Relazione Temple-Tower per "Programmazione ad Oggetti"

Davide Vignali, Marko Cobo, Mattia Mularoni, Nicolas Montanari

15 febbraio 2025

Indice

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Il progetto Temple Tower si ispira ai classici dungeon crawler, offrendo un'esperienza di gioco a livelli in cui il giocatore esplora piani di un dungeon circolare per raccogliere tesori, sconfiggere nemici e salire verso il livello successivo. Il gioco culmina in un epico scontro con un boss finale. La meccanica dei livelli circolari prende ispirazione dal gioco Ring of Pain.

Requisiti funzionali

- L'area di gioco è circolare ed è composta da diverse caselle contententi gli elementi di gioco che possono essere positivi o negativi per il giocatore.
- Gli elementi di gioco sono rappresentati da tesori (possono contenere punti esperienza, armi), trappole (tolgono punti vita), scale (permettono il passaggio a un livello superiore).
- Il giocatore si può muovere all'interno del livello, può combattere contro i nemici, interagire con gli elementi di gioco.
- Dopo ogni scontro con un nemico o dopo aver attraversato una trappola il giocatore perde i punti vita, al termine dei quali la partita termina e si ritorna alla schermata inziale.
- Generazione casuale della torre per permettere partite sempre diverse.

Requisiti non funzionali

- Una volta arrivato all'ultimo piano il giocatore incontrerà il boss finale, il quale avrà comportamenti unici e una difficoltà maggiore rispetto ai nemici normali .
- Difficoltà variabile in base al progresso di gioco.
- Musica di sottofondo e feedback sonori associati alle varie azioni.
- Esistono più tipologie di armi la quale efficacia varia rispetto al tipo di nemico.

1.2 Modello del Dominio

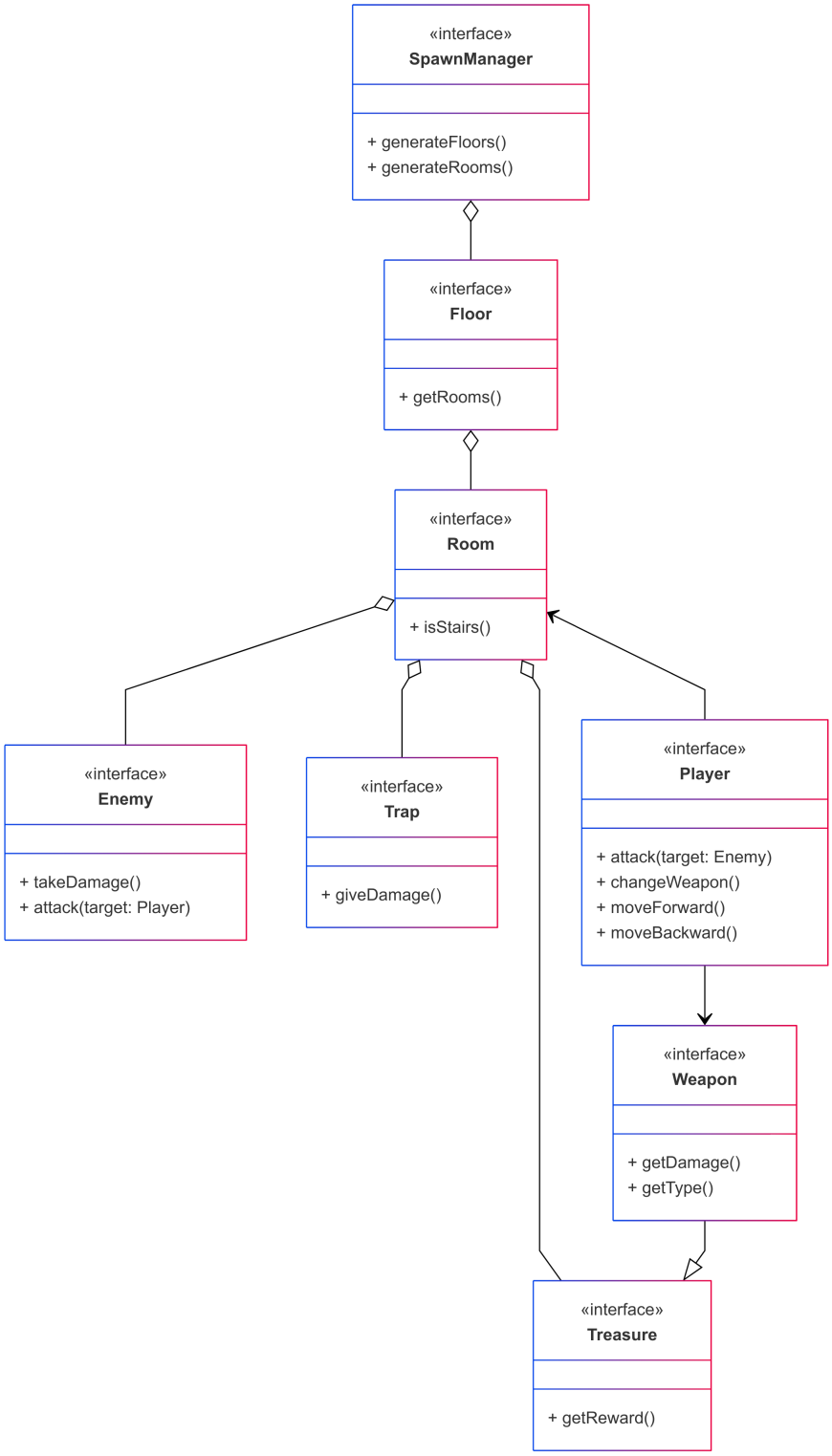
Temple Tower sarà rappresentato da una torre composta da un certo numero di piani (*Floor*), ciascuno dei quali conterrà stanze collegate tra loro. Ogni stanza potrà contenere una trappola, un nemico, un tesoro e ospitare il giocatore.

- **Trappole**: infliggono danni al giocatore.
- **Nemici**: attaccano il giocatore in combattimenti a turni.
- **Tesori**: forniscono punti vita, nuove armi, oppure potrebbe essere una trappola.

