelle prettamente grafiche, da quelle logiche.

Davide Di Marco

- **Main Character**
  Il main character rappresenta l'entità principale del gioco. L'interfaccia `MainCharacter` estende da `DynamicObject` e la sua implementazione è contenuta nella classe `MainCharacterImpl`. Il main character quindi possiede tutti i metodi di `DynamicObject` e ne aggiunge degli ulteriori caratteristici proprio di questa entità di gioco come per esempio: la vita e le monete raccolte. Inoltre sono presenti ulteriori metodi che ne rappresentano lo stato di vittoria e di sparo del proiettile. Il main character è l'unica entità di gioco che può essere controllata dall'utente e il suo movimento è gestito dall'interfaccia `MainCharacterMovement`.

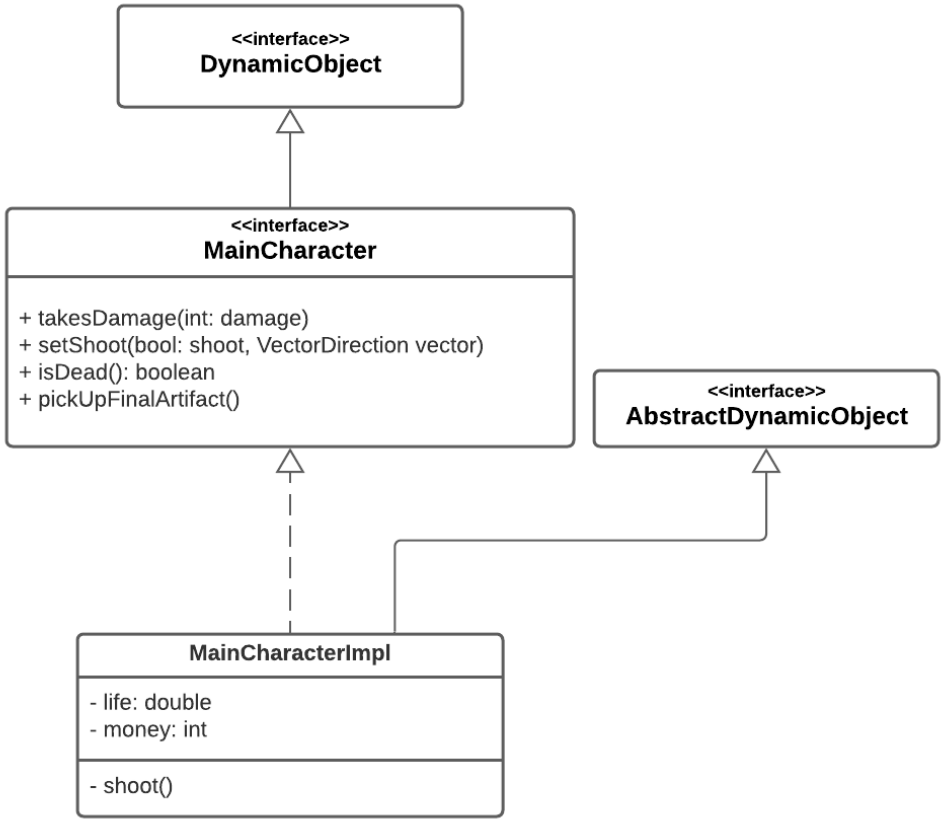


Figura 2.5: Schema UML che rappresenta il main character

- **Command e MainCharacterMovement**
  Per la gestione dell'input da tastiera che permette il controllo del personaggio principale del gioco ho creato un'interfaccia `Command` la quale implementazione è contenuta all'interno della classe `CommandImpl`. `CommandImpl` è responsabile di gestire il movimento del personaggio utilizzando il riferimento all'oggetto `MainCharacterImpl` e all'oggetto `MainCharacterMovementImpl` che contiene i metodi creati per il movimento del personaggio.
  La separazione del concetto di movimento da quello di personaggio consente la suddivisione delle responsabilità tra l'interfaccia `MainCharacter` e la sua relativa interfaccia di movimento. Infine consente la creazione di nuove tipologie di movimento senza modificare l'implementazione del `MainCharacter`.

