Relazione
“Isaccoop”

Marco Costantini
Andrea Dotti
Daniele Pancottini
Giacomo Pierbattista
Dilaver Shtini

10 aprile 2023

Indice

Capitolo 1

Analisi

Il software mira a realizzare un gioco roguelike liberamente ispirato a "The Binding Of Isaac". Con "roguelike" si intende un videogioco tipicamente caratterizzato dall'esplorazione di un dungeon attraverso livelli generati proceduralmente e morte permanente del personaggio giocante.

1.1 Requisiti

Requisiti funzionali obbligatori

- Il protagonista comincerà la partita in una stanza iniziale vuota.
  Il giocatore può:
  - muoversi liberamente all'interno di ogni stanza e sparare "sfere di energia"
  - passare da una stanza all'altra, solo dopo aver sconfitto tutti i **nemici** che si trovano al suo interno, se presenti
  - raccogliere monete e cuori ("**item**") sparsi nelle varie stanze
  - comprare potenziamenti (**powerup**) nell'apposito negozio (**shop room**), usando le monete raccolte precedentemente
  - raccogliere un powerup gratis nella stanza **treasure room**
  - perdere una vita ogni volta che viene colpito da un nemico o da un proiettile lanciato da esso
  - visualizzare la struttura del livello giocato nella **minimappa** per sapere in quale stanza si trova e vedere quali sono state completate

  - visualizzare le sue **"statistiche/caratteristiche"**, ad esempio: il numero di vite rimaste (indicate dai cuori), di monete raccolte, quanto danno riesce a infliggere ai nemici...
  - il gioco finisce (game over) se il giocatore perde tutte le sue vite/cuori o quando sconfigge tutti i nemici in tutte le stanze, compreso il boss finale

- Nel gioco vi sono cinque tipi di stanze:
  - **START**: stanza iniziale vuota in cui il giocatore si trova all'inizio della partita
  - **TREASURE**: stanza in cui si trova un powerup gratuito, che il giocatore può raccogliere per sconfiggere più facilmente i nemici
  - **SHOP**: stanza in cui il giocatore può spendere le monete raccolte precedentemente per comprare altri powerup
  - **STANDARD**: stanza in cui si trovano dei nemici e da cui il giocatore può uscire solo dopo averli sconfitti tutti
  - **BOSS**: stanza in cui si trova il boss finale; una volta sconfitto, la partita termina.

- Una semplice AI (Artificial Intelligence) gestirà i movimenti e gli attacchi dei nemici e del boss finale, i quali cercheranno di colpire il giocatore. Per AI si intende un semplice software che gestisce il movimento e il comportamento dei nemici.

- Implementazione della guida/tutorial al gioco, visualizzabile nel menu di gioco prima di iniziare una partita

- Gestione della pausa durante il gioco

Requisiti funzionali opzionali

- gestione di più livelli, ognuno indipendente dagli altri, con stanze differenti e difficoltà incrementali

- gestione del salvataggio a chiusura dell'applicazione per continuare la partita successivamente

- effetti sonori

- impostazioni di gioco (esempi a titolo indicativo e non esaustivo):

  - scelta del numero minimo e massimo delle stanze che possono formare un livello
  - numero di livelli da giocare in una partita

Requisiti non funzionali

- il software deve essere portabile, quindi deve essere eseguibile su macchine che eseguono Windows, Mac OS X e Linux/UNIX.
- Il software deve garantire un'esperienza di gioco adeguata (fluida, gradevole), indipendentemente dallo stato della partita.

1.2 Analisi e modello del dominio

Il software permetterà al giocatore di avviare una sessione di gioco, che gestirà l'avanzamento del personaggio principale attraverso le varie stanze che formano il mondo di gioco (**"livello/level"**). In un livello sono presenti un numero semicasuale di stanze; in alcune di esse saranno presenti i "nemici" da sconfiggere e gli "item", mentre in altre si potranno comprare/raccogliere i "powerup". Una volta entrato in una stanza in cui si trovano i nemici, il giocatore dovrà sconfiggerli tutti per poter passare ad un'altra stanza. I nemici saranno gestiti da una semplice AI (come descritto precedentemente) i quali cercheranno di attaccare e uccidere il giocatore. Per sconfiggere tali nemici, il giocatore spara le "sfere di energia". Utilizzando i powerup raccolti o comprati nello shop, il giocatore può sconfiggere più facilmente i nemici. Il giocatore perde una vita (indicate con i cuori/"hearts") ogni volta che viene colpito da un nemico o da un proiettile sparato da esso. Il gioco terminerà se il giocatore perde tutte le vite.

Le challenge principali di questo progetto sono state:

- corretto utilizzo del pattern MVC
- gestione delle collisioni tra protagonista, item, powerup, nemici e muri delle stanze
- creazione dinamica delle stanze con nemici, powerup e item al loro interno
- implementazione e gestione dei vari potenziamenti/oggetti utilizzando pattern di progettazione adeguati

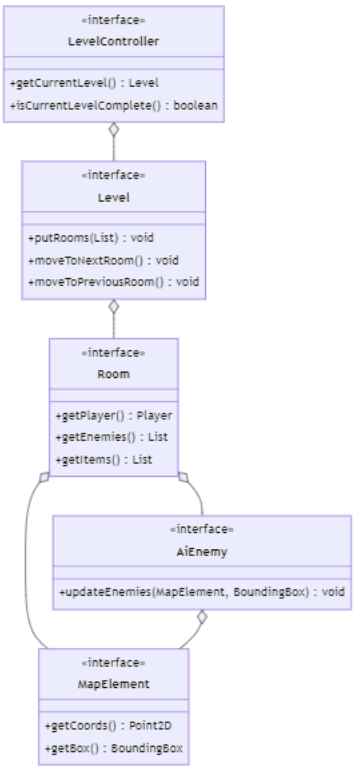

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

Nella progettazione, si è scelto di adottare il pattern architetturale MVC.

- **Model**: Il model viene affidato al Level che contiene tutto il necessario per la parte del model. Cambiare l'implementazione del livello e della stanza implicherebbe il cambiamento dell'intero model e quindi la creazione di un gioco con caratteristiche differenti.
- **Controller**: Il GameEngine si occupa di gestire il controller interfacciando il model con la view.
- **View**: L'interfaccia Scene.java ha il compito di renderizzare la scena di gioco composta dalla room corrente e da una minimappa con informazioni utili riguardanti il player, mentre l'interfaccia Graphics si occupa di renderizzare giocatore, nemici, powerup e item sulla mappa di gioco.

L'utilizzo di MVC consente di cambiare la tecnologia alla base della View senza modificare né Model né Controller del software: infatti sarà necessario cambiare l'implementazione dei file Scene, Graphics e degli altri file che contengono l'implementazione delle altre GUI realizzate, utilizzando la nuova tecnologia, sostituendo così l'implementazione attuale basata sul framework Swing con la nuova tecnologia scelta.

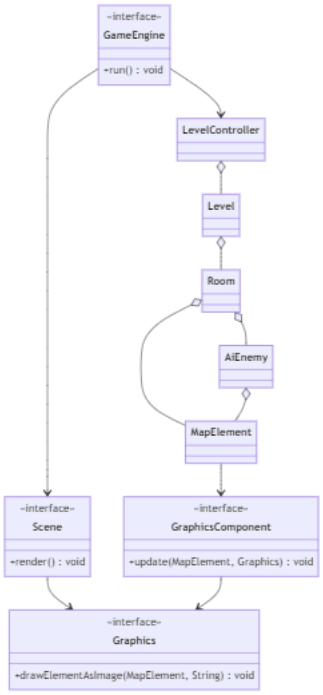

Figura 2.1: Schema UML del MVC

2.2 Design dettagliato

Marco Costantini

Gestione di PowerUp e Item

Problema Il gioco implementa diversi tipi di Item e di PowerUp che vanno ad interagire con le statistiche del player

Soluzione All'interno del gioco sono presenti Item e PowerUp che si differenziano principalmente dal fatto che gli Item non hanno un prezzo e un "super". Ho pensato di creare l'interfaccia Item e una classe astratta che va ad implementare l'interfaccia. Gli Item che non hanno un prezzo andranno ad estendere la classe astratta implementando il metodo astratto Interact(). Mentre i vari PowerUp estenderanno la classe astratta PowerUp implementando anche essi il metodo Interact(), ma con caratteristiche aggiuntive rispetto all'Item classico, come un prezzo e un attributo che va a differenziare i PowerUp super e non. In questo modo il PowerUp concettualmente resta un Item, ma con caratteristiche diverse.

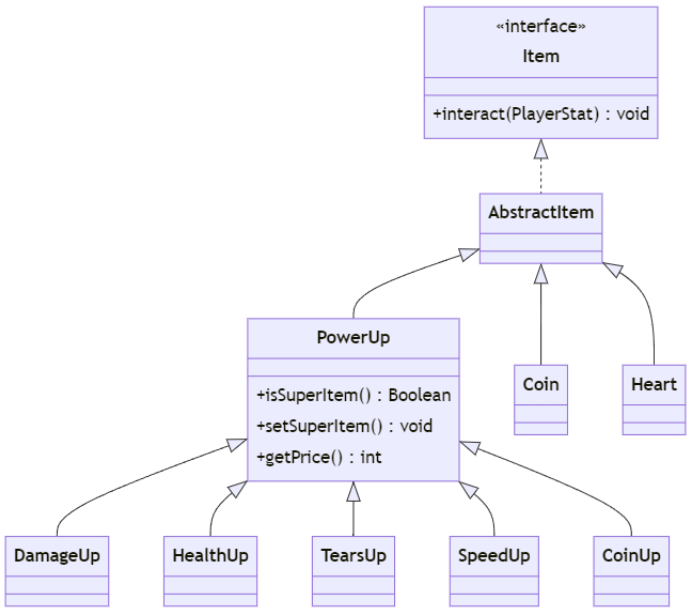

Figura 2.2: Gestione degli item e powerup

Game Loop e GameEngine

Problema Separare la logica del loop (Input, Update e Render), dalla inizializzazione del gioco

Soluzione Il game engine andrà a inizializzare il gioco (i vari controller e il GameState) e utilizzerà nella funzione run il metodo di GameLoop che si occuperà del loop vero e proprio, separando così i compiti.

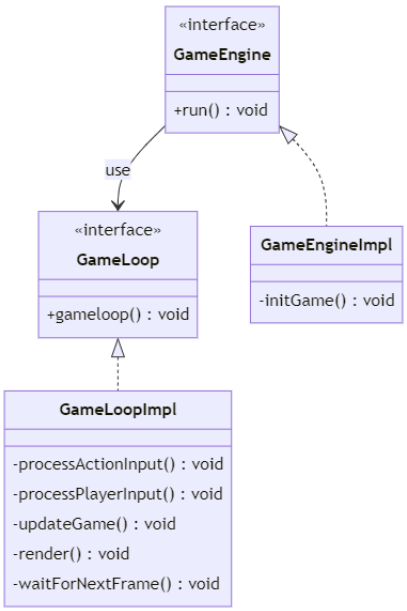

Figura 2.3: Gestione GameEngine e GameLoop

Check delle collisioni

Problema Controllo delle collisioni

Soluzione Ho gestito il controllo delle collisioni (CollisionCheckFactory, CollisionCheck) con una factory, nel momento in cui avviene la collisione, utilizzando le apposite funzioni presenti in BoundingBox, notifico l'evento alla room utilizzando i metodi per la gestione degli eventi presenti in EventFactory. In questo modo se si vuole aggiungere un check di una collisione basta aggiungere un metodo nella factory e notificare alla room l'evento desiderato.