Il giocatore potrà utilizzare armi per affrontare i nemici. Poiché gli attacchi avvengono a turni, sarà fondamentale bilanciare le statistiche per evitare vittorie "matematiche" dovute a differenze di efficacia delle armi. Il giocatore e i nemici avranno a disposizione diverse tipologie di mosse di attacco e armi, che possono variare in potenza ed effetto. Dopo aver esplorato le stanze di un piano, il giocatore potrà utilizzare le scale per salire al piano successivo della torre. Durante la partita, sia il giocatore sia i nemici avranno delle barre di stato: **vita**.

- **Barra della vita**: si riduce subendo danni da nemici o trappole. Quando arriva a zero, la partita termina e si ricomincia dall'inizio.

All'ultimo piano, il giocatore affronterà il **boss finale**, un nemico più forte, dotato di mosse di attacco avanzate e particolari rispetto ai nemici ordinari. Per arricchire l'esperienza, sarà presente un sottofondo musicale durante tutto il gioco.


«interface»
SpawnManager
+ generateFloors()
+ generateRooms()
«interface»
Floor
+ getRooms()
«interface»
Room
+ isStairs()
«interface»
Enemy
+ takeDamage()
+ attack(target: Player)
«interface»
Trap
+ giveDamage()
«interface»
Player
+ attack(target: Enemy)
+ changeWeapon()
+ moveForward()
+ moveBackward()
«interface»
Weapon
+ getDamage()
+ getType()
«interface»
Treasure
+ getReward()

Capitolo 2

Design

2.1 Architettura

L'architettura del gioco **Temple Tower** segue il pattern architetturale **Model-View-Controller (MVC)** per garantire una chiara separazione delle responsabilità tra la logica di business, la presentazione e la gestione degli eventi.

Composizione del Pattern MVC

- **Model**:
  - Rappresenta la logica principale del gioco e include classi come `Tower`, `Floor`, `RoomBehavior`, `Player`, e i vari tipi di stanze (`EnemyRoom`, `TreasureRoom`, `StairsRoom`) che implementano il pattern Strategy.
  - Questo approccio consente di definire comportamenti specifici per ogni tipologia di stanza in modo modulare, rendendo semplice l'aggiunta di nuovi tipi di stanze senza modificare il codice esistente.
  - La logica di gioco, come il movimento del giocatore o gli effetti delle interazioni con nemici, trappole o tesori, è interamente contenuta nel model.
- **View**:
  - Tutta la view è gestita da uno (`SceneManager`), il quale si occupa, mediante l'implementazione di un pattern Factory, di gestire il cambiamento della vista in base alla situazione attuale.

  - Grazie al manager è possibile decentralizzare le responsabilità della vista a n classi, le quali si occuperanno della gestione degli eventi della singola interfaccia.

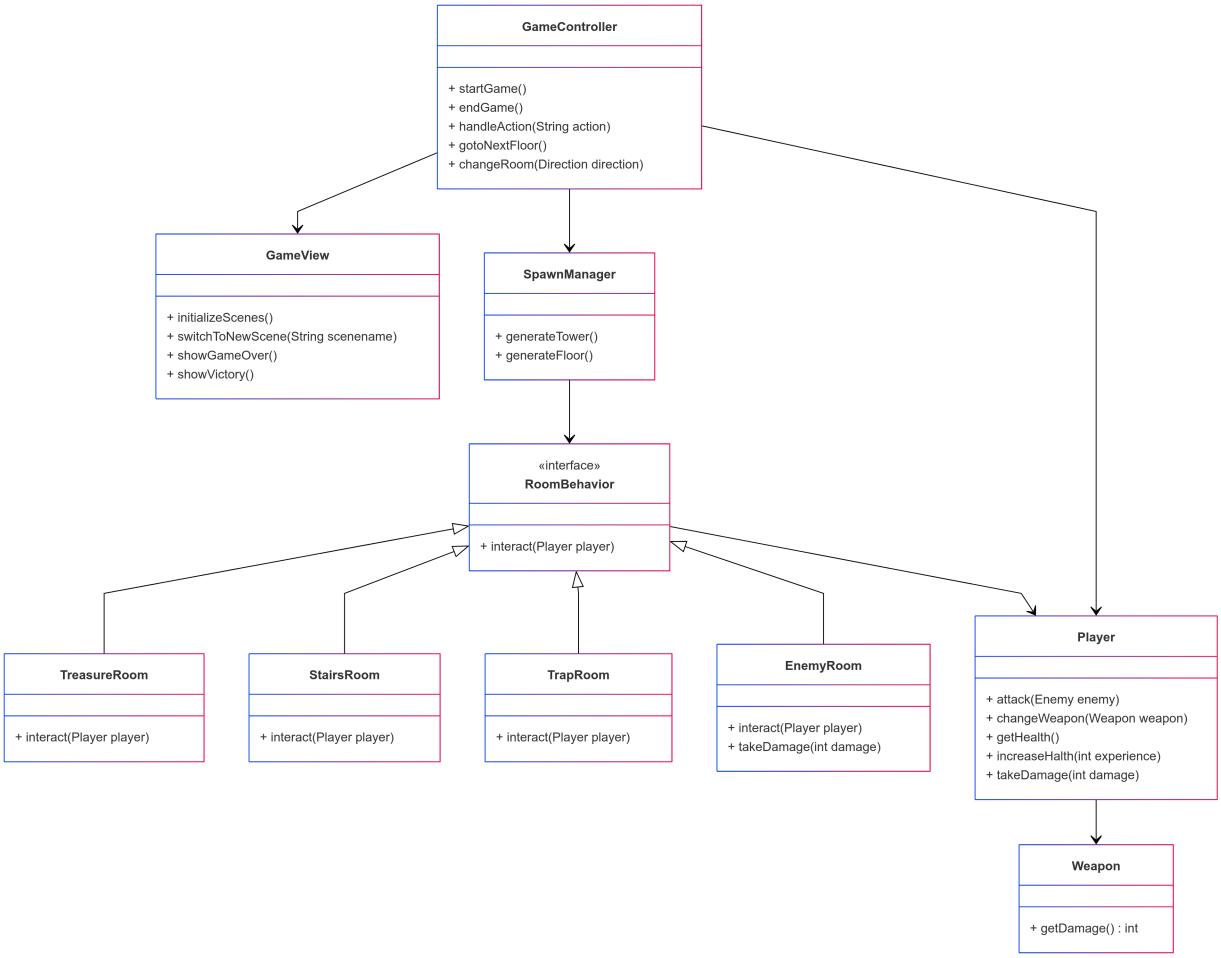
- **Controller**:
  - La classe `GameController` funge da intermediario tra il modello e la vista, orchestrando il flusso degli eventi nel gioco.
  - Gestisce le azioni dell'utente, permette al giocatore di muoversi tra le stanze (`changeRoom`) o salire al piano successivo (`gotoNextFloor`), si occupa inoltre di iniziare e terminare il gioco.