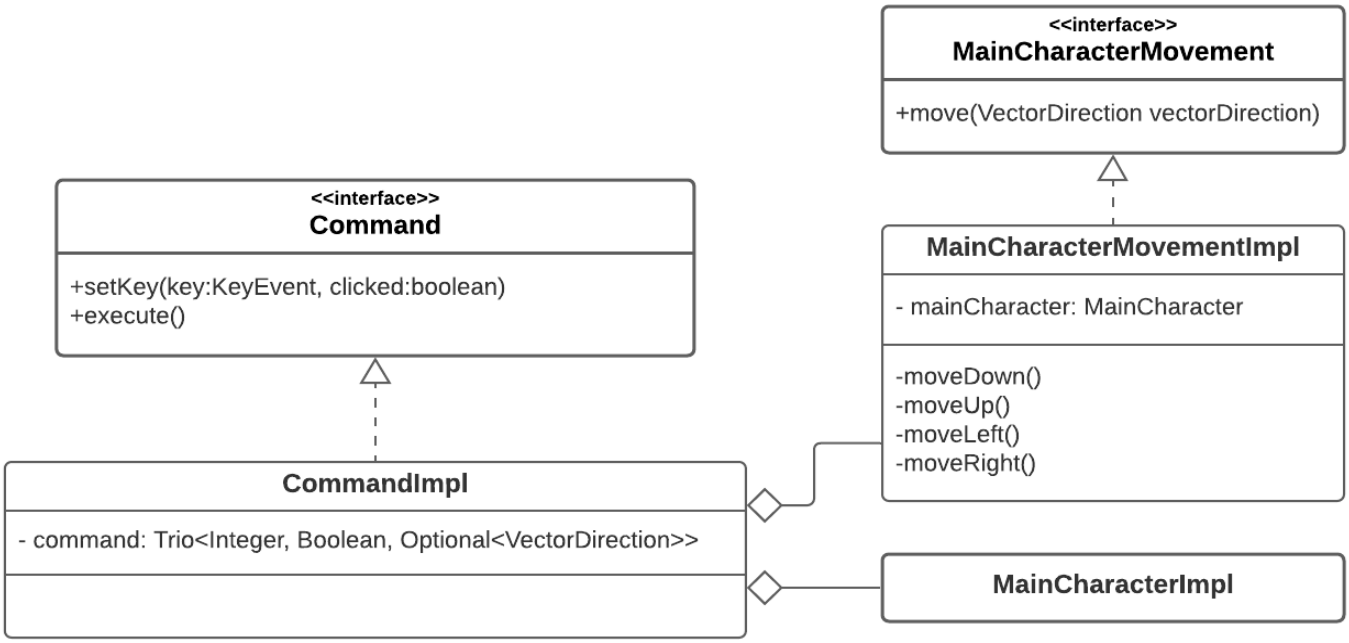


Figura 2.6: Schema UML raffigurante la gestione dei comandi e del movimento

- **Factory di bullet**
  La factory di Bullet presenta al suo interno una serie di metodi adibiti alla creazione di differenti tipologie di proiettili abbinati ognuno al proprio personaggio. L'implementazione dei metodi dell'interfaccia `Bulletfactory` è contenuta all'interno della `BulletFactoryImpl` la quale ha il compito di gestire la creazione del proiettile e quindi di creare un nuovo oggetto di tipo `Bullet` con specificati i parametri desiderati.
  Si ricorda che l'interfaccia `Bullet` estende da `DynamicObject` e quindi il `Bullet` è a tutti gli effetti un oggetto dinamico il quale implementa i metodi della suddetta interfaccia. L'implementazione dell'interfaccia `Bullet` e' contenuta all'interno della classe `BulletImpl`.
  L'utilizzo del `pattern Factory` consente la facile creazione di nuove tipologie di proiettili ognuno con le proprie caratteristiche rendendo così estendibile il videogioco per future implementazioni.