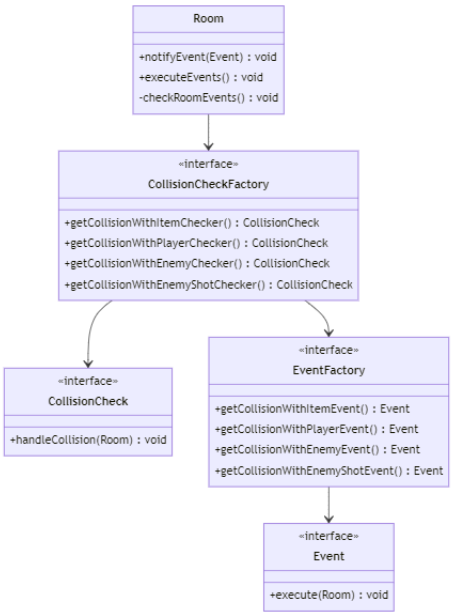

Figura 2.4: Check delle collisioni

Creazione di Item e PowerUp

Problema Come creare gli item e i powerup nelle varie stanze

Soluzione Utilizzando la factory ho creato l'interfaccia funzionale generica Creator con un metodo che restituisce una lista generica. Nella creator factory i vari metodi torneranno una Creator dove verranno creati i vari Item e PowerUp differenziando la stanza, per esempio nello shop verranno creati tre powerUp, nella Treasure uno mentre nella stanza standard verranno creati randomicamente da uno a tre Item. La stessa logica è utilizzata per la creazione dei nemici per le stanze standard e per il boss nella boss room.

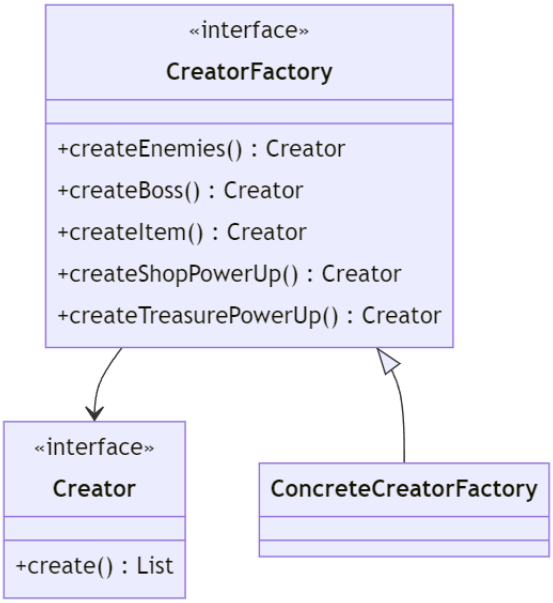

Figura 2.5: Creazione di Item e PowerUp

Andrea Dotti

Problema Occorre un "controller" per l'utente in modo da interagire con il programma (ad esempio per muovere il player).

Soluzione La linea guida per la soluzione che abbiamo deciso di seguire prende spunto dal pattern mostratoci dal professor. Ricci: `https://github.com/aricci303/game-as-a-lab/tree/master/Game-As-A-Lab-Step-7-mosquito-AI/src/rollball/input`. Questa parte del progetto era una dei tre fondamenti del pattern MVC, ovvero la logica di aggiornamento dall'esterno (in-put) della parte del model. Ho creato una interfaccia InputController con i metodi per controllare le quattro direzioni. La KeyboardInputController implementa l'interfaccia precedentemente descritta e la estende con due metodi per aggiornare la premuta / il rilascio dei tasti che definiscono le direzioni. Ho deciso di creare questa classe sufficientemente generica in modo che potesse essere usata per l'input da tastiera sia del movimento che per sparare i proiettili. Successivamente ho creato l'interfaccia dell'InputComponent con il metodo update, il quale viene in seguito implementato dalle due sottoclassi PlayerInputComponent e ShotInputComponent. Questo metodo è molto importante perché va effettivamente a legare l'input dall'InputController al model. Infatti le sottoclassi implementano rispettivamente il loro update che effettua azioni differenti in base alle informazioni che arrivano dal controller.

Il Player contiene nei sui campi i due controller, per il movimento e per lo sparo.

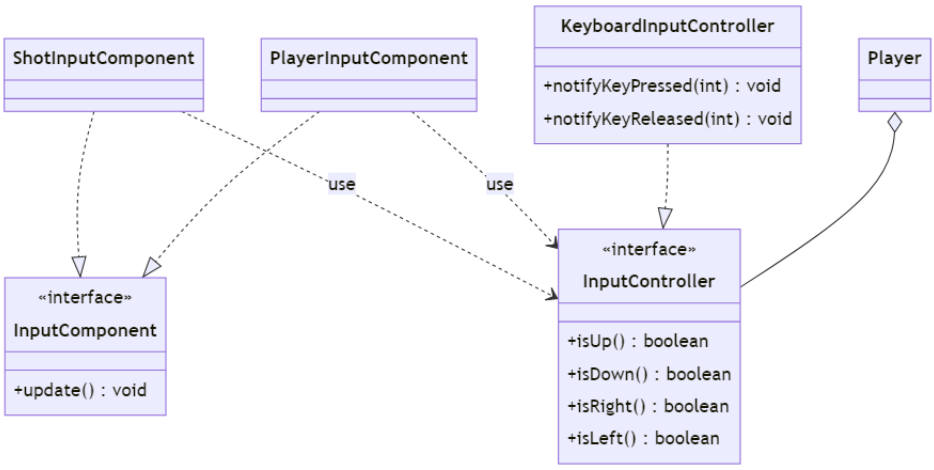

Figura 2.6: Rappresentazione UML della parte di Controller del pattern MVC

Problema Occorre un altro "controller" all'utente per cambiare stanza e per mettere in pausa il gioco.

Soluzione Analogamente al pattern seguito in precenza ho creato un nuovo controller, che si differenzia dal precedente per il tipo di notifica di cui deve tenere traccia anche se il principio di design rimane lo stesso. Qui ad esempio le azioni non sono continuative, ma si potrebbero definire atomiche perché avvengono istantaneamente se le condizioni richieste in seguito nella parte che fa da ponte tra controller e model (ActionComponent) sono soddisfatte.

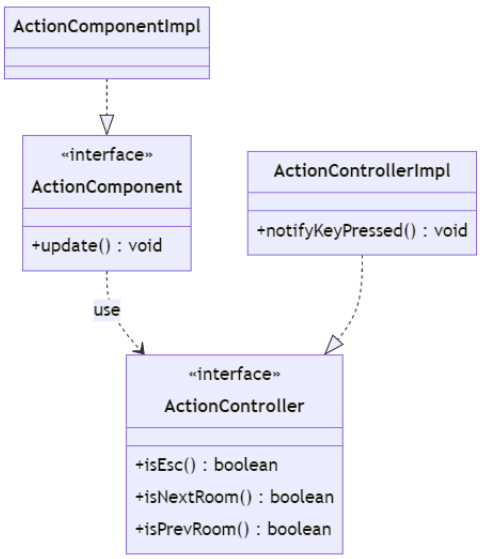

Figura 2.7: Rappresentazione UML della parte di ActionController

Problema Occorre schierare diversi tipi di elementi (es. enemy, boss, items ecc.) all'interno di una stanza, in modo casuale o ordinato in base al tipo di stanza.

Soluzione Ho sfruttato la generalizzazione di MapElemet che viene esteso da tutti gli elementi del gioco, in modo che a prescindere dal tipo di elemento si possa andare a settare le sue coordinate all'interno di una stanza di cui vengono passate le dimensioni come parametri del metodo "setPosition". Dall'interfaccia Spawn ne implemento due sottoclassi diverse (specializzazione), una per gli elementi che devono essere schierati in modo ordinato e una per quello casuale. Sarà poi la Room che dopo aver creato gli elementi chiamerà il metodo della sottoclasse che le serve. Per esempio una ShopRoom e una TreasureRoom andranno a chiamare il metodo dalla classe SpawnOrdered, mentre la stanza di nemici normale andrà chiamare il metodo dalla classe SpawnRandom.

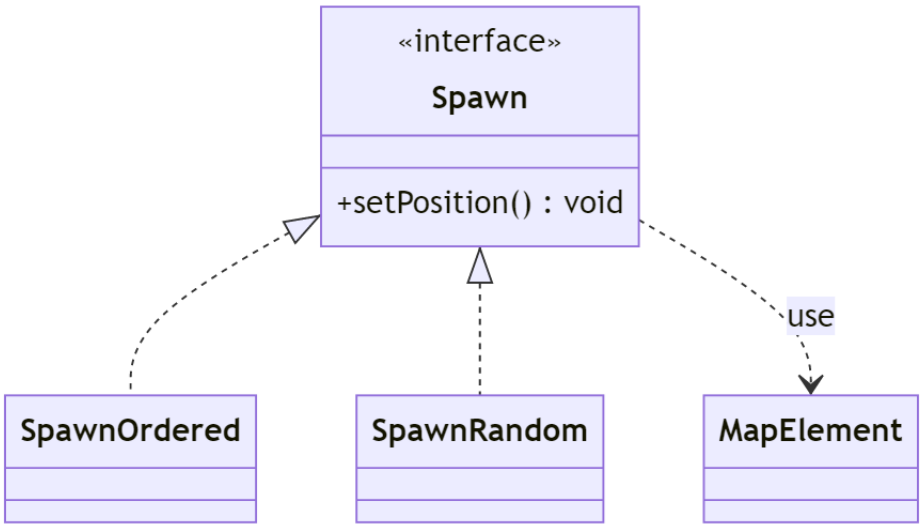

Figura 2.8: Rappresentazione UML per lo schieramento di MapElemnt dentro la stanza

Problema Occorre effettuare l'acquisto dei PowerUp da parte del Player, controllando che quest'ultimo abbia abbastanza monete e nel caso affermativo applicare il PowerUp al Player.

Soluzione In questo caso ho creato una interfaccia Shop che definisce il metodo "buyItem", il quale ha come parametri il Player e un PowerUp. La classe ShopImpl implementa questo metodo andando a controllare la quantità delle monete del player con il costo del PowerUp. Nel caso il Player abbia abbastanza monete avviene l'aquisto andando a rimuovere dalle monete del Player il prezzo del PowerUp e in seguito esegue il metodo "interact" del PowerUp che applica al Player il potenziamento.

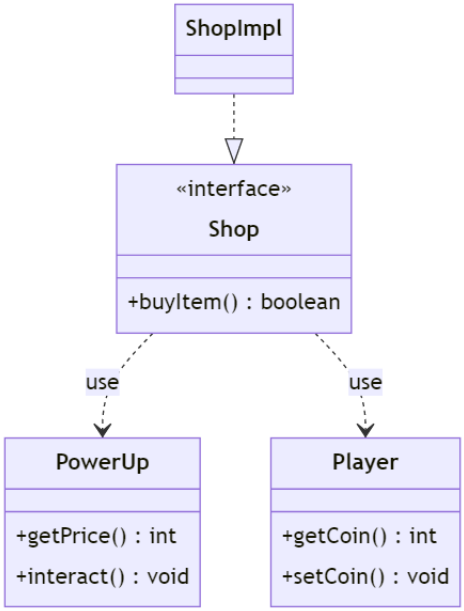

Figura 2.9: Rappresentazione UML per la funzione di acquisto dei PowerUp da parte del Player

Problema Occorre effettuare un controllo per capire se avviene una collisione tra due oggetti in uno spazio bidimensionale.

Soluzione Per risolvere questo problema abbiamo visionato il progetto del professor Ricci: `https://github.com/aricci303/game-as-a-lab/blob/master/Game-As-A-Lab-Step-7-mosquito-AI/src/rollball/model/BoundingBox.java`, rifattorizzandolo in modo oppurtuno. Ho implementato l'interfaccia BoundingBox in due sottoclassi (RectBoundingBox e CircleBoundingBox) che si differenziano per la forma del BoundingBox, uno circolare e uni rettangolare. Entrambi ereditano i metodi isCollidingWithCricle e isCollidingWithRecPerimeter che ovviamente vengono implementati in modo diverso a seconda della sottoclasse.