Scalabilità e Manutenibilità

Grazie all'uso combinato dei pattern MVC e Strategy:

- **Aggiunta di nuove stanze**: È possibile introdurre nuove tipologie di stanze semplicemente aggiungendo nuove implementazioni dell'interfaccia `RoomBehavior`, senza modificare altre parti del codice.
- **Separazione delle responsabilità**: La gestione della logica di gioco, della presentazione grafica e delle interazioni dell'utente è ben separata, favorendo la manutenibilità e la possibilità di cambiare singole componenti senza influenzare le altre.

Questa architettura rende il sistema flessibile, modulare e facilmente estensibile, adattandosi alle necessità di futuri miglioramenti o aggiunte.



2.2 Design dettagliato

2.2.1 Cobo

Sistema di Gestione Audio

Il sistema richiede una gestione per la riproduzione della musica di sottofondo, le cui funzionalità principali sono:

- Gestione del volume;
- Controllo dello stato di riproduzione della musica;
- Corretto caricamento e gestione delle risorse audio;
- Mantenimento di una corretta separazione tra logica, controllo e view.

Architettura MVC

La soluzione proposta implementa un pattern Model-View-Controller (MVC) con le seguenti componenti:

- **Model (MusicModel)**: Mantiene informazioni sul volume e sullo stato di riproduzione, gestisce il ciclo di vita del clip audio e lo stato interno del sistema audio.
- **Controller (MusicController)**: Gestisce le operazioni sulla riproduzione musicale e fa da intermediario tra View e Model.
- **View**: Richiama i metodi del controller per gestire la riproduzione.

Vantaggi e svantaggi

Pro: Separazione delle responsabilità, il MusicModel gestisce i dettagli tecnici mentre il MusicController gestisce il flusso principale.

Contro: Richiede una buona coordinazione tra le classi.

Diagramma UML

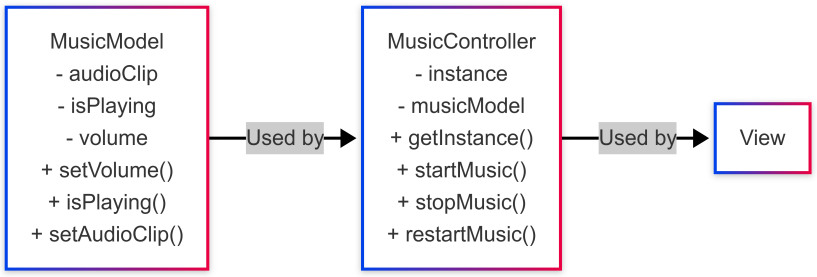


Figura 2.1: Diagramma UML del sistema di gestione audio

Pattern Singleton

Il pattern Singleton è stato utilizzato nella classe `MusicController` per garantire che esista solo un'istanza del controller musicale durante l'intera esecuzione dell'applicazione. Questo evita conflitti e garantisce un punto di accesso globale alla gestione della musica.

Metodo Template: getInstance()

Motivazione: Assicura che esista solo un'istanza di `MusicController`, facilitando la gestione centralizzata della riproduzione musicale. Il pattern Singleton è stato utilizzato in modo appropriato per il `MusicController`, poiché

la gestione centralizzata della riproduzione musicale richiede una singola istanza.

2.2.2 Montanari

Gestione dei Popup tramite Factory Method

Problema Nel gioco, esistono diversi tipi di finestre di dialogo (popup) che vengono mostrate in varie situazioni, come la raccolta di un'arma o il guadagno di esperienza. La creazione manuale di questi popup in ogni punto del codice porta a una duplicazione del codice e a una ridotta manutenibilit'a. Inoltre, senza un'astrazione adeguata, ogni nuova finestra di dialogo richiederebbe la scrittura ripetitiva della sua struttura, aumentando il rischio di incoerenze nel comportamento.

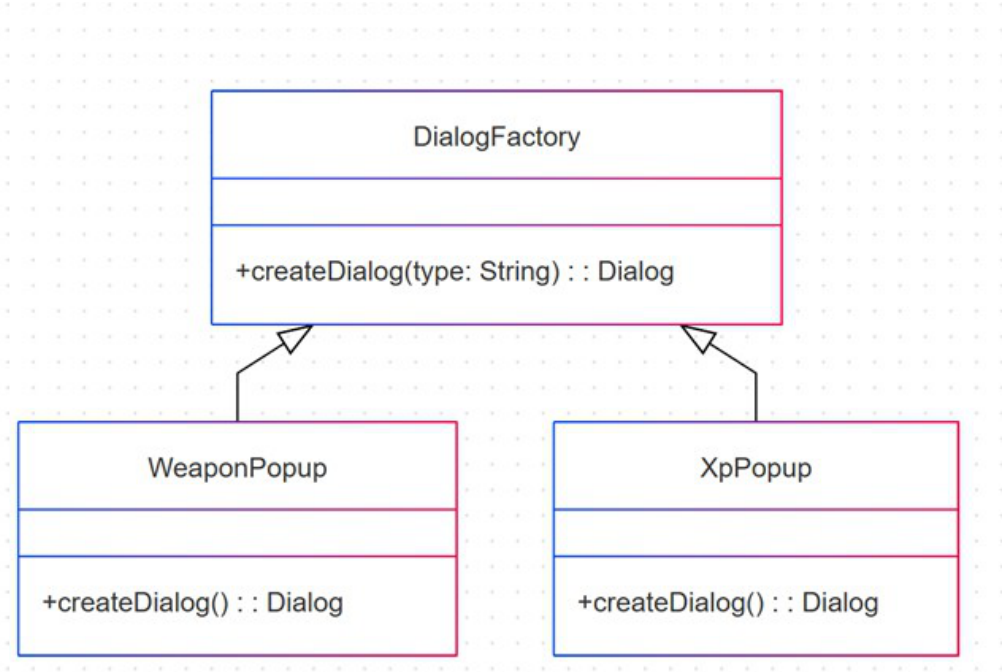
Soluzione Per risolvere questo problema, 'e stato adottato il Factory Method, che consente di delegare la creazione dei popup a una classe centralizzata (`DialogFactory`). Questa classe definisce un metodo `createDialog` che, in base al tipo di popup richiesto, restituisce un oggetto `Dialog` con il contenuto e il comportamento appropriati.