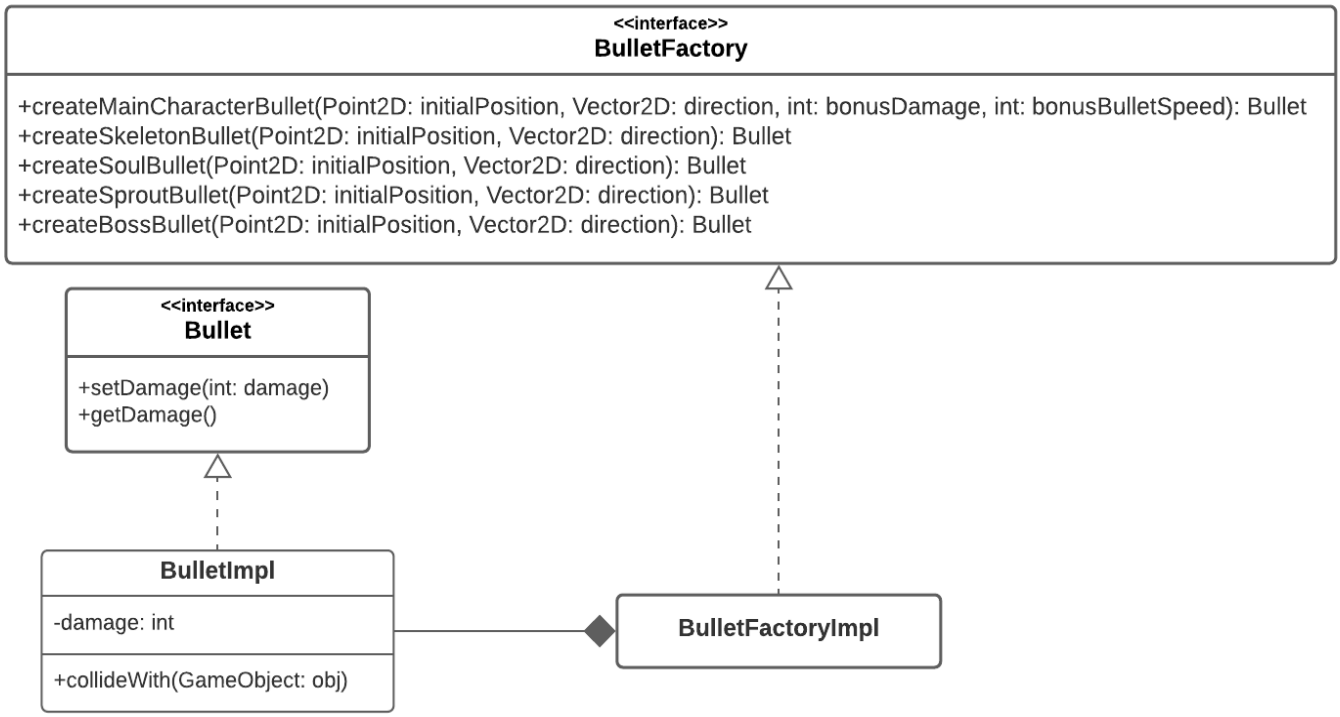


Figura 2.7: Schema UML che rappresenta la BulletFactory

Nicolò Malucelli

- **RoomManager**

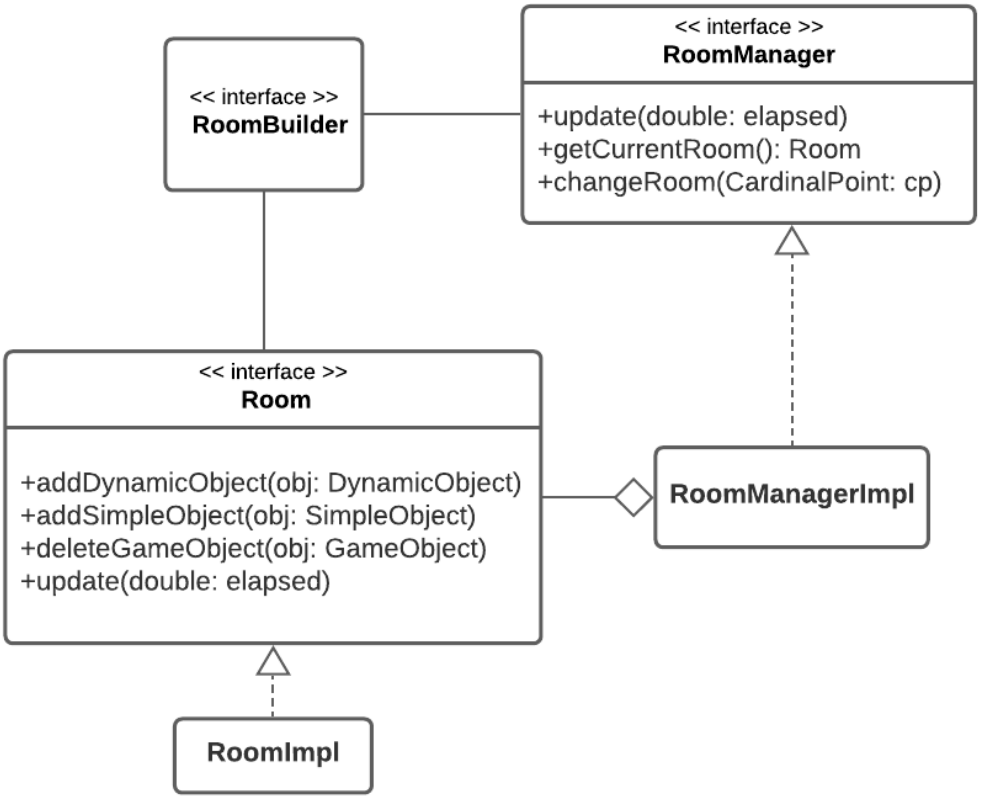


Figura 2.8: Schema UML descrivente il RoomManager

Il `RoomManager` è il gestore di stanze. Il suo compito è quello di creare l'ambiente di gioco, decidendo in modo randomico la posizione delle stanze all'interno della mappa, designando una delle stanze come quella di partenza ed un'altra come "stanza del boss".
Ad ogni stanza (`Room`) è possibile aggiungere e rimuovere sia `SimpleObject` che `DynamicObject`. Tutte le volte che viene richiamato l'update del `RoomManager`, esso chiamerà a sua volta l'update della stanza corrente che invocherà l'aggiornamento di stato su tutti i `DynamicObject` e controllerà tutte le collisioni tra gli oggetti al suo interno.
È il `RoomManager` che permette al `MainCharacter` di muoversi tra le varie stanze, selezionando una nuova stanza come stanza corrente ogniqualvolta sarà richiesto.

Con l'organizzazione logica appena descritta si ha una netta divisione delle responsabilità. Il `RoomManager` conosce la disposizione delle varie stanze e sa qual è stanza corrente, senza però sapere il contenuto di ogni singola stanza. Al contrario le diverse stanze non conoscono l'intero ambiente di gioco o la posizione della stanza all'interno di esso, ma soltanto ciò che contengono.
Questa organizzazione permette di implementare facilmente future feature come un gioco a più livelli o l'aggiunta di stanze aventi caratteristiche particolari.