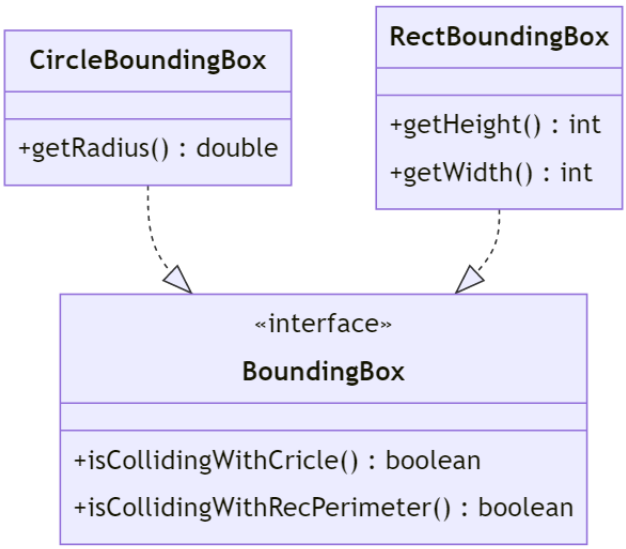

Figura 2.10: Rappresentazione UML del controllo collisioni tra due BoundingBox

Daniele Pancottini

Creazione e gestione AI per nemici

Problema Occorre modellare la gestione automatica del movimento e dello sparo dei nemici e del boss, generalizzando per quanto possibile il concetto di sparo rendendolo comune anche al player

Soluzione Si è creata l'interfaccia `AIEnemy` che rappresenta il concetto di AI per nemici e boss, l'implementazione di questa interfaccia `ConcreteAIEnemy` contiene una lista di nemici che sono gli stessi gesiti dall' AI, viene inoltre implementato il metodo per aggiornare lo stato dei nemici (sparo e movimento), quest'ultimo riceve dal chiamante informazioni riguardo la posizione del player ed il bounding box della stanza, in modo tale da realizzare correttamente le strategie di movimento e sparo. I nemici sono stati modellati tramite un'interfaccia `Enemy` che contiene i metodi per ottenere lo stato del nemico, ed i metodi per gestire il movimento e lo sparo. Sono stati inoltre divisi i nemici in tre tipi: nemici che sparano, che non sparano e boss (ibrido tra nemico che spara e che non spara). Le azioni di movimento e sparo sono state delegate a delle strategie apposite (utilizzando il pattern "Strategy") in modo da separare in modo corretto le responsabilità, e generalizzare il concetto di sparo e movimento per permettere il riuso anche da parte del

player e del boss (che essendo un ibrido, aveva bisogno sia del movimento del non shooting enemy, sia del movimento dello shooting enemy). Si è inoltre generalizzato il concetto di armi e colpi tramite l'interfaccia `Weapon` e `WeaponShot`, il concetto di arma viene utilizzato dalla strategia di sparo (utilizzata allo stesso modo dai nemici e dal player), al momento è stato necessario modellare un solo tipo di arma: quella con sparo ad intervalli di tempo regolari. Allego uno schema UML che chiarificherà la soluzione al problema, per lo schema delle strategie rimandiamo alla figura 2.16 e 2.18

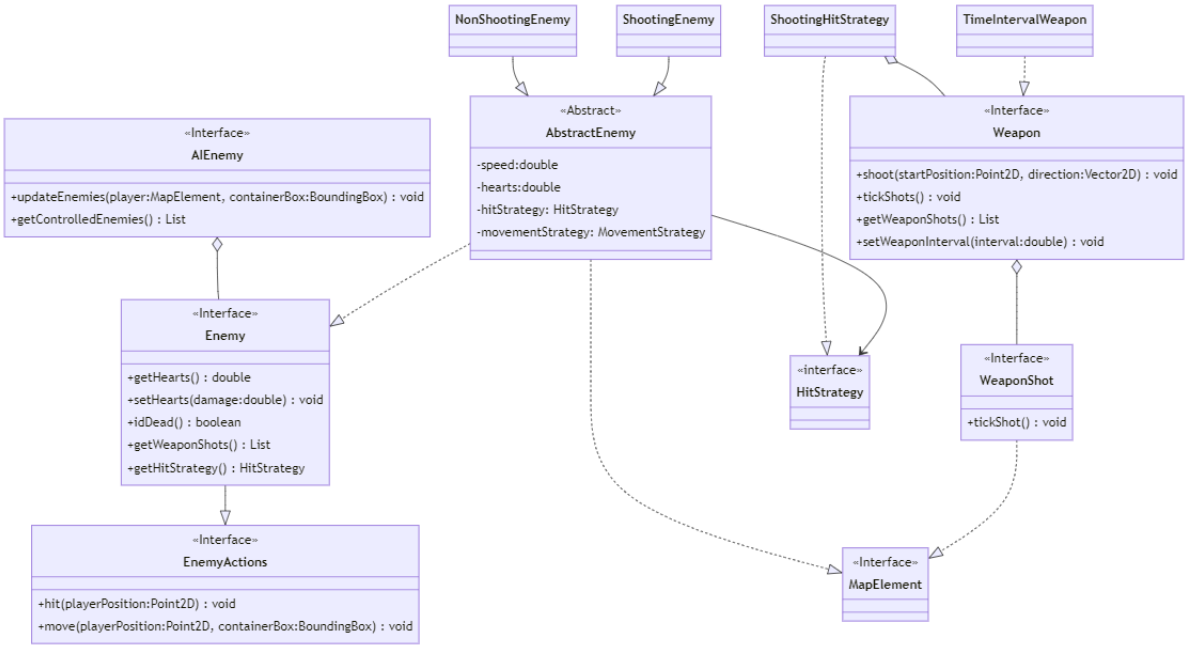

Figura 2.11: Rappresentazione UML dell'AI per nemici e boss con l'aggiunta della modellazione del concetto di arma e colpo

Gestione degli eventi in caso di collisione

Problema Occorre individuare le eventuali collisioni tra elementi della mappa, con particolare attenzione alle collisioni tra player e nemici

Soluzione Per quanto riguarda la parte da me realizzata, è stato modellato il concetto di evento (che scaturisce da una collisione avvenuta) tramite l'interfaccia `Event`, quest'ultima è un'interfaccia funzionale che definisce il comportamento dell'evento (il metodo che realizza il corpo dell'evento riceve un riferimento alla stanza in cui è avvenuta la collisione, questo perché la stanza funge da contenitore e memorizza tutti i map element presenti in essa), per garantire l'estendibilità la creazione degli eventi è delegata ad una factory `EventFactory`, interfaccia che modella il concetto di factory degli eventi, e dichiara tutti i metodi necessari alla loro creazione. Per lo schema UML di questa parte rimando alla figura 2.4

Creazione di nemici e boss

Problema Occorre creare nemici e boss delineando una modellazione comune con la creazione degli altri elementi della mappa

Soluzione Per risolvere questo problema si è pensato di utilizzare una factory di `Creator`, interfaccia che modella le entità delegate alla creazione di elementi della mappa, questo permette di tenere la struttura piuttosto flessibile, soprattutto in ottica futura, se si dovesse decidere di aggiungere altri map element e quindi altre `Creator`. Per la rappresentazione UML di questa parte rimandiamo alla figura 2.5

Gestione grafica di nemici, boss e proiettili

Problema Occorre gestire graficamente i nemici, il boss ed i proiettili, rendendo la gestione comune agli altri elementi della mappa

Soluzione Per risolvere questo problema è stata nuovamente utilizzata una factory, questo per rendere comune la gestione grafica a tutti gli elementi della mappa, e rendere più facile l'aggiunta di nuovi elementi in ottica futura. Per i nemici, il boss ed i proiettili è stata creata una `EnemyGraphicsComponentFactory` con tutti i metodi necessari alla creazione di `GraphicsComponent` da attaccare agli elementi della mappa in esame, in linea generale è stata creata una factory per ogni macro categoria di map element: item, powerup, nemici, player, room. Viene di seguito mostrato un diagramma UML relativo alla sola macro categoria dei nemici, ma la modellazione è identica anche per le altre macro categorie.

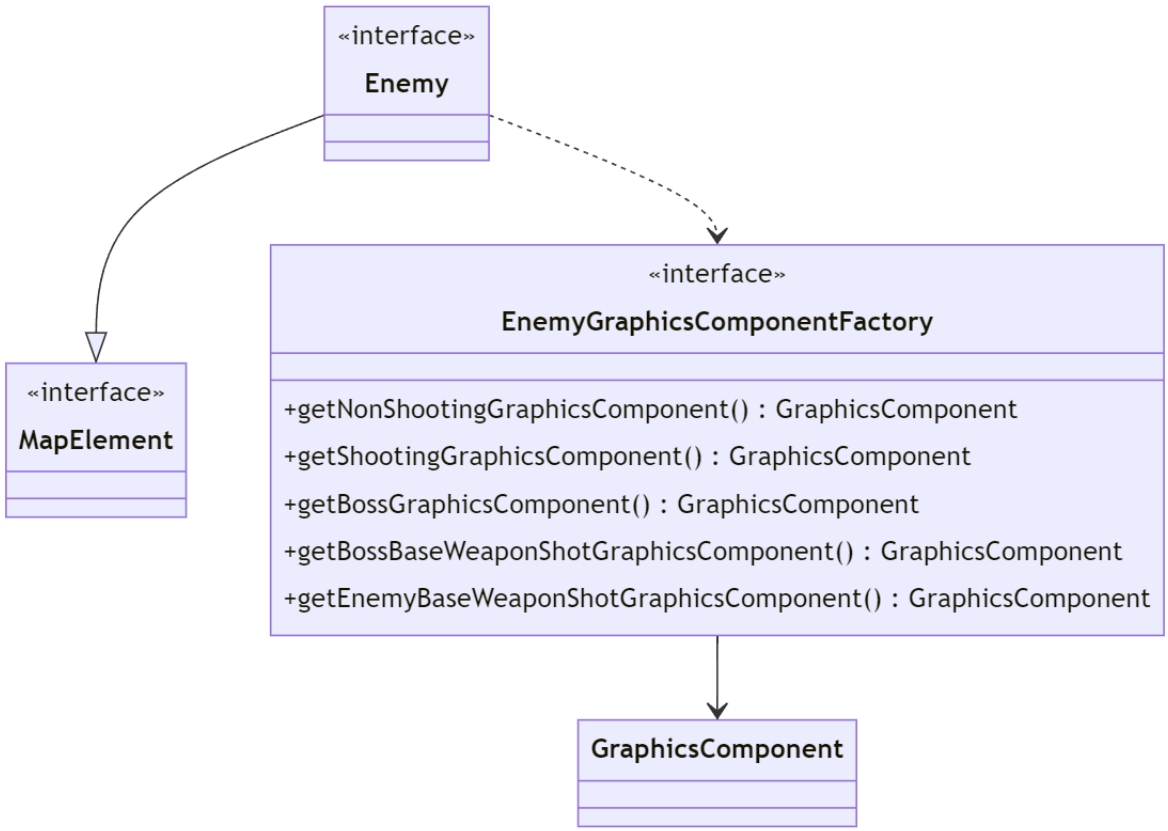

Figura 2.12: Rappresentazione UML della gestione grafica della macro categoria dei nemici

Gestione della pausa durante il gioco

Problema Occorre gestire la pausa durante il gioco

Soluzione La soluzione prevede l'aggiunta, nello stato del game loop, di un campo per memorizzare lo stato di pausa del gioco (questo implica anche l'aggiunta di metodi setter e getter per la modifica e la lettura del campo). Utilizzando l'interfaccia `ActionController` viene verificata la pressione del tasto ESC che, se premuto, manda in pausa il gioco o recupera dalla pausa, in base allo stato corrente del game loop. Si noti che non viene effettuata nessuna operazione sul thread che esegue il game loop, ma semplicemente non viene aggiornato lo stato del gioco e non vengono eseguiti gli eventi se il gioco stesso è in pausa.