Questa soluzione offre diversi vantaggi:

- **Miglior riuso del codice**: la logica di costruzione dei popup 'e riutilizzabile e centralizzata.
- **Maggiore manutenibilit'a**: per aggiungere nuovi tipi di popup, basta estendere il metodo `createDialog` senza modificare il codice esistente.
- **Maggiore coerenza**: tutti i popup rispettano uno stile uniforme e una gestione degli eventi standardizzata.

Un'alternativa considerata era l'uso di una classe `DialogUtil` con metodi statici per ogni tipo di popup. Tuttavia, questo approccio avrebbe reso pi'u difficile estendere il sistema senza modificare direttamente la classe di utilit'a, violando il principio Open/Closed.

Schema UML Lo schema seguente mostra l'implementazione del Factory Method per la creazione dei popup:



Applicazione del Pattern Factory Method

- `DialogFactory` 'e la classe che definisce il metodo `createDialog`, delegando la creazione alle classi specifiche.
- `WeaponPopup` e `XpPopup` sono le sottoclassi che sovrascrivono il metodo factory per fornire l'implementazione specifica del popup.
- Il codice client utilizza `DialogFactory.createDialog()` per ottenere il popup corretto senza conoscere i dettagli della sua implementazione.

Questa implementazione segue il pattern Factory Method, migliorando la separazione delle responsabilit'a e facilitando l'aggiunta di nuovi tipi di popup in futuro.

Gestione della Sincronizzazione tra Attacco, Punti Vita e UI

Problema Nel contesto della gestione della scena di combattimento, è emersa la necessità di garantire una sincronizzazione corretta tra l'attacco del giocatore, la riduzione dei punti vita dell'avversario e l'aggiornamento dell'interfaccia utente (UI). Un'errata gestione di questo processo potrebbe causare incongruenze visive o problemi di gameplay, come:

- Attacchi non registrati;

- Aggiornamenti tardivi della UI;
- Reazioni non coerenti tra azione e conseguenza.

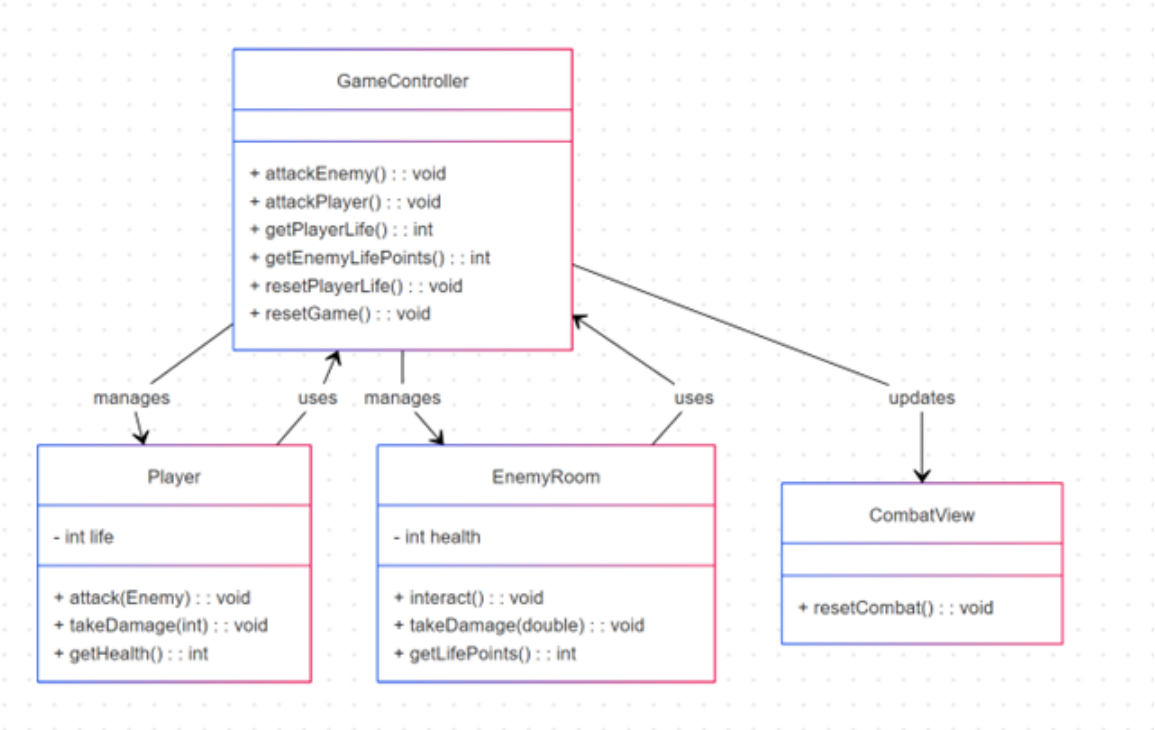
Soluzione Per affrontare questa problematica, è stata implementata una gestione sequenziale delle azioni attraverso l'uso combinato di:

- **Animazioni** (`Timeline`) per gestire i movimenti visivi;
- **Ritardi programmati** (`PauseTransition`) per sincronizzare le azioni;
- **Aggiornamenti sincroni della UI** (`Platform.runLater`) per garantire modifiche sicure nel thread principale.