- **RoomBuilder**

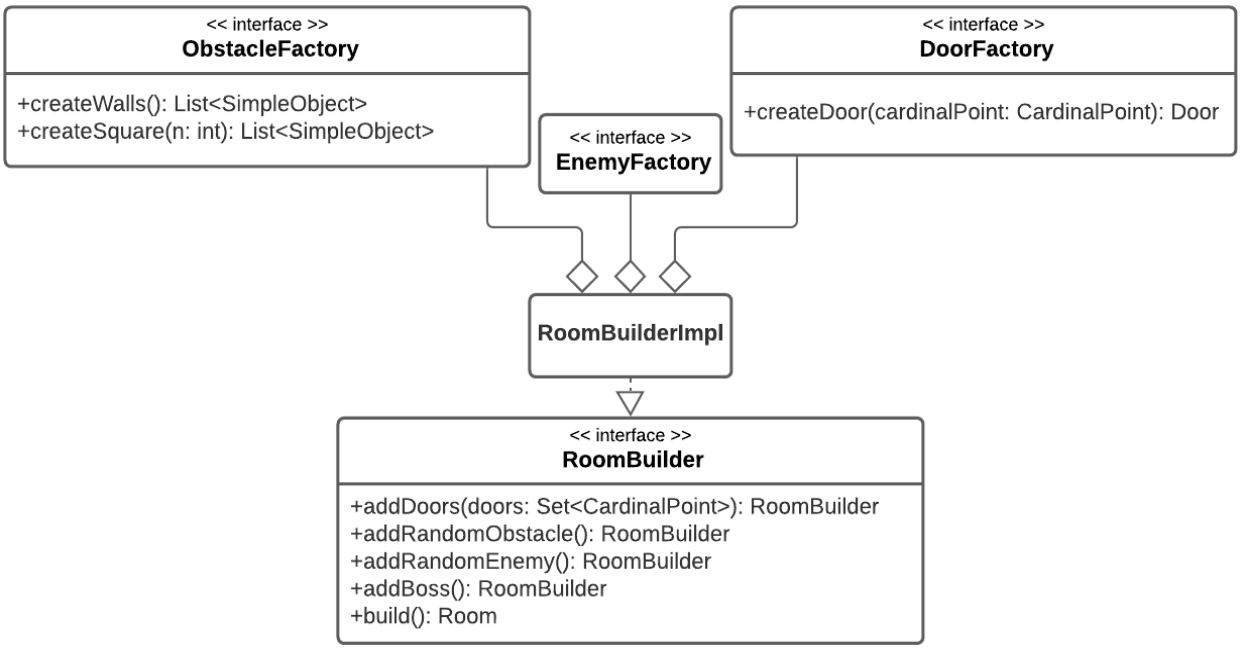


Figura 2.9: Schema UML descrivente il RoomBuilder

Dato che ogni `Room` richiederebbe un numero troppo elevato di parametri per poter essere creata, si è scelto di utilizzare il pattern Builder. L'utilizzo di questo pattern permette di creare ed inizializzare in modo facile e veloce le stanze, consentendo inoltre di imporre vincoli aggiuntivi nella creazione che altrimenti non sarebbero possibili. È ad esempio impossibile aggiungere al momento della creazione ostacoli o nemici ad una stanza se essa contiene già il Boss.
Questi vincoli sono inoltre molto semplici da realizzare e in versioni future dell'applicazione potrebbero essere velocemente rimossi o aggiunti.

Il pattern Builder offre una forte personalizzazione, permettendo anche successivamente la creazione di nuovi metodi in seguito all'aggiunta di nuove funzionalità, come ad esempio trappole o botole.
All'interno di `RoomBuilderImpl` sono istanziate tre Factory: una per i nemici, una per gli ostacoli ed una per le porte.

- **Factory**
  La `DoorFactory` (fig. 2.9) permette di creare un'oggetto `Door`, specificando come parametro il relativo punto cardinale all'interno della stanza.
  La `ObstacleFactory` (fig. 2.9) permette invece di creare disposizioni randomiche di ostacoli. Si è scelto di utilizzare il pattern Factory per poter poi in futuro creare nuovi tipi di configurazioni semplicemente aggiungendo un nuovo metodo all'interfaccia.

- **BoundingBox**
  Ad ogni `GameObject` è associato un `BoundingBox`. Il `BoundingBox` è identificato da una posizione, dalla larghezza e dall'altezza e serve per verificare se è in corso oppure no una collisione tra due oggetti del mondo di gioco.
  Il metodo `move()` permette di settare la nuova posizione del `BoundingBox` in modo da farla combaciare con quella dell'oggetto ed è invocato tutte le volte che il `GameObject` si sposta.