Giacomo Pierbattista

Creazione delle stanze

Problema Occorre creare una stanza rispettando certi requisiti

Soluzione Come descritto nella sezione "requisiti funzionali", in questo gioco sono previsti 5 tipi di stanza, ognuno con le sue caratteristiche.
Ho utilizzato il pattern Builder per creare le stanze "a basso livello", nel senso che la classe RoomBuilder (insieme alla classe di utility RoomBuilderUtils) si è occupata di creare fisicamente la stanza, nascondendo la logica dietro la creazione della stanza stessa. In questo modo, le altre classi non hanno bisogno di sapere come deve essere costruita una stanza. RoomBuilder crea un stanza usando il cosiddetto "fluent style" [1]: per costruire una stanza è necessario innanzitutto chiamare i metodi obbligatori (necessari per tutti i tipi di stanza); successivamente, a seconda della stanza da creare, devono essere chiamati gli altri metodi che si occuperanno di impostare le caratteristiche rimanenti. Chiamando il metodo `build()` la stanza verrà effettivamente creata solo se la configurazione è corretta.
Il pattern Builder, quindi, porta alcuni vantaggi:

- non vengono mai create stanze mal configurate
- nasconde la logica dietro la creazione della stanza stessa
- è possibile cambiare la configurazione della stanza (e i relativi controlli).

[1] `https://en.wikipedia.org/wiki/Fluent_interface`

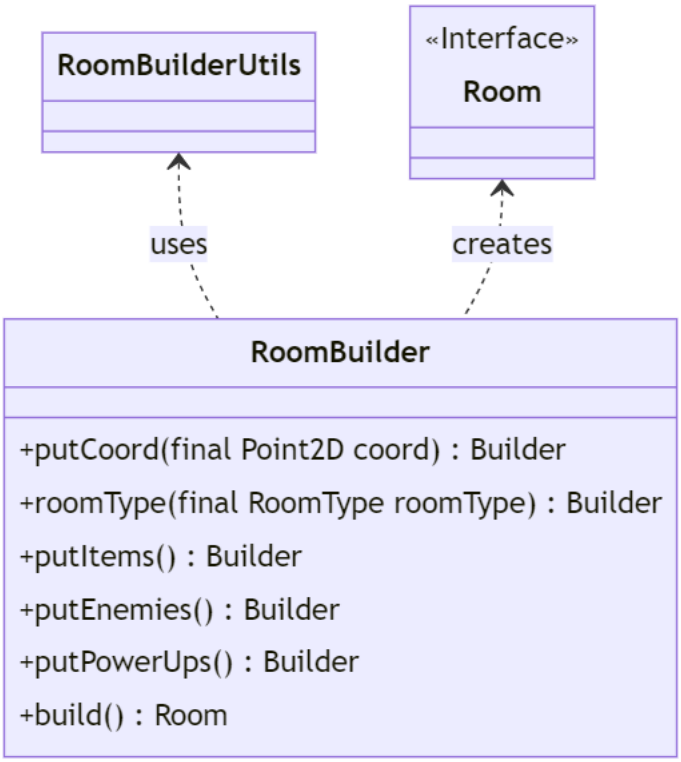

Figura 2.13: Rappresentazione UML del pattern Builder per la creazione delle stanze a "basso livello"

Problema Occorre creare un numero random di stanze, ognuna con le sue caratteristiche, nascondendone la logica.

Soluzione Nel punto precedente, RoomBuilder crea le stanze a "basso livello"; occorre qualcosa che permetta di istanziarle ad "alto livello", incapsulando la procedura di creazione delle stesse.
Ho utilizzato il pattern Factory Method: l'interfaccia RoomFactory offre un metodo per creare ogni tipo di stanza (`buildStartRoom(), buildStandardRoom()...`), più un metodo "intelligente" `buildRoomInProperOrder()`, che crea una stanza del tipo appropriato ogni volta che viene chiamato.
RoomFactory, per creare le stanze, si appoggia a RoomBuilder precedentemente descritto: in RoomFactory è incapsulata la procedura per la creazione delle stanze, mentre RoomBuilder si limita a creare le stanze, seguendo le indicazioni contenute in RoomFactory.

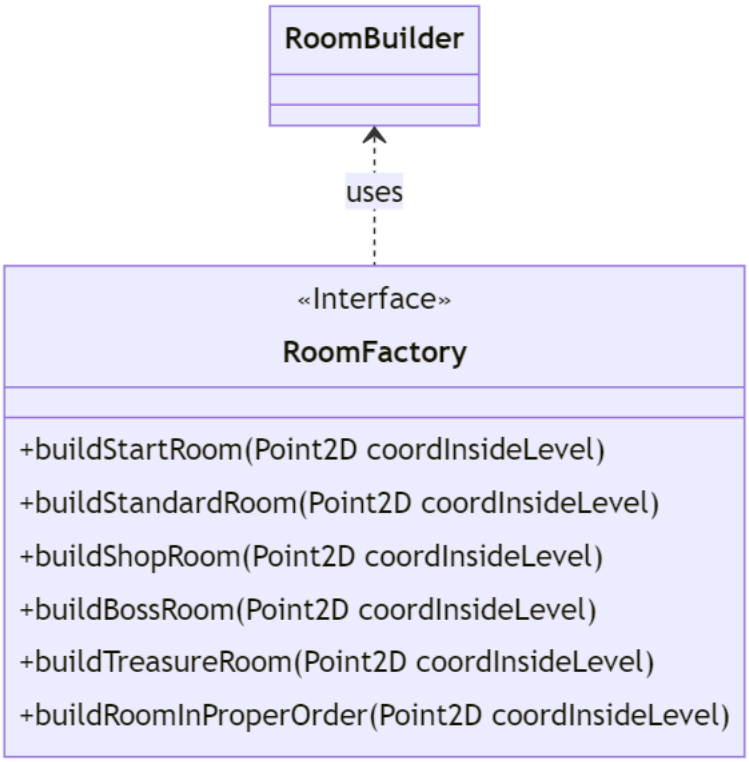

Figura 2.14: Rappresentazione UML del pattern Factory Method per la creazione delle stanze ad "alto livello"

Problema Le stanze hanno bisogno di una coordinata all'interno del livello.

Soluzione Poiché tutte le entità del gioco (nemici, powerup e item) hanno una coordinata, è stata creata l'interfaccia MapElement, che permette di avere un getter e un setter per la coordinata. L'interfaccia Room estende MapElement, in modo tale che le entità di cui sopra e le stanze abbiano lo stesso metodo per accedere e settare la coordinata, evitando così inutili ripetizioni codice e rispettando il principio DRY - Don't Repeat Yourself. La coordinata è necessaria in quanto indica la posizione della stanza stessa all'interno del Livello. Questa scelta è dovuta al fatto che, per trovare una soluzione valida per il contesto, ho immaginato il livello come una griglia 2D, in cui ogni casella ha una coordinata (x, y). Per effetto dello sviluppo incrementale, prima abbiamo realizzato un gioco funzionante posizionando in orizzontale le stanze una dietro l'altra (quindi tutte le stanze hanno una coordinata (x, 0), con $x \geq 0$). Per mancanza di tempo, non abbiamo implementato la possibilità di avere un livello con le stanze disposte in una griglia

2D, ma le coordinate utili per tale scopo sono rimaste. Tutta la parte di model riguardante le stanze e il livello è stata pensata per poter funzionare anche per livelli con stanze disposte non necessariamente in linea.

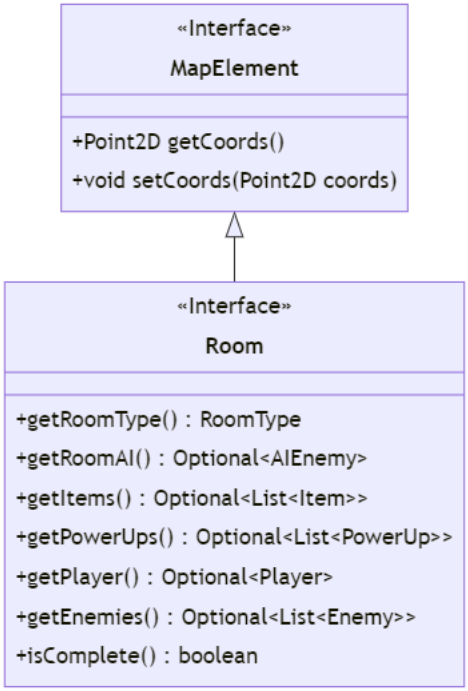

Figura 2.15: Rappresentazione UML dell'interfaccia Room che estende l'interfaccia MapElement, per ereditarne i metodi evitando ripetizioni

Creazione del mondo di gioco (livello/level)

Problema E' necessario creare il livello in modo dinamico, nascondendone la logica.

Soluzione Anche in questo caso ho utilizzato il pattern Factory Method. Infatti, LevelFactory si occupa di creare un livello attraverso il metodo `createLevel()`, ma internamente usa il metodo `buildRoomInProperOrder()` di RoomFactory per creare le stanze richieste, il quale, a sua volta, si occuperà di costruire e restituire la stanza appropriata. In questo modo, anche il livello viene creato senza che le altri classi si preoccupino di come farlo effetivamente.

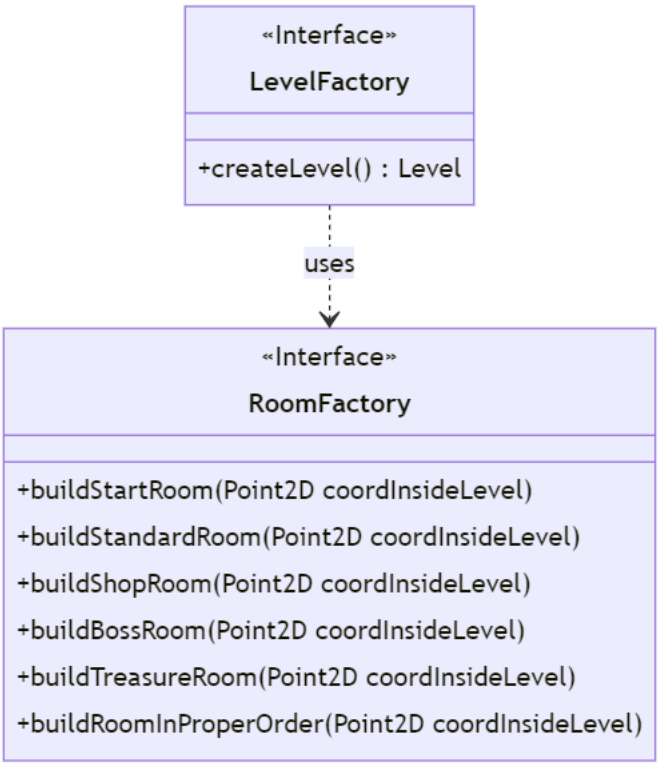

Figura 2.16: Rappresentazione UML del pattern Factory Method per la creazione dei livelli

Gestione di un livello

Problema Occorre gestire lo stato di un livello

Soluzione La stanza è l'unità fondamentale di gioco: tutto si svolge al suo interno. Nel mondo di gioco esistono più stanze: le stanze nel loro insieme costituiscono il "livello".
Il livello gestisce il suo stato interno (cioè riesce a determinare in quale stanza si trova il giocatore, se una certa stanza è completa, se tutto il livello è completo..). [2] Ho realizzato anche l'interfaccia Minimap, la quale semplifica l'ottenimento di informazioni riguardanti lo stato del livello stesso, che saranno poi mostrate nella GUI (le stanze completate, non completate e quella in cui si trova il giocatore).