Il flusso di esecuzione prevede i seguenti passaggi:

1. L'attacco viene eseguito, attivando un'animazione di avanzamento del giocatore o un effetto visivo (ad esempio, una fiamma).
2. Una volta terminata l'animazione, il danno viene applicato riducendo i punti vita dell'avversario.
3. La UI viene aggiornata, modificando la barra della salute (`ProgressBar`) e il valore numerico degli HP dell'avversario.
4. Se il nemico è ancora in vita, viene avviato un ritardo tramite `PauseTransition`, che simula il tempo di reazione prima della sua risposta.
5. Il nemico esegue il contrattacco, aggiornando la UI con la nuova quantità di punti vita del giocatore.
6. Si verifica se il combattimento è terminato:
    - In caso di vittoria del giocatore, il pulsante di attacco viene disabilitato e la salute viene ripristinata.
    - In caso di sconfitta, viene mostrato un popup che reindirizza il giocatore al menu principale.

Schema UML Lo schema seguente mostra l'implementazione di una parte delle classi utilizzate



Benefici ottenuti Grazie a questa soluzione, si ottiene una transizione fluida tra attacco, difesa e aggiornamento dell'UI, evitando problemi di desincronizzazione. L'uso di `Platform.runLater` assicura che ogni aggiornamento dell'interfaccia venga eseguito nel thread principale di JavaFX, prevenendo errori di concorrenza e migliorando la responsività dell'applicazione.

Inoltre, l'introduzione di pause controllate tra le fasi di combattimento permette di migliorare il feedback visivo per il giocatore, rendendo l'esperienza più chiara e coinvolgente.

2.2.3 Vignali

Modular Game Data Loading and Tower Configuration

Problema Il sistema necessita di caricare e gestire dati di gioco da file JSON esterni in modo flessibile e modulare, permettendo l'uso sia di torri predefinite che di mod create dagli utenti. I dati devono essere validati e supportare riferimenti relativi tra file.

Soluzione Ho valutato due approcci principali:

1. Un sistema di caricamento dati decentralizzato dove ogni componente carica i propri dati

2. Un gestore centralizzato che coordina tutto il caricamento

Ho scelto la seconda opzione implementando GameDataManagerImpl come Singleton per garantire un punto di accesso unico ai dati di gioco. Ho considerato l'uso di Dependency Injection, ma data la natura globale dei dati di gioco e la necessità di garantire uno stato coerente, Singleton è risultato più appropriato.

Il sistema utilizza Record immutabili (FloorData, Enemy, Weapon) e custom deserializer GSON per garantire type-safety e validazione durante il caricamento. Questa soluzione permette una chiara separazione tra dati e logica, facilitando l'estensione del sistema con nuove mod.

Schema UML:

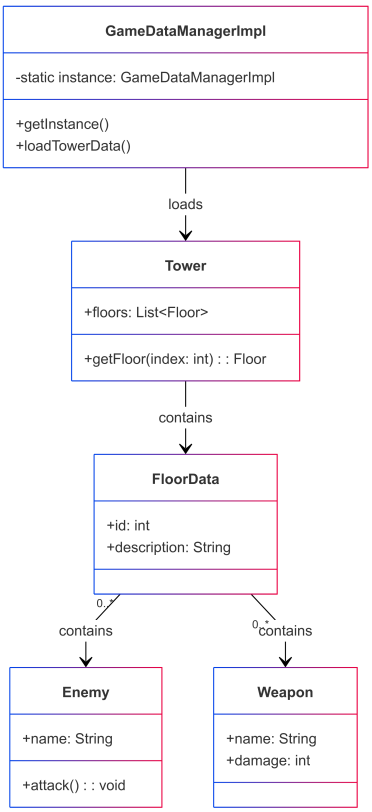


Figura 2.2: GameDataManagerImpl as Singleton and its relations.

Sistema di Modding

Problema Il gioco deve permettere agli utenti di creare e importare torri personalizzate (mod) sia da cartelle che da file ZIP, gestendo validazione, conflitti di nomi e isolamento tra mod.

Soluzione Ho considerato due possibili architetture:

1. Un sistema event-based con chiamate asincrone per l'importazione
2. Un'architettura MVC con Observer pattern per la sincronizzazione UI

Ho scelto la seconda opzione perché offre una separazione più chiara delle responsabilità e una gestione più prevedibile dello stato. Il pattern Observer è stato preferito a un sistema di callback perché permette di aggiungere facilmente nuovi observer senza modificare il codice esistente.

Ho implementato anche il pattern Strategy per l'importazione, permettendo di aggiungere facilmente nuovi formati di mod oltre a ZIP e cartelle.

Schema UML:

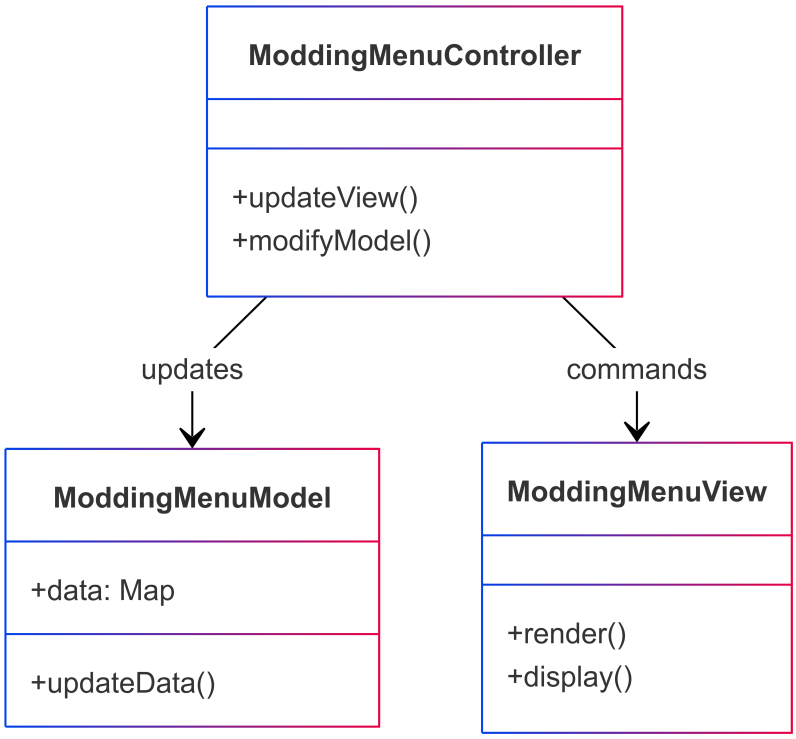


Figura 2.3: Modding Menu UML: MVC and Observer pattern overview.