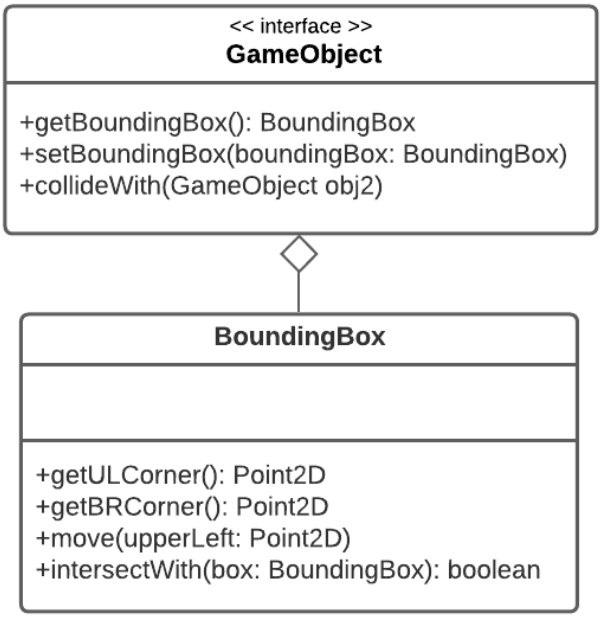


Figura 2.10: Schema UML descrivente i BoundingBox

Alberto Di Girolamo

- **InGameMenu**

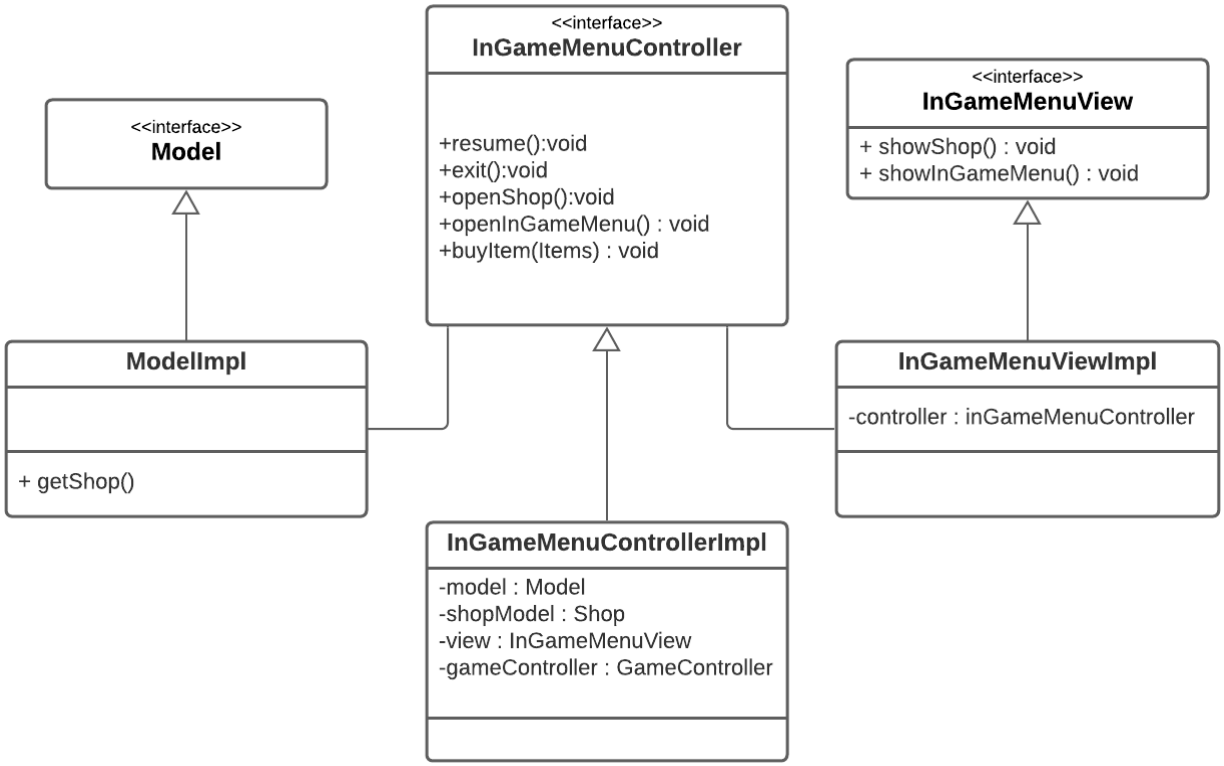


Figura 2.11: Schema UML che rappresenta InGameMenu

Il menù di gioco è strutturato con il pattern architetturale MVC. Questo permette di mantenere separate le tre parti: `Model`, `View` e `Controller`. Il `Model` rappresenta il centro di questa struttura: interpreta il comportamento del menu di gioco in maniera del tutto indipendente dall' interfaccia utente `View`.
Quest' ultima permette di mostrare il menu di gioco e lo shop, alternando quando necessario i vari elementi grafici.
Il `Controller` si occupa di gestire e far interagire la grafica con la logica, richiamando gli opportuni metodi.
Il menù di gioco è strutturato su due livelli: in primo piano
l' `InGameMenuController` dà la possibilità di riprendere la partita corrente chiudendo il menu (Resume), di uscire dal gioco (Exit) e di entrare nel negozio (Shop).

Il secondo livello si presenta nel caso venisse effettuata quest' ultima scelta, poichè si verrà indirizzati nello `Shop` che dà la possibilità al giocatore di acquistare degli oggetti che potenziano le statistiche del personaggio.

Quando viene comprato un `Item` l' `InGameMenuController` richiama un oggetto `Shop`. Questo deve verificare che l'acquisto sia legale poichè se un oggetto è stato già comprato non è possibile acquistarlo nuovamente, ciò viene realizzato tenendo traccia degli acquisti precedentemente effettuati.
Inoltre controlla che il personaggio abbia abbastanza monete per ottenere l'item selezionato. Nel caso le due precedenti condizioni siano verificate lo `Shop` si occuperà di fruire l' `Item` richiesto tramite l'utilizzo del `Builder`.