[2]Una stanza è completa se non sono rimasti nemici da sconfiggere;
un livello è completo se tutte le stanze che lo compongono sono complete.

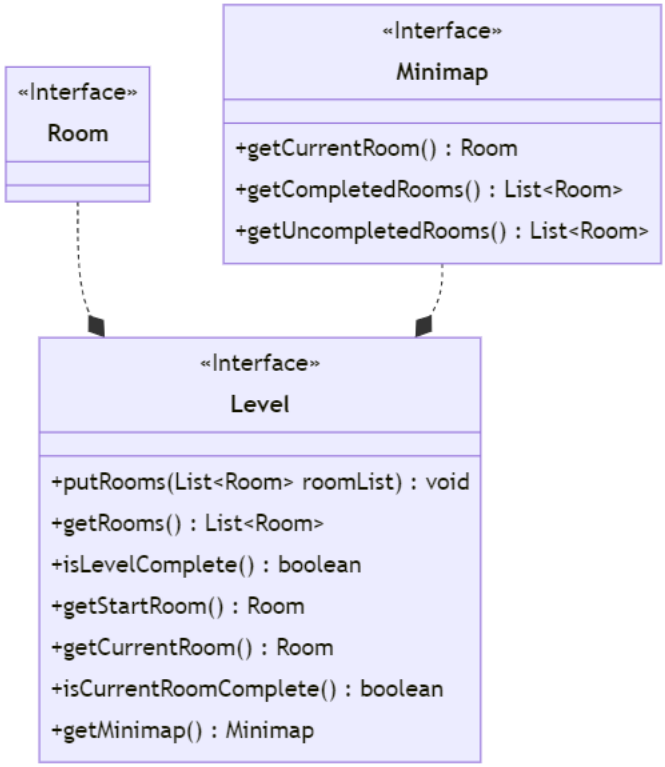

Figura 2.17: Rappresentazione UML del Livello che incapsula una Minimap per poter ottenere più facilmente le informazioni riguardanti lo stato del livello

Gestione di più livelli

Problema Occorre gestire più livelli dinamicamente

Soluzione Come descritto precedentemente, un singolo livello riesce a gestire il suo stato. Ho progettato l'interfaccia LevelController pensando di generalizzare il più possibile: un LevelController, infatti, può gestire più livelli, che idealmente vengono giocati uno dopo l'altro. Quest'ultima funzionalità, però, non è stata implementata completamente in quanto era prevista come requisito opzionale, e per mancanza di tempo non è stata inserita effettivamente nel gioco. LevelController si occupa di creare uno o più livelli, a seconda della configurazione interna, utilizzando LevelFactory: anche in questo caso ho delegato la creazione di un oggetto a una classe/interfaccia apposita. Poiché il LevelController incapsula i vari livelli generati, offre metodi che richiamano direttamente i metodi corrispondenti del Level attual-

mente giocato, mentre altri sono metodi utili nel caso in cui si implementi la gestione effettiva di livelli multipli (ad esempio: `areAllLevelsComplete()`).

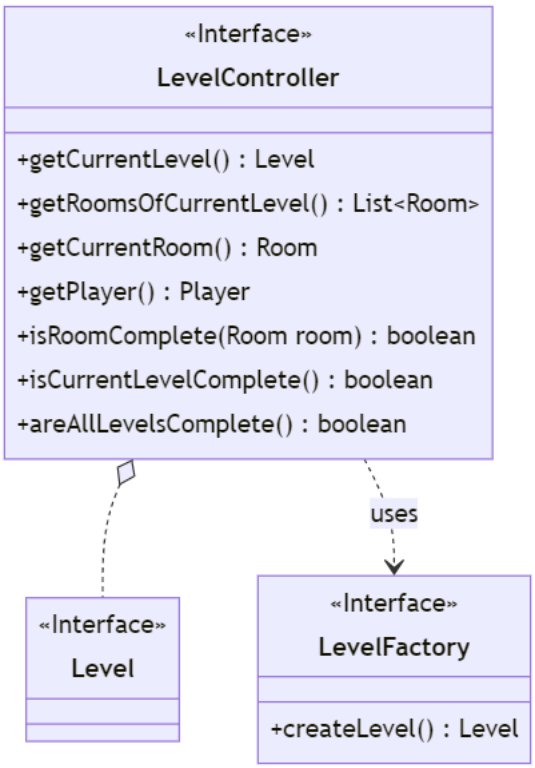

Figura 2.18: Rappresentazione UML dell'incapsulamento di Level e Minimap all'interno di LevelController

Dilaver Shtini

Creazione e gestione del Player

Problema Il concetto di sparo è in comune anche con i nemici e il boss, quindi, non può essere specifico solo per il player, deve essere generico, inoltre il movimento deve essere gestito diversamente da quello dei nemici.

Soluzione Per lo sparo viene utilizzato il pattern "Strategy" dove viene creata un'interfaccia funzionale con il metodo "hit" e due sottoclassi che la implementano, "NonshootingHitStrategy" e "ShootingHitStrategy", così da poter essere utilizzato sia dal player, il quale userà lo "ShootingHitStrategy", che dai nemici, che sono di tue tipi, e dal boss, che, come il player, utilizzerà

lo “ShootingHitStrategy”. Per il movimento invece viene creata un’interfaccia “MapElement”, che viene usata da tutti gli elementi che hanno una posizione nella mappa, per gestire le coordinate, che viene implementata da una classe astratta “AbstractMapElement”, la quale aggiunge il raggio e il boundingBox, estesa dal player.

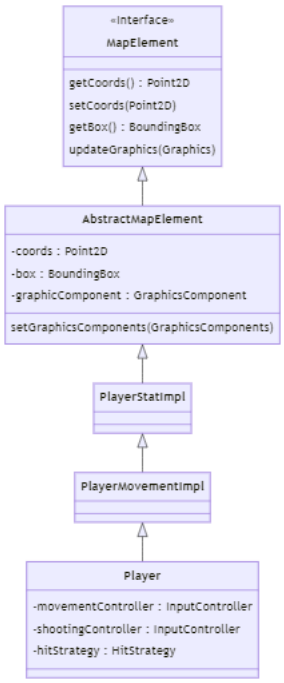

Figura 2.19: Creazione del player

Creazione e gestione del Boss

Problema Gestire il cambio del tipo di movimento e lo sparo.

Soluzione Per il movimento dei nemici, quindi anche del Boss, viene utilizzato il pattern "Strategy" creando un'interfaccia funzionale con il metodo "move", implementata poi dai due tipi di movimento: "NonShootingMovementStrategy" e "ShootingMovementStrategy", che prende in ingresso la posizione attuale del nemico e la posizione del player, ovvero la posizione verso cui andrà il nemico nella modalità "NonShootingMovementStrategy". Nel Boss, inoltre, si crea una mappa per gestire i due tipi di movimento e un metodo, "changeMode", che si appresta al cambio del tipo ogni n secondi. Per lo sparo invece, simile al Player, si crea la strategia di tipo "ShootingHitStrategy".

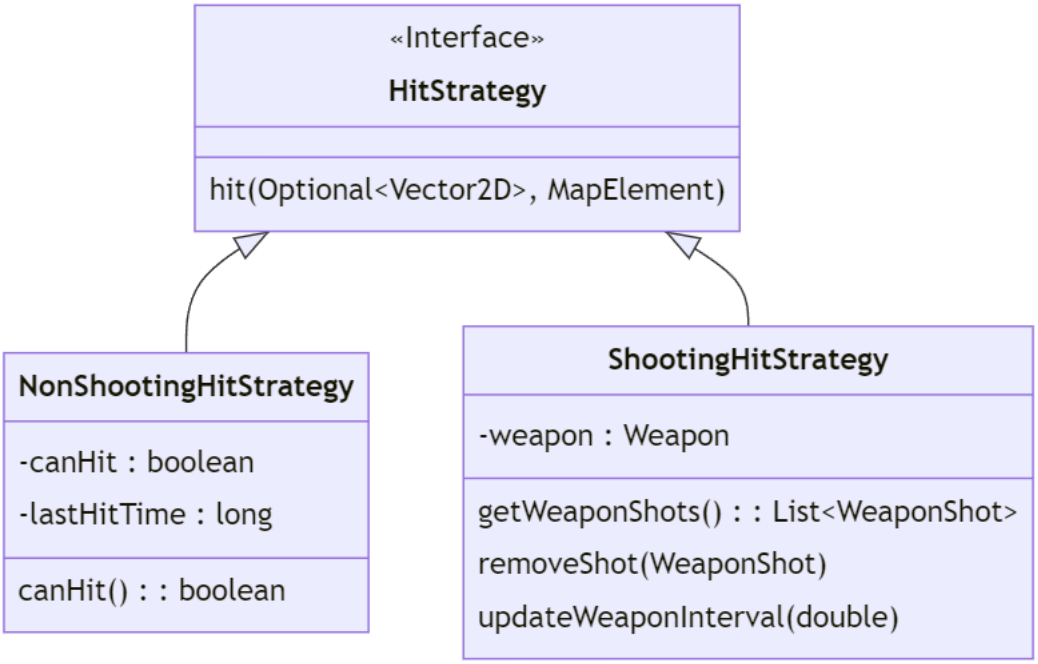

Figura 2.20: Pattern strategy per il colpo

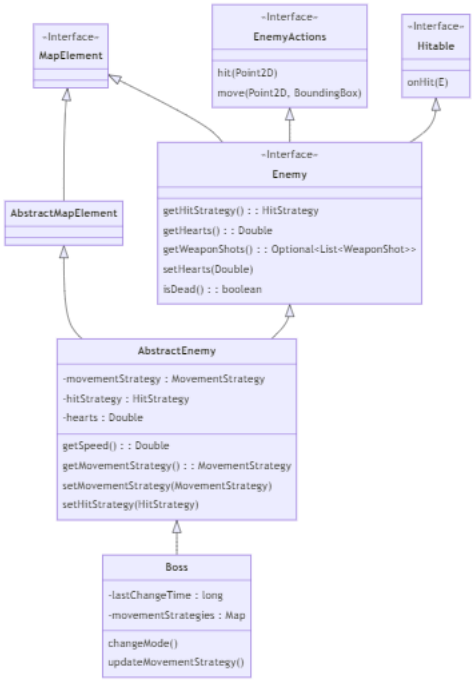

Figura 2.21: Creazione del boss

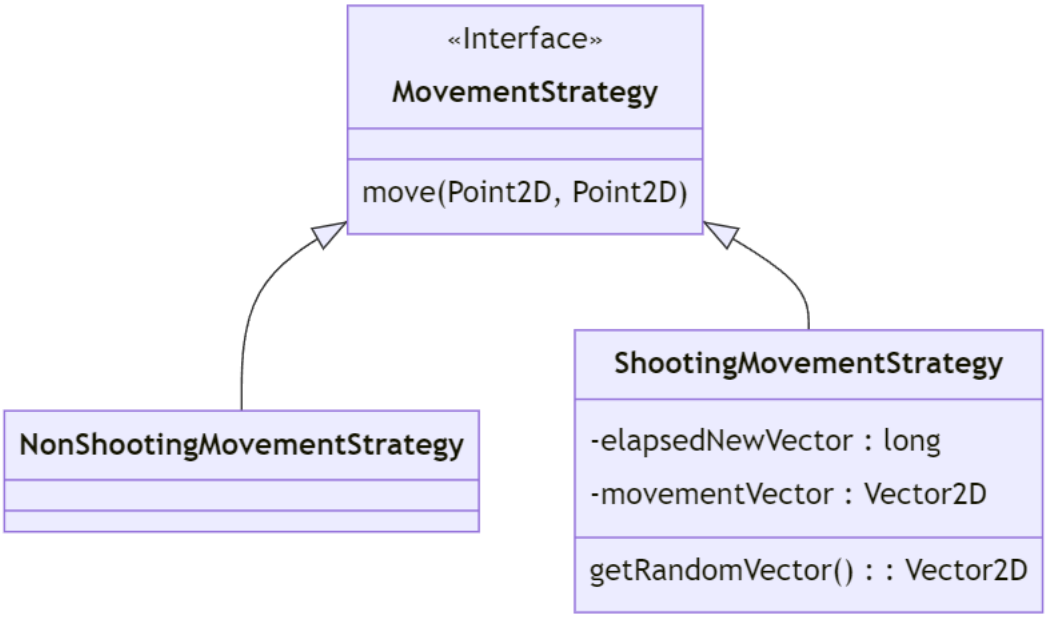

Figura 2.22: Pattern strategy per il movimento

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Sono state realizzate diverse classi di test automatizzato, utilizzando JUnit 5. Non sono stati realizzati test automatici sulla GUI per mancanza di tempo. I test sono stati realizzati secondo quanto segue:

Marco Costantini

- TestItem

Andrea Dotti

- TestShop

Daniele Pancottini

- TestEnemy

Giacomo Pierbattista

Ognuno dei seguenti file contiene un test per ogni metodo delle rispettive interfacce, in modo da verificare il corretto funzionamento di ogni singolo metodo.

- LevelControllerTest
- LevelFactoryTest
- LevelTest

- RoomBuilderTest
- RoomFactoryTest
- RoomTest

Dilaver Shtini

- TestPlayer

3.2 Metodologia di lavoro

Il lavoro è stato diviso tra i componenti del gruppo all'inizio, in modo da permettere a tutti di lavorare indipendentemente dagli altri il più possibile. E' stato necessario riunirsi per discutere sull'implementazione e/o sulla dichiarazione di interfacce e classi comuni.

Abbiamo utilizzato lo strumento di version control **Git** per poter sviluppare il software in modo più agevole.
Il branch principale è stato denominato *master*. Dal *master* sono stati creati vari branch, chiamati "feature-featureName"; ognuno di essi identificava la feature che doveva essere implementata. La divisione delle feature tra i vari componenti del gruppo è stata tale da permettere a ognuno di lavorare nel proprio branch indipendentemente dagli altri.
Abbiamo lavorato in modo tale che tutte le modifiche venissero apportate nei vari branch "feature-featureName"; solo dopo aver testato che tutto funzionasse correttamente, tali modifiche sono state unite sul branch *master*.
In questo modo, il software contenuto nel branch *master* funzionava sempre correttamente.

Marco Costantini

In autonomia, ho sviluppato

- `it.unibo.isaccoop.core.*`

  esclusa la funzione di pausa in GameLoopImpl.java
- `it.unibo.isaccoop.model.item.*`
- `it.unibo.isaccoop.model.powerup.*`

Ho sviluppato:

- in collaborazione con Daniele Pancottini:
  - `it.unibo.isaccoop.model.creator.*`
  - `it.unibo.isaccoop.graphics.factory.*`
- in collaborazione con Daniele Pancottini e Andrea Dotti:
  - `it.unibo.isaccoop.model.collision.*`

Andrea Dotti

Mi sono concentrato principalmente dello spawn di nemici, powerup e item nelle varie stanze, ho gestito gli input da tastiera del giocatore. Mi sono focalizzato anche sul controllo delle collisioni con i BoundingBox, che sono serviti ai miei compagni nella CollisionCheck. In autonomia ho sviluppato:

- `Shop.java e ShopImpl.java`
- `it.unibo.isaccoop.input.*`
- `it.unibo.isaccoop.model.spawn.*`
- `it.unibo.isaccoop.model.boundingBox.*`

Ho sviluppato:

- in collaborazione con Marco Costantini e Daniele Pancottini
- `it.unibo.isaccoop.model.collision.*`

Daniele Pancottini

Ho sviluppato:

- in collaborazione con Marco Costantini
  - `it.unibo.isaccoop.graphics.factory.*`
  - `it.unibo.isaccoop.model.creator.*`
- in collaborazione con Dilaver Shtini
  - `it.unibo.isaccoop.model.action.*`
  - `it.unibo.isaccoop.model.ai.*`
  - `it.unibo.isaccoop.model.enemy.*`

  - `it.unibo.isaccoop.model.weapon.*`

- in collaborazione con Marco Costantini e Andrea Dotti

  `it.unibo.isaccoop.model.collision.*`

- la funzione di pausa in GameLoopImpl.java

Giacomo Pierbattista

Mi sono concentrato principalmente sulla progettazione del mondo di gioco (Level) e delle stanze (Room). In autonomia, nel model, ho sviluppato le classi contenute nel package

`it.unibo.isaccoop.model.room.*`

(escludendo Shop.java e ShopImpl.java) che implementano quanto segue:

- creazione dinamica della mappa di gioco/livello, costituita da stanze
- passaggio del giocatore da una stanza all'altra
- minimappa.

e le enum Direction.java e RoomType.java contenute nel package

`it.unibo.isaccoop.model.common`

che implementano, rispettivamente, il concetto di direzione, utilizzato per il movimento del giocatore e dei nemici, e i vari tipi di stanza.
Nella view ho sviluppato quanto segue:

- guida utente (HelpGUI.java)
- pannello contenente la minimappa e le statistiche del giocatore (OverlayGUI.java)
- AbstractGUIFrame.java, GUIFrame.java

Dilaver Shtini

In autonomia ho sviluppato:

- `GameMenu.java`
- `it.unibo.isaccoop.model.player.*`

In collaborazione con Daniele Pancottini ho sviluppato:

- `it.unibo.isaccoop.model.action.*`
- `it.unibo.isaccoop.model.ai.*`
- `it.unibo.isaccoop.model.enemy.*`
- `it.unibo.isaccoop.model.weapon.*`

Parti sviluppate in collaborazione

- `SwingGraphics.java`
  e i file rimanenti riguardanti la view
- `it.unibo.isaccoop.model.common`

3.3 Note di sviluppo

Abbiamo iniziato lo sviluppo a partire dalla repository `https://github.com/unibo-oop/sample-gradle-project`, compreso il **Gradle wrapper** fornito, per poter eseguire la build del progetto.

Marco Costantini

Nello sviluppo ho utilizzato

- **Optional** ad esempio in:
  - in ConcreteCreatorFactory.java `https://github.com/jackprb/OOP22-isaccoop/blob/master/src/main/java/it/unibo/isaccoop/model/creator/ConcreteCreatorFactory.java`
- **Stream** ad esempio in:

  - in ConcreteCreatorFactory.java per la precisione nei metodi che riguardano gli item e i powerUp https://github.com/jackprb/OOP22-isaccoop/blob/master/src/main/java/it/unibo/isaccoop/model/creator/ConcreteCreatorFactory.java

  - in GameLoopImpl.java https://github.com/jackprb/OOP22-isaccoop/blob/master/src/main/java/it/unibo/isaccoop/core/GameLoopImpl.java

- **Lambda** ad esempio in:

  - in ConcreteCreatorFactory.java https://github.com/jackprb/OOP22-isaccoop/blob/master/src/main/java/it/unibo/isaccoop/model/creator/ConcreteCreatorFactory.java

  - in GameLoopImpl.java https://github.com/jackprb/OOP22-isaccoop/blob/master/src/main/java/it/unibo/isaccoop/core/GameLoopImpl.java

- **Generici** ad esempio in:

  - in Creator.java https://github.com/jackprb/OOP22-isaccoop/blob/master/src/main/java/it/unibo/isaccoop/model/creator/Creator.java

Andrea Dotti

Nello sviluppo ho utilizzato

- **Stream** ad esempio in:

  - in SpawnOrdered.java https://github.com/jackprb/OOP22-isaccoop/blob/master/src/main/java/it/unibo/isaccoop/model/spawn/SpawnOrdered.java#L28
  - in CollisionCheckFactoryImpl.java https://github.com/jackprb/OOP22-isaccoop/blob/master/src/main/java/it/unibo/isaccoop/model/collision/CollisionCheckFactoryImpl.java#L56

- **Lambda**

  - in RoomImpl.java https://github.com/jackprb/OOP22-isaccoop/blob/master/src/main/java/it/unibo/isaccoop/model/room/RoomImpl.java#L210

Daniele Pancottini

Nello sviluppo ho utilizzato

- **Optional** ad esempio in:
  - in Enemy.java in quanto non tutti gli enemy sparano e quindi non tutti i nemici possiedono proiettili `https://github.com/jackprb/OOP22-isaccoop/blob/master/src/main/java/it/unibo/isaccoop/model/enemy/Enemy.java#L40`
  - Gli Optional sono stati usati anche nella realizzazione delle strategie di movimento e sparo
- **Stream** ad esempio in:
  - in ConcreteCreatorFactory.java per i metodi di creazione dei nemici e boss `https://github.com/jackprb/OOP22-isaccoop/blob/master/src/main/java/it/unibo/isaccoop/model/creator/ConcreteCreatorFac` `java`
- **Lambda** ad esempio in:
  - in ConcreteCreatorFactory.java `https://github.com/jackprb/OOP22-isaccoop/blob/master/src/main/java/it/unibo/isaccoop/model/creator/ConcreteCreatorFactory.java`
  - in EventFactory.java `https://github.com/jackprb/OOP22-isaccoop/blob/master/src/main/java/it/unibo/isaccoop/model/collision/EventFactory.java`
- **Generici** ad esempio in:
  - in Creator.java `https://github.com/jackprb/OOP22-isaccoop/blob/master/src/main/java/it/unibo/isaccoop/model/creator/Creator.java`
  - in Hitable.java `https://github.com/jackprb/OOP22-isaccoop/blob/master/src/main/java/it/unibo/isaccoop/model/enemy/Hitable.java`

Giacomo Pierbattista

Nello sviluppo ho utilizzato

- **Optional** ad esempio:
  - frequentemente in RoomBuilder.java (da riga 28 in poi) `https://github.com/jackprb/OOP22-isaccoop/blob/master/src/main/java/it/unibo/isaccoop/model/room/RoomBuilder.java#L34`
  - e in RoomImpl.java (da riga 28 in poi) `https://github.com/jackprb/OOP22-isaccoop/blob/master/src/main/java/it/unibo/isaccoop/model/room/RoomImpl.java#L28`
- **Stream** ad esempio:
  - in LevelTest.java (in vari punti da riga 75 in poi) `https://github.com/jackprb/OOP22-isaccoop/blob/master/src/test/java/it/unibo/isaccoop/test/model/room/LevelTest.java#L75`
  - in LevelControllerImpl.java (righe 26-28, 58-60) `https://github.com/jackprb/OOP22-isaccoop/blob/master/src/main/java/it/unibo/isaccoop/model/room/LevelControllerImpl.java#L26`
- **Lambda**
  - in RoomImpl.java (righe 132-133, 151-153) `https://github.com/jackprb/OOP22-isaccoop/blob/master/src/main/java/it/unibo/isaccoop/model/room/RoomImpl.java#L132`
  - in OverlayGUI.java (righe 103-107, 111-122) `https://github.com/jackprb/OOP22-isaccoop/blob/master/src/main/java/it/unibo/isaccoop/graphics/OverlayGUI.java#L103`
- **Wildcard**
  - in RoomBuilderUtils.java `https://github.com/jackprb/OOP22-isaccoop/blob/master/src/main/java/it/unibo/isaccoop/model/room/RoomBuilderUtils.java#L107` (linee 107 e 117).
    Nelle signature dei metodi ho utilizzato le wildcard per poter gestire una lista di MapElement e relative sottoclassi.