Gestione dello Spawn System

Problema Il sistema deve generare contenuti procedurali bilanciati rispettando vincoli di livello e configurazioni di spawn.

Soluzione Ho valutato due approcci:

1. Generazione puramente casuale con post-validazione
2. Sistema template-based con strategie di Generazione

Ho scelto il secondo approccio implementando un Template Method in SpawnManagerImpl che standardizza il processo di generazione permettendo variazioni nel comportamento specifico. Questo pattern è stato preferito a una soluzione più flessibile ma potenzialmente caotica basata su eventi. Il metodo template definisce tre fasi:

1. Selezione del tipo di piano (template method).
2. Generazione delle stanze (hook method).
3. Popolamento delle stanze (hook method).

2.2.4 Mularoni

Cambio vista

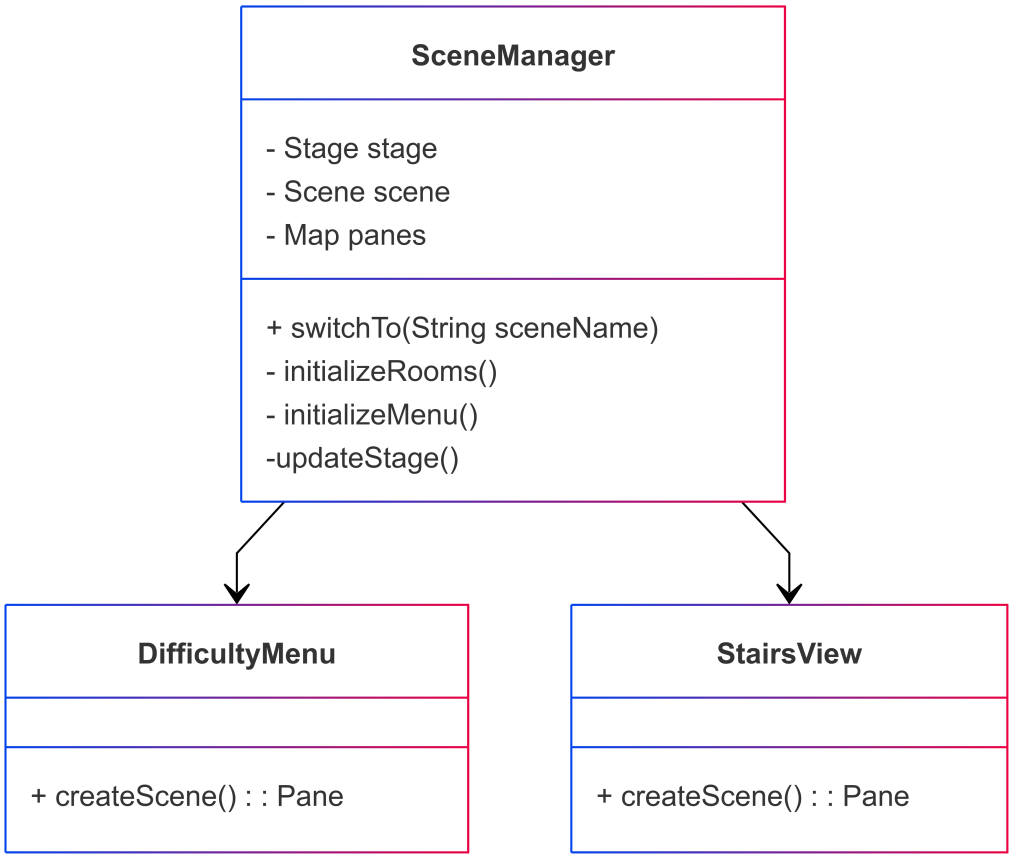


Figura 2.4: Rappresentazione UML del pattern Factory per la gestione delle viste (esempio con 2 viste)

Problema Il gioco ha diverse viste, una per ogni situazione (Menu iniziale, vista sulle stanze, stanze singole ecc...), è necessario gestire la visualizzazione in un modo coerente.

Soluzione Il sistema per la gestione delle viste utilizza il *pattern Factory*, come da Figura 2.4: le classi che implementano le viste `NomevistaView` una volta create, vengono inserite nel mapping interno di `SceneManager` il quale si occupa di:

- All'avvio del gioco: inizializzare le viste relative alle schermate inziali.

- Dopo il caricamento della torre: inizializzare le viste degli elementi di gioco.
- Quando si chiama `switchTo`: caricare il nuovo Pane nella scena e visualizzarlo.

Questa soluzione permette in ogni parte della view, richiamando `switchTo` di passare alla view successiva passando il nome di essa, permettendo una certa flessibilità nella creazione e linking di eventuali nuove viste.