- **ItemBuilder**

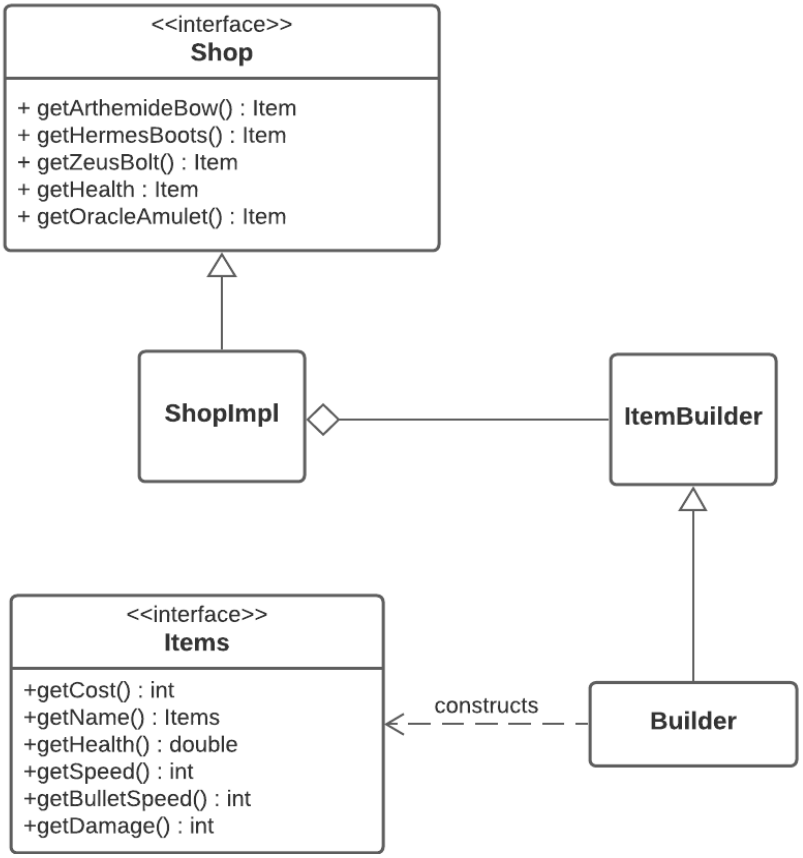


Figura 2.12: Schema UML che rappresenta il builder di item

Quando viene acquistato un `Item`, per ottenerlo si fa uso del pattern creazionale `Builder`. Grazie a questo è possibile costruire i vari `Item` settando, in base al bisogno, le varie proprietà che devono possedere (più velocità di movimento, maggiore velocità di sparo, maggiore danno e aumento di vita).
Ogni Item ha due parametri obbligatori: il nome dell' oggetto (indicato da un' enumerazione) e il suo prezzo. I parametri restanti invece, sono inizializzati con un valore di default. Questa scelta progettuale permette di rendere indipendente l'oggetto creato dalle varie parti che lo compongono.
Infatti con il pattern `Builder` è possibile in futuro implementare ulteriori `Item` anche concettualmente complessi con poca difficoltà. Procedendo con una strategia step-by-step è necessario solamente creare un oggetto di tipo `Item` impostandogli i valori voluti, evitando di lavorare così con i costruttori.

- **Boss**
  La classe `Boss` estende la classe astratta `AbstractEnemy` poiché possiede caratteristiche comuni a tutti gli `Enemy` ma la logica del comportamento di gioco è diversa. La routine di movimento del `Boss` cambia quando questo perde metà della sua vita modificando inoltre anche il comportamento del proiettile. Ogni qual volta che il `Boss` collide con un oggetto la sua velocità di movimento aumenta, per rendere più arduo il completamento del gioco.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

In questa parte verranno illustrati i test che sono stati effettuati per verificare il corretto funzionamento delle componenti principali di MazeDungeon.

- **TestMainCharacter :**
  Il test consente di verificare il corretto funzionamento del main character verificando le proprie caratteristiche iniziali e le sue dinamiche di gioco come: sparo, danno, raccolta di una moneta e vittoria. Inoltre verifica anche la corretta implementazione dell'aggiunta del personaggio nella `Room`.

- **TestCommand :**
  Il test corrente permette di verificare il corretto funzionamento dei comandi da tastiera simulando la pressione dei tasti ed il movimento del personaggio.

- **TestBulletFactory :**
  Questo test permette di verificare il buon funzionamento della `BulletFactory`. Il test crea i diversi tipi di proiettile tramite la factory e verifica il corretto assegnamento delle caratteristiche del proiettile nelle varie fasi del gioco. Inoltre verifica la corretta creazione del proiettile all'interno della stanza.

- **TestBoss :**
  Questo test si occupa di controllare il funzionamento del `Boss` di gioco. Viene testato il corretto funzionamento della vita, della posizione e della velocità di movimento.

- **TestEnemyFactory :**
  Questo test si occupa di verificare il corretto funzionamento della `EnemyFactory` e dei nemici da essa istanziati. I metodi testano il corretto posizionamento dei nemici, verificano che le loro statistiche siano coerenti, che lo sparo avvenga correttamente e che la perdita di vita sia funzionante.

- **TestItem :**
  I test presenti in questa classe verificano il corretto funzionamento dello `Shop` e degli `Item`. Per far ciò per ogni oggetto che viene comprato vengono lette le statistiche del `MainCharacter` prima e dopo l'acquisto degli `Item`. Successivamente i dati vengono confrontati controllando che le abilità del personaggio siano state correttamente modificate.

- **TestRoomManager :**
  Questo test si occupa di verificare la correttezza della creazione della mappa di gioco. In particolare verifica che tutte le stanze siano connesse e che il cambio di stanza funzioni correttamente.

- **TestRoomBuilder :**
  Il test controlla la corretta creazione di una `Room` attraverso il `RoomBuilder`. Verifica la correttezza dei vincoli di costruzione imposti, ad esempio che non è possibile aggiungere nemici e boss insieme. Verifica inoltre la correttezza dei parametri e l'impossibilità di richiamare molteplici volte il metodo `build()`.

- **TestRoom :**
  Il test verifica la corretta apertura e chiusura delle porte all'interno della stanza in presenza o meno di nemici ed il corretto inserimento dei `GameObject` al suo interno.

- **TestBoundingBox :** Il test verifica il corretto funzionamento delle collisioni di gioco creando molteplici `BoundingBox` e controllandone le intersezioni.

- **TestGUI :** Per quanto riguarda il funzionamento dei pulsanti delle interfacce grafiche e il posizionamento dei componenti all'interno di esse, sono stati effettuati test manuali, verificandone la portabilità su diversi sistemi operativi.