Per poter caricare il file contenente la guida utente dal filesystem, ho utilizzato **ClassLoader** in HelpGui.java; ho guardato qui

https://github.com/unibo-oop/example-with-get-resources/blob/master/src/main/java/it/unibo/getresource/UseGetResource.java#L35.

Per ridimensionare un Jframe a seconda della risoluzione dello schermo in AbstractGUIFrame.java ho guardato qui: https://github.com/unibo-oop/lab08/blob/solutions/83-mvc-io/src/main/java/it/unibo/mvc/SimpleGUI.java#L83 (linee 83-102).

Dilaver Shtini

Nello sviluppo ho utilizzato

- **Lambda** ad esempio in:
  - https://github.com/jackprb/OOP22-isaccoop/blob/master/src/main/java/it/unibo/isaccoop/model/player/Player.java#L38

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Marco Costantini

Mi sento soddisfatto del lavoro fatto, il lavoro in team è stato molto interessante e stimolante, in generale mi sento di aver contribuito in modo attivo nel gruppo. Questo progetto mi ha aiutato anche ad imparare nuovi concetti, e capire l'importanza di altri, come ad esempio la parte di progettazione che fondamentale prima di mettersi a scrivere codice, o l'utilizzo di pattern che non avevo mai testato.

Andrea Dotti

Questo progetto è stato sicuramente il più importante tra quelli a cui io ho preso parte, sia dal punto di vista delle dimensioni che della difficoltà. Credo di essere riuscito ad apportare il mio contributo al team. Uno degli aspetti che penso mi sia sarà utile in futuro è l'aver imparato ad utilizzare il DVCS (Git), che mi ha dato la possibilità di apprezzare la grande facilità che offre per lo sviluppo in parallelo del codice, senza che nel mentre avvengano dei conflitti. Ritengo inoltre di poter migliorare ulteriormente e arrivare ad essere un buon progettista/programmatore a oggetti. Durante il progetto ho cercato il pi'u possibile di seguire i fondamenti della programmazione/modellazione a oggetti, come ad esempio la ricerca della massima riusabilità, l'utilizzo di specializzazioni e il principio di design: "keep it simple".

Daniele Pancottini

Sono soddisfatto del risultato ottenuto, per me è stata la prima esperienza con un progetto in gruppo di queste dimensioni, è stato bello risolvere i problemi di progettazione e successivamente implementare tutto discutendo continuamente con i miei compagni e vedere passo dopo passo la progressione nello sviluppo. Da questo progetto ho imparato quanto sia importante la progettazione, ai fini di uno sviluppo qualitativamente soddisfacente, ed ho acquisito un po' di esperienza, seppur basilare, riguardo al lavoro di gruppo.

Giacomo Pierbattista

Sono nel complesso soddisfatto di come ho realizzato la mia parte di model. Ho cercato di progettarla già dall'inizio nel modo più generale possibile, basandomi sui requisiti del dominio. Nonostante l'inesperienza, ritengo di aver realizzato codice non perfetto ma migliorabile: ho cercato il più possibile di seguire i principi della programmazione a oggetti, soprattutto la riusabilità per evitare ripetizioni. Questo progetto mi ha permesso di imparare sul campo l'uso di un DVCS (Git), che finora ho usato in altri corsi in modo più semplice.

Dilaver Shtini

Questa è stata la mia prima esperienza per quanto riguarda la realizzazione di un software, ho capito subito le mie limitazioni, dettate anche dalla poca esperienza, ma lavorare in gruppo mi ha aiutato molto a comprendere nuovi modi di riflettere e affrontare problemi. Lavorare in gruppo mi ha permesso di approfondire un tool comodo per gruppi di lavoro come git, che permettere di lavorare in sincrono con gli altri membri del gruppo ed evitare di lavorare in maniera indipendente. In questo progetto ho appreso l'importanza di una buona progettazione iniziale per la buona riuscita del software, infatti, ho dovuto rivedere e correggere la progettazione degli aspetti da me implementati sia per motivi semantici che per esigenze riscontrate durante la realizzazione del software. In conclusione, penso di aver aiutato il gruppo per il completamento dell'applicazione, in quanto ho partecipato alla progettazione inziale e alla maggior parte dei problemi che sono stati rinvenuti durante l'implementazione con discussioni e scambio di opinioni. Per quanto riguarda la mia parte non sono pienamente soddisfatto perché non ho potuto sfruttare appieno le potenzialità offerte da java optando avvolte per la soluzione più semplice rispetto alla più corretta e pulita.

4.2 Difficoltà incontrate e commenti per i docenti

Marco Costantini

Lavorare in team è stato molto costruttivo, siamo stati un gruppo molto affiatato fin dall'inizio dandoci supporto a vicenda. Il corso di Programmazione ad oggetti è, a mio parere, uno dei più importanti del corso di laurea e anche uno dei più impegnativi, ho trovato le lezioni stimolanti e interessanti, con professori sempre pronti a dare una dritta, grazie anche al forum per gli studenti su virtuale.

Andrea Dotti

A mio parere questo corso è uno dei più importanti del corso di laurea, se non il più importante, perché credo che sia quello che richiede dallo studente un maggior carico di lavoro sia durante le lezioni (dove ci si approccia ad un nuovo modo di modellare e pensare che è l'OOP), che individualmente a casa (lo sviluppo di un progetto richiede una conoscenza significativa della modellazione a oggetti oltre all'abilità di riuscire a tradurre in seguito il modello in linguaggio Java).

Daniele Pancottini

Le difficoltà incontrate durante lo sviluppo di questo progetto non sono state poche, fortunatamente discutendo e ragionando tutti insieme siamo riusciti piano piano a superarle tutte e ad arrivare ad un buon risultato. Il corso di programmazione ad oggetti mi è piaciuto particolarmente, oltre che per gli argomenti trattati e per l'importanza del corso stesso, per le modalità d'insegnamento, volte a far capire davvero allo studente il "perché" delle nozioni studiate e non soffermandosi solo sulla produzione di codice.

Giacomo Pierbattista

Per me è stato il primo progetto di gruppo di dimensioni consistenti, che ha richiesto un grande impegno.
Parere sul corso: prevedere un esame in laboratorio più un progetto di gruppo di queste dimensioni lo rende più impegnativo di altri corsi da 12 crediti, ma ne ero consapevole.
Tutto sommato, il corso è stato molto interessante, ma forse alcuni argomenti trattati soffrono di "anzianità": potrebbe essere più utile sviscerare le nuove

librerie di Java oppure soluzioni equivalenti di terze parti, specialmente se possono essere utili nello sviluppo del progetto. Non avevo mai programmato in Java, né in altri linguaggi Object Oriented; ho avuto bisogno di un po' di tempo per familiarizzare con le novità di Java, che in altri linguaggi non si trovano.

Dilaver Shtini

Sicuramente è stato il progetto più impegnativo fin'ora affrontato, farlo in gruppo è di certo un grande aiuto, non solo per la realizzazione dell'applicazione in sè ma anche per crescere e aumentare/migliorare le proprie conoscenze. Ho avuto difficoltà nell'avvio del progetto, ho dovuto riguardare vari concetti della programmazione ad oggetti, come ad esempio l'uso dei generici. Sono soddisfatto di questo corso perchè, almeno personalmente, ha aiutato a capire, specialmente con il progetto, dinamiche che prima non consideravo.

Appendice A

Guida utente

Menu principale

All'avvio dell'applicativo, l'utente si ritroverà nel menu di gioco, in cui troverà tre pulsanti:

- **Play** per avviare la partita
- **Help** per visualizzare la guida utente
- **Quit** per chiudere il gioco.

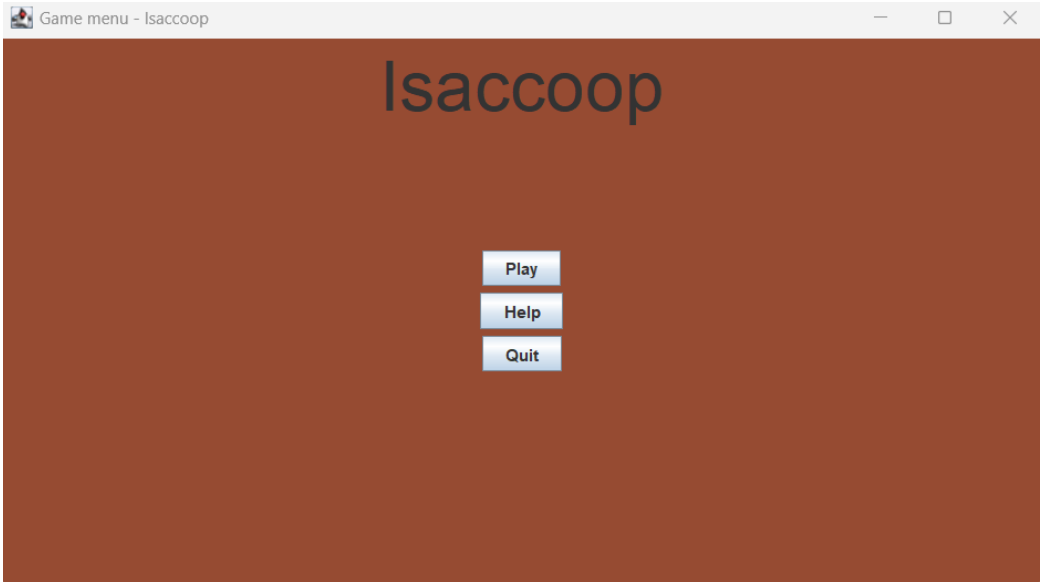

Figura A.1: Menu principale

Modalità di gioco

Comandi del giocatore

Esattamente come nel gioco "The Binding of Isaac", a cui questo applicativo si ispira liberamente, i comandi riguardanti il giocatore sono i seguenti:

- **W**: Movimento verso l'alto
- **A**: Movimento verso sinistra
- **S**: Movimento verso il basso
- **D**: movimento verso destra

Il giocatore, per sconfiggere nemici, spara le "onde di energia"

- ↑ (Freccia SU): verso l'alto
- ↓ (Freccia GIU): verso il basso
- ← (Freccia SINISTRA): verso sinistra
- → (Freccia DESTRA): verso destra

Altri comandi

- **ESC**: pausa/riprende il gioco
- **N**: il giocatore si sposta nella stanza successiva
- **P**: il giocatore si sposta nella stanza precedente

Guida utente

Dopo aver cliccato il pulsante "Help" nel menu principale, l'utente visualizzerà la seguente schermata:

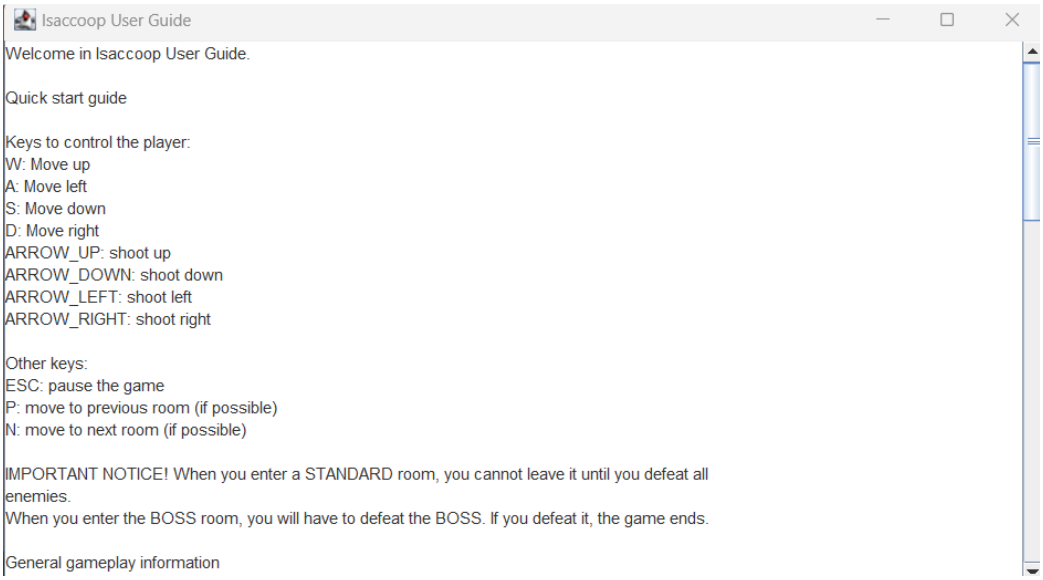

Figura A.2: Guida utente

Schermata di gioco

Dopo aver cliccato il pulsante “Play” nel menu principale, l’utente visualizzerà la seguente schermata, caratterizzata da due sezioni principali:

- **la stanza di gioco**, in alto, dove si muovono il giocatore e i nemici
- **la barra di stato**, in basso, dove vengono mostrate, da sinistra verso destra:
  - *minimappa*: mostra la struttura del livello, il numero di stanze che lo compongono e se tali stanze sono complete oppure no (una stanza è completa se non sono rimasti altri nemici da sconfiggere)
  - *statistiche del giocatore*: mostra alcune informazioni sullo stato del giocatore
  - *legenda*: spiega il significato dei colori utilizzati nella minimappa

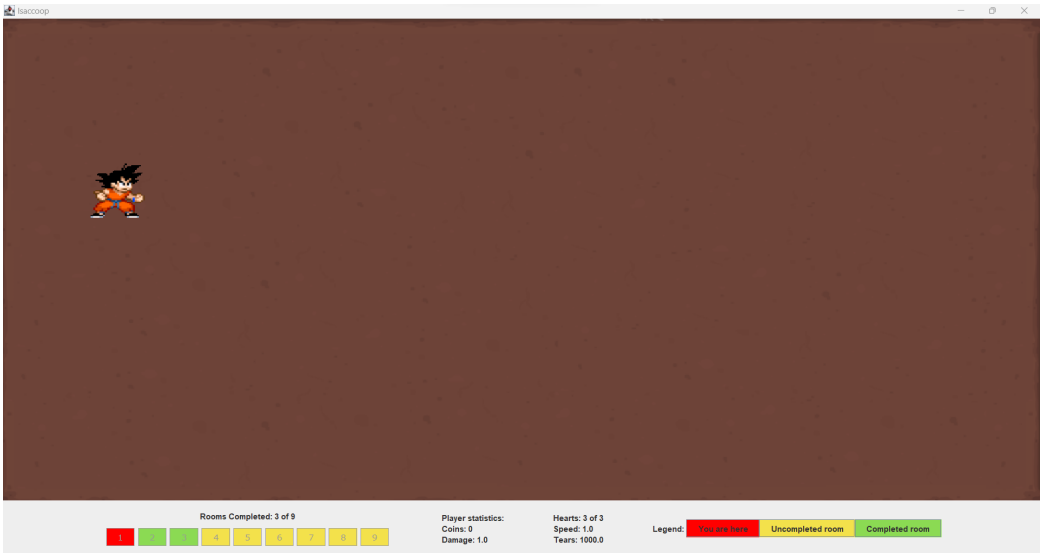

Figura A.3: La stanza iniziale

Segue screenshot di una stanza standard in cui sono presenti nemici (elefante grigio e pianta rossa e bianca) e item (i cuori).

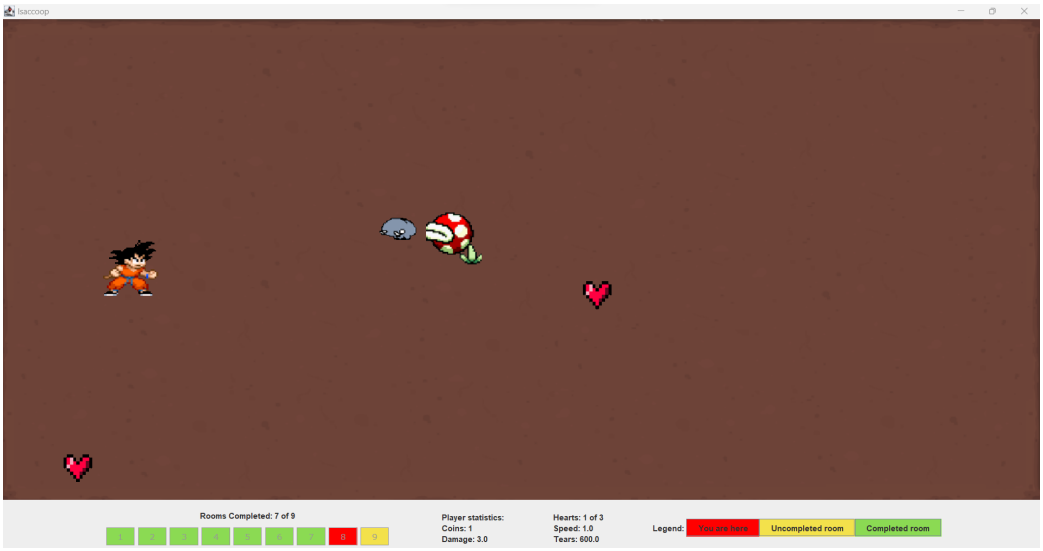

Figura A.4: Standard room con nemici e item

Segue screenshot dello shop, in cui il giocatore può comprare i powerup con le monete precedentemente raccolte.

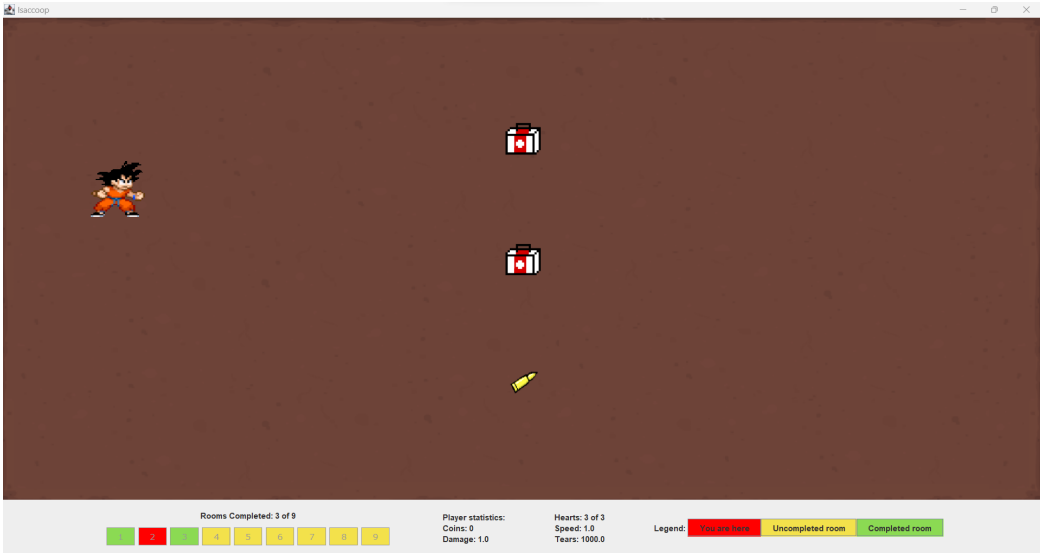

Figura A.5: Shop room con i powerup acquistabili

Se il giocatore viene colpito dai nemici e perde tutte le vite, la partita terminerà, mostrando la seguente schermata di game over.

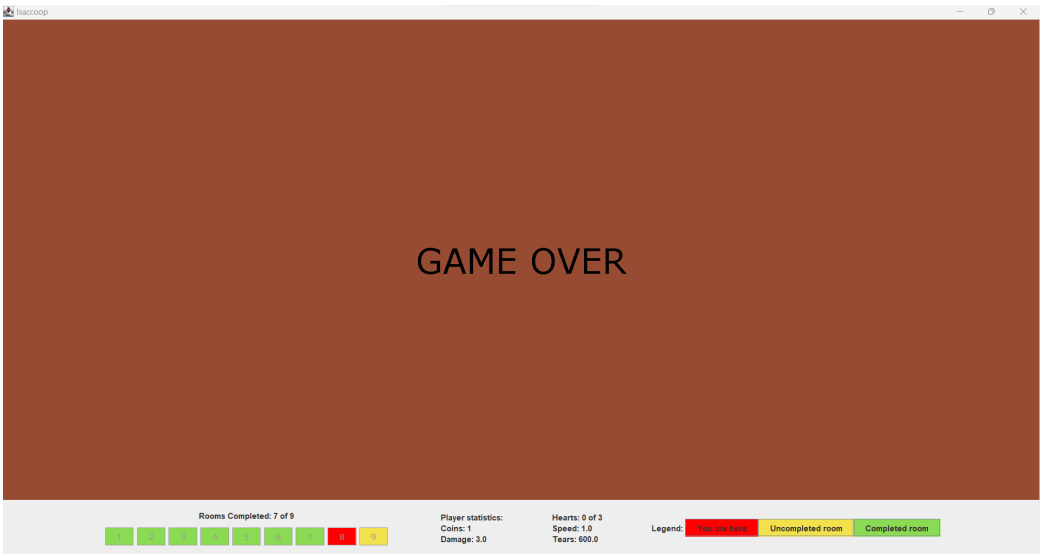

Figura A.6: Schermata di game over

Quando il giocatore completa tutte le stanze, compresa quella del boss finale, comparirà una schermata simile alla precedente, con la scritta “GAME COMPLETED”.

Sia nel caso di game over, sia nel caso di vittoria, per tornare al menu principale, è sufficiente cliccare ovunque sopra la mappa di gioco.

Appendice B

Esercitazioni di laboratorio

Marco Costantini

B.0.1 marco.costantini7@studio.unibo.it

- Laboratorio 06: `https://github.com/MarcoCostantini26/OOP-Lab06`
- Laboratorio 07: `https://github.com/MarcoCostantini26/OOP-Lab07`
- Laboratorio 08: `https://github.com/MarcoCostantini26/OOP-Lab08`
- Laboratorio 09: `https://github.com/MarcoCostantini26/OOP2021-Lab09`
- Laboratorio 10: `https://github.com/MarcoCostantini26/OOP2021-Lab10`

Daniele Pancottini

B.0.2 daniele.pancottini@studio.unibo.it

- Laboratorio 05: `https://github.com/DanielePancottini/OOP-Lab05`
- Laboratorio 06: `https://github.com/DanielePancottini/OOP-Lab06`
- Laboratorio 07: `https://github.com/DanielePancottini/OOP-Lab07`
- Laboratorio 08: `https://github.com/DanielePancottini/OOP-Lab08`
- Laboratorio 09: `https://github.com/DanielePancottini/OOP2021-Lab09`
- Laboratorio 10: `https://github.com/DanielePancottini/OOP2021-Lab10`
- Laboratorio C 1: `https://github.com/DanielePancottini/lab-csharp-simple`

- Laboratorio C 2: `https://github.com/DanielePancottini/oop-lab-csharp`

- Nella fase di analisi abbiamo preso spunto dalla repository del prof. Alessandro Ricci.
  `https://github.com/pslab-unibo/oop-game-prog-patterns-2022`