Stanze del gioco

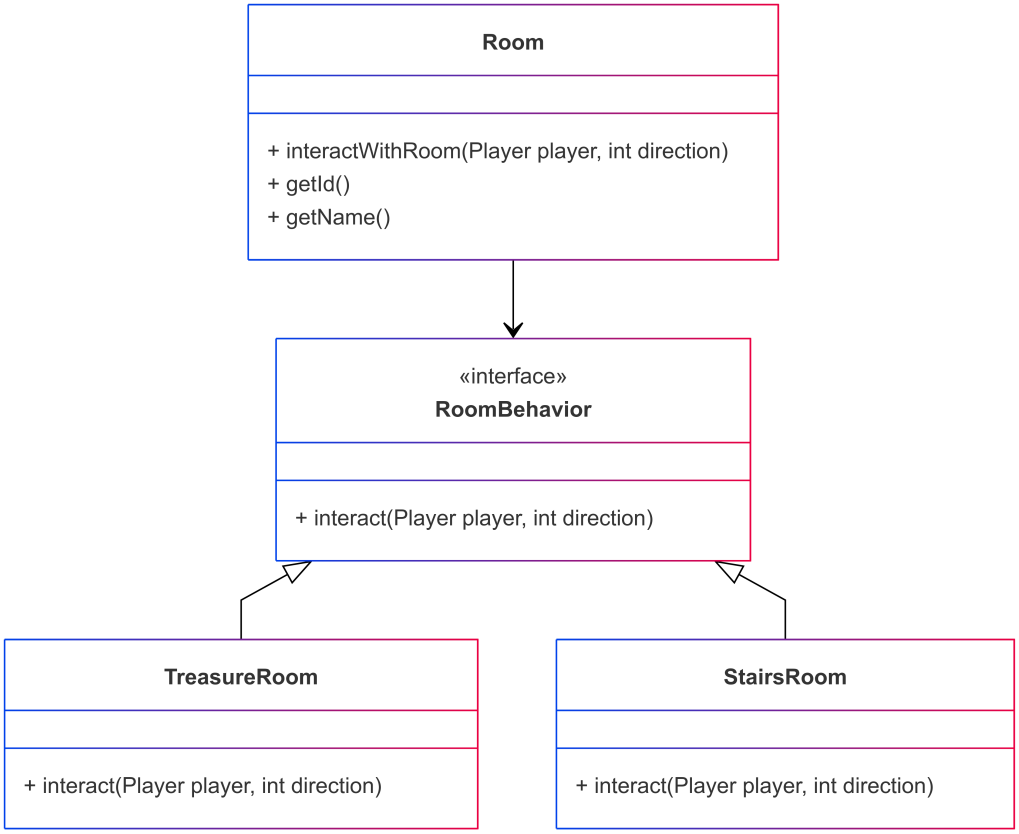


Figura 2.5: Rappresentazione UML del pattern Strategy per le stanze del gioco

Problema Temple Tower ha più stanze, ognuna con un contenuto differente.

Soluzione Il sistema per la modellazione delle stanze utilizza il *pattern Strategy*, come da Figura 2.5: le implementazioni di `RoomBehavior` possono essere modificate, e a seconda di cosa si inserisce nel metodo `interact()`, il giocatore subirà o lancierà (a seconda di `direction`) delle azione da/alla stanza. Per esempio un metodo interact dentro Trap toglierà punti vita al Player, mentre la stanza Stairs cambierà il piano di quest'ultimo. Questo pattern permette di ampliare i tipi possibili di stanze richiedendo poche modifiche all'interno del codice già esistente.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per il testing generale di Temple Tower è stato utilizzata la suite JUnit per il testing automatizzato.

Controller

Sono state testate nel controller, tutte le classi che interagiscono tramite esso, nella view.

- Giocatore: vita, cambio stanza e armi.
- Nemico: vita e attacchi.
- Cambio piano.
- Boss finale: sistema di decisione sull'arrivo alla stanza del boss finale.

3.2 Note di sviluppo

Questa sezione, come quella riguardante il design dettagliato va svolta **singolarmente da ogni membro del gruppo**. Nella prima parte, ciascuno dovrà mostrare degli esempi di codice particolarmente ben realizzati, che dimostrino proefficienza con funzionalità avanzate del linguaggio e capacità di spingersi oltre le librerie mostrate a lezione.

- **Elencare** (fare un semplice elenco per punti, non un testo!) le feature *avanzate* del linguaggio e dell'ecosistema Java che sono state utilizzate. Le feature di interesse sono:

  - Progettazione con generici, ad esempio costruzione di nuovi tipi generici, e uso di generici bounded. L'uso di classi generiche di libreria non è considerato avanzato.
  - Uso di lambda expressions
  - Uso di `Stream`, di `Optional` o di altri costrutti funzionali
  - Uso di reflection
  - Definizione ed uso di nuove annotazioni
  - Uso del Java Platform Module System
  - Uso di parti della libreria JDK non spiegate a lezione (networking, compressione, parsing XML, eccetera...)
  - Uso di librerie di terze parti (incluso JavaFX): Google Guava, Apache Commons...

- Si faccia molta attenzione a non scrivere banalità, elencando qui features di tipo "core", come le eccezioni, le enumerazioni, o le inner class: nessuna di queste è considerata avanzata.

- Per ogni feature avanzata, mostrata, includere:

  - Nome della feature
  - Permalink GitHub al punto nel codice in cui è stata utilizzata

In questa sezione, *dopo l'elenco*, vanno menzionati ed attributi con precisione eventuali pezzi di codice "riadattati" (o scopiazzati...) da Internet o da altri progetti, pratica che tolleriamo ma che non raccomandiamo. Si rammenta agli studenti che non è consentito partire da progetti esistenti e procedere per modifiche successive. Si ricorda anche che i docenti hanno in mano strumenti antiplagio piuttosto raffinati e che "capiscono" il codice e la storia delle modifiche del progetto, per cui tecniche banali come cambiare nomi (di classi, metodi, campi, parametri, o variabili locali), aggiungere o togliere commenti, oppure riordinare i membri di una classe vengono individuate senza problemi. Le regole del progetto spiegano in dettaglio l'approccio dei docenti verso atti gravi come il plagiarismo.

I pattern di design **non** vanno messi qui. L'uso di pattern di design (come suggerisce il nome) è un aspetto avanzato di design, non di implementazione, e non va in questa sezione.

Elementi positivi

- Si elencano gli aspetti avanzati di linguaggio che sono stati impiegati
- Si elencano le librerie che sono state utilizzate
- Per ciascun elemento, si fornisce un permalink
- Ogni permalink fa riferimento ad uno snippet di codice scritto dall'autore della sezione (i docenti verificheranno usando `git blame`)
- Se si è utilizzato un particolare algoritmo, se ne cita la fonte originale. Ad esempio, se si è usato Mersenne Twister per la generazione di numeri pseudo-random, si cita [**?**].
- Si identificano parti di codice prese da altri progetti, dal web, o comunque scritte in forma originale da altre persone. In tal senso, si ricorda che agli ingegneri non è richiesto di re-inventare la ruota continuamente: se si cita debitamente la sorgente è tollerato fare uso di di snippet di codice open source per risolvere velocemente problemi non banali. Nel caso in cui si usino snippet di codice di qualità discutibile, oltre a menzionarne l'autore originale si invitano gli studenti ad adeguare tali parti di codice agli standard e allo stile del progetto. Contestualmente, si fa presente che è largamente meglio fare uso di una libreria che copiarsi pezzi di codice: qualora vi sia scelta (e tipicamente c'è), si preferisca la prima via.