3.2 Metodologia di lavoro

Per la realizzazione di MazeDungeon lo sviluppo delle varie parti è stato suddiviso equamente, assegnando ad ognuno un importante parte di logica, determinante per il corretto funzionamento dell'applicazione. Non sono state apportate grandi modifiche alla suddivisione iniziale durante la realizzazione del progetto. È stato utilizzato il DVCS Git, ogni membro del gruppo è in possesso di una copia locale del repository e ad ogni importante modifica procede all'aggiornamento di quello remoto. Git è stato essenziale per permettere la collaborazione dei vari membri a distanza, senza troppe difficoltà nell'integrazione delle parti sviluppate separatamente. Di seguito vengono illustrate le parti sviluppate da ogni membro del team.

Filippo Venturini: Realizzazione della gerarchia dei `GameObject`, che ha permesso la caratterizzazione delle entità di gioco all'interno di una stanza in oggetti statici e oggetti dinamici. Implementazione degli `Enemy`, con la definizione del loro individuale comportamento a seconda della tipologia (`Sprout`, `Skeleton`, `Soul`). Implementazione del `FinalArtifact` che permette la conclusione del gioco con vittoria dell'utente. Costruzione dell'interfaccia utente e della logica del `MainMenu` che permette di iniziare una nuova partita.

Davide Di Marco: Creazione del personaggio principale e gestione delle sue interazioni con il mondo di gioco. Gestione dello sparo e del movimento del personaggio principale. Ho gestito inoltre la creazione dei vari proiettili del gioco sia quelli del personaggio principale sia quelli dei nemici e la loro implementazione. Infine mi sono occupato della gestione degli input da tastiera.

Nicolò Malucelli: Creazione dell'ambiente di gioco, più in particolare la creazione randomica della mappa con l'aggiunta di porte che permettono al personaggio di muoversi tra una stanza e l'altra. Generazione di nemici ed ostacoli all'interno delle singole stanze e controllo delle collisioni tra i vari oggetti di gioco attraverso i `BoundingBox`. Implementazione dell' HUD per mostrare graficamente la vita del personaggio, le monete raccolte e gli item acquistati.

Alberto Di Girolamo: Realizzazione dell'interfaccia grafica e della logica dell' `InGameMenu`. A questo va aggiunta la generazione degli `Item` e il funzionamento dello `Shop` che deve occuparsi del loro acquisto e della loro gestione. Infine implementazione del `Boss`, realizzando la sua personale routine di movimento e di sparo.

In cooperazione:

Il nodo centrale dell'applicazione è rappresentato dalla coppia `GameController`, `GameView`. Dato che all'interno di essi sono contenute porzioni di codice utili ad ogni membro come : eventi per pressioni di tasti, aperture menù, GameLoop, rendering di immagini di gioco e animazioni; queste classi sono state realizzate in cooperazione e ogni membro ha contribuito aggiungendo le funzionalità necessarie alla propria parte.

Divisione classi e package svolti singolarmente :

Filippo Venturini:

- gamestructure.mainmenu
- model.gameobject.dynamicobject.enemy
- model.gameobject.GameObject.java
- model.gameobject.simpleobject.SimpleObject.java
- model.gameobject.simpleobject.AbstractSimpleObject.java
- model.gameobject.simpleobject.FinalArtifact.java
- model.gameobject.dynamicobject.DynamicObject.java
- model.gameobject.dynamicobject.AbstractDynamicObject.java

Davide Di Marco:

- input.Command.java
- input.CommandImpl.java
- input.Trio.java
- model.gameobject.dynamicobject.maincharacter
- model.gameobject.dynamicobject.bullet
- model.common.vectorDirection

Nicolò Malucelli:

- model.room
- model.gameobject.simpleobject.door
- model.gameobject.simpleobject.obstacle
- model.common.BoundingBox
- gamestructure.game.HUDPanel

Alberto di Girolamo:

- gamestructure.ingamemenu
- model.shop
- model.gameobject.dynamicobject.enemy.Boss.java

3.3 Note di sviluppo

Di seguito si elencano per ogni componente del team le feature avanzate dell-linguaggio utilizzate.

Filippo Venturini:

- `Stream`: nella configurazione dei componenti della GUI del MainMenu.
- `Lambda expression`: aggiunta e configurazione dei componenti grafici del MainMenu e sparo di un nemico in quattro direzioni.

Davide Di Marco:

- `Lambda expression`: utilizzata per ciclare una mappa all'interno di commandImpl.
- `Stream`: utilizzato per la ricerca in una mappa del bottone premuto.
- `Optional`: restituire il bottone premuto che potrebbe anche essere nessuno o non presente all'interno dei bottoni cliccabili.

Nicolò Malucelli:

- `Stream`: identificare la stanza adatta ad ospitare il boss, ricerca di una stanza nella mappa di gioco, ricerca dell'id del boss, ottenimento della lista di tutti gli id di gioco associati ad una immagine.
- `Optional`: restituire l'id del boss.

Alberto Di Girolamo:

- `Lambda expression`: nell'aggiunta degli eventi dei JButton e dei componenti grafici dell' InGameMenu.
- `Stream`: nell'aggiunta dei componenti grafici alla GUI dell' InGameMenu.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Avendo sviluppato un clone di "The Binding of Isaac" non riteniamo utile l'espansione del videogioco se non per fini puramente personali. Siamo comunque soddisfatti in quanto il software è stato predisposto a ulteriori espansioni come : l'aggiunta di nuovi livelli, l'aggiunta di nuovi Item, l'aggiunta di nuovi nemici ecc. In un'ottica di evoluzione del software, si potrebbe inventare una trama che possa collegare gli elementi presenti nel gioco, caratterizzandolo con un filo logico, per migliorarne l'esperienza di gioco. Di seguito si presentano le autovalutazioni dei singoli componenti del team.

Filippo Venturini:

Sono molto soddisfatto del mio contributo per la realizzazione di questo progetto, che oltre ad avermi messo alla prova, ha stimolato il mio interesse verso la progettazione software e la programmazione ad oggetti.
Penso che la realizzazione di un clone di un videogioco già esistente (seppur con grafica differente) ci abbia penalizzato dal punto di vista dell'originalità, ma ci ha permesso comunque di realizzare un software che lo emulasse e abbiamo messo in pratica strategie di progettazione e programmazione che nella mia esperienza non avevo mai avuto occasione di attuare.
Per quanto riguarda le difficoltà incontrate da me, penso si siano presentate per la maggior parte durante la fase di progettazione iniziale, in particolare nella concezione della gerarchia dei GameObject, avendo la necessità di trovare una struttura che permettesse solidamente e facilmente l'espandibilità e la caratterizzazione delle entità di gioco.

Davide Di Marco:

Sono rimasto complessivamente contento del lavoro svolto dall'intero gruppo. Inizialmente non è stato facile interfacciarsi con la progettazione del progetto, ma approfondendo un po' i vari aspetti progettuali sono rimasto contento dell'architettura dell'intero software. È il primo progetto che ho fatto in gruppo e mi ha permesso di capire il paradigma ad oggetti sicuramente in maniera più approfondita rispetto alle conoscenze pregresse. Per quanto riguarda la mia parte, sono rimasto abbastanza soddisfatto anche se ho trovato un po' di difficoltà nella gestione degli input da tastiera simultanei ed allo stesso tempo reattivi anche a livello visivo per avere un gioco fluido. Questa esperienza mi ha permesso di integrare delle conoscenze che non avevo e sicuramente mi ha fatto ben capire cosa vuol dire lavorare in team e quindi mi ha dato un aiuto non indifferente dal punto di vista professionalizzante

Nicolò Malucelli:

Sono soddisfatto del mio contributo dato al progetto. Mi sento di aver partecipato attivamente e nonostante questo fosse il mio primo progetto di gruppo vero e proprio sono complessivamente soddisfatto, non solo del risultato finale ma anche dell'organizzazione generale all'interno del gruppo.
Inoltre il progetto è stato molto stimolante e sono felice che mi sia stata assegnata la parte di gestione dell'ambiente di gioco poiché ho potuto cimentarmi in nuove sfide come la creazione della mappa di gioco, cosa che non avevo mai fatto prima.
Penso che la parte più stimolante e difficile sia stata la fase di progettazione e di design che prima dell'inizio del progetto un po' sottovalutavo e consideravo minore rispetto alla programmazione vera e propria.

Alberto Di Girolamo:

Mi reputo soddisfatto della realizzazione di questo progetto.
Durante la progettazione concettuale ci sono state alcune difficoltà poiché questo rappresenta, almeno per me, il primo vero "lavoro di gruppo". La difficoltà principale è stata quella di implementare concettualmente il problema su carta (UML) per poi riportarlo su codice mantenendo la coerenza tra le due parti.
Un' ulteriore difficoltà l'ho riscontrata con l'utilizzo del pattern architetturale MVC. Poiché inizialmente nessun membro lo conosceva "approfonditamente" ho dovuto usare parte del tempo nello studio di questo e del suo utilizzo.
Lo sviluppo di questo progetto l'ho trovato stimolante, in quanto mi ha permesso attraverso il "problem solving" di affrontare e trovare soluzioni ai problemi riscontrati.

Appendice A

Guida utente

Di seguito si illustra la configurazione dei comandi di gioco.

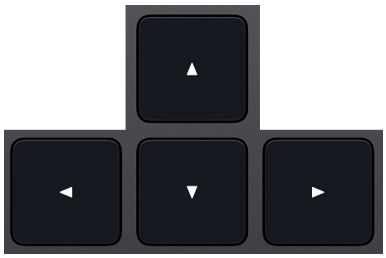
Main Menu:

Figura A.1 : schermata del Main Menu.

Movimento:

W: Movimento del personaggio verso l'alto.
S : Movimento del personaggio verso il basso.
A: Movimento del personaggio verso sinistra.
D: Movimento del personaggio verso destra.

Sparo:



Freccia su: sparo verso l'alto.
Freccia giù : sparo verso il basso.
Freccia sinistra: sparo verso sinistra.
Freccia destra: sparo verso destra.

Apertura Menu In Game:

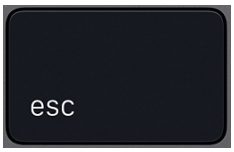


Figura A.2 : schermata del Menu In Game.

Figura A.3 : schermata dello Shop.

Appendice B

Esercitazioni di laboratorio

Di seguito si elencano gli esercizi di laboratorio svolti da ogni componente del team.

B.0.1 Filippo Venturini

- Laboratorio 07: `https://virtuale.unibo.it/mod/forum/discuss.php?d=62582`
- Laboratorio 08: `https://virtuale.unibo.it/mod/forum/discuss.php?d=63865`
- Laboratorio 09: `https://virtuale.unibo.it/mod/forum/discuss.php?d=64639`

B.0.2 Alberto Di Girolamo

- Laboratorio 07: `https://virtuale.unibo.it/mod/forum/discuss.php?d=62582`
- Laboratorio 09: `https://virtuale.unibo.it/mod/forum/discuss.php?d=64639`
- Laboratorio 10: `https://virtuale.unibo.it/mod/forum/discuss.php?d=66753`