Elementi negativi

- Si elencano feature core del linguaggio invece di quelle segnalate. Esempi di feature core da non menzionare sono:
  - eccezioni;
  - classi innestate;
  - enumerazioni;
  - interfacce.
- Si elencano applicazioni di terze parti (peggio se per usarle occorre licenza, e lo studente ne è sprovvisto) che non c'entrano nulla con lo sviluppo, ad esempio:
  - Editor di grafica vettoriale come Inkscape o Adobe Illustrator;
  - Editor di grafica scalare come GIMP o Adobe Photoshop;

  - Editor di audio come Audacity;
  - Strumenti di design dell'interfaccia grafica come SceneBuilder: il codice è in ogni caso inteso come sviluppato da voi.

- Si descrivono aspetti di scarsa rilevanza, o si scende in dettagli inutili.
- Sono presenti parti di codice sviluppate originalmente da altri che non vengono debitamente segnalate. In tal senso, si ricorda agli studenti che i docenti hanno accesso a tutti i progetti degli anni passati, a Stack Overflow, ai principali blog di sviluppatori ed esperti Java, ai blog dedicati allo sviluppo di soluzioni e applicazioni (inclusi blog dedicati ad Android e allo sviluppo di videogame), nonché ai vari GitHub, GitLab, e Bitbucket. Conseguentemente, è *molto* conveniente *citare* una fonte ed usarla invece di tentare di spacciare per proprio il lavoro di altri.
- Si elencano design pattern

3.2.1 Vignali

Utilizzo della libreria gson

Utilizzato principalmente in GameDataManager per l'importazione e il caricamento delle torri: permalink/

Utilizzo della libreria JavaFX

Utilizato All'interno di ModdingMenuView insieme a codice CSS: permalink/

Utilizzo della libreria SL4J

Utilizzato per la gestione dei log principalemente nei test ma anche in giro per il codice: permalink/

Utilizzo della libreria Apache Commons IO

Utilizzato per la gestione dei file insime alla java util per l'importazione dei file: permalink/

Utilizzo della libreria Java util zip

Utilizzato per la decompressione dei file: permalink

Utilizzo di stream e lambda expressions

Utilizzato in diverse parti qui un esempio: permalink

Utilizzo di Optional

Utilizzato in diverse parti qui un esempio: permalink

3.2.2 Mularoni

Utilizzo di `Stream` e lambda expressions

- Esempio: permalink/

Utilizzo di `Threading e Task` in Javafx per prevenire race condition

- Esempio: permalink/

Utilizzo della libreria SLF4J

- Esempio: permalink/

Cerchi con javafx

- Fonte: Stackoverflow/

Immagini con javafx

- Fonte: Stackoverflow/

Uso di Platform.runLater()

- Fonte: Stackoverflow/

3.2.3 Montanari

Uso di SLF4J per il logging

Il progetto utilizza il framework SLF4J per la gestione dei log. Questo consente un monitoraggio efficace degli eventi di gioco.
Esempio di codice:

```
private static final Logger LOGGER = LoggerFactory.getLogger(GameControllerImpl.
```

Link al codice su GitHub

Uso di Lambda Expressions

Per la gestione degli eventi, il codice utilizza espressioni lambda che migliorano la leggibilità e riducono la verbosità.
Esempio:

```
attackBt.setOnAction(e -> this.performAttack());
```

Link al codice su GitHub

Gestione del multithreading con Platform.runLater

JavaFX richiede che gli aggiornamenti dell'interfaccia avvengano nel thread principale. Platform.runLater viene utilizzato per garantire che le modifiche alla UI siano eseguite in modo sicuro.
Esempio:

```
Platform.runLater(() -> playerHpBar.setProgress(newHealth / (double) maxHealth))
```

Link al codice su GitHub

Utilizzo di Timeline e KeyFrame per le animazioni

Il codice implementa animazioni utilizzando Timeline e KeyFrame, creando effetti visivi per gli attacchi e la riduzione della vita dei personaggi.
Esempio:

```
Timeline attackAnimation = new Timeline(
new KeyFrame(Duration.seconds(0.5), e -> enemyHpBar.setProgress(newEnemyHp / (do
);
attackAnimation.play();
```

Link al codice su GitHub

Utilizzo di JavaFX Scene Graph

La costruzione dell'interfaccia utente avviene attraverso l'uso di StackPane, HBox, VBox e BorderPane.
Link al codice su GitHub

3.2.4 Cobo

3.2.5 Esempio

Utilizzo della libreria SLF4J

Utilizzata in vari punti. Un esempio è `https://github.com/AlchemistSimulator/Alchemist/blob/5c17f8b76920c78d955d478864ac1f11508ed9ad/alchemist-swingui/src/main/java/it/unibo/alchemist/boundary/swingui/effect/impl/EffectBuilder.java#L49`

Utilizzo di `LoadingCache` dalla libreria Google Guava

Permalink: `https://github.com/AlchemistSimulator/Alchemist/blob/d8a1799027d7d685569e15316a32e6394632ce71/alchemist-incarnation-protelis/src/main/java/it/unibo/alchemist/protelis/AlchemistExecutionContext.java#L141-L143`

Utilizzo di `Stream` e lambda expressions

Usate pervasivamente. Il seguente è un singolo esempio. Permalink: `https://github.com/AlchemistSimulator/Alchemist/blob/d8a1799027d7d685569e15316a32e6394`
`alchemist-incarnation-protelis/src/main/java/it/unibo/alchemist/model/ProtelisIncarnation.java#L98-L120`

Scrittura di metodo generico con parametri contravarianti

Permalink: `https://github.com/AlchemistSimulator/Alchemist/blob/d8a1799027d7d685569e15316a32e6394632ce71/alchemist-incarnation-protelis/src/main/java/it/unibo/alchemist/protelis/AlchemistExecutionContext.java#L141-L143`

Protezione da corse critiche usando `Semaphore`

Permalink: `https://github.com/AlchemistSimulator/Alchemist/blob/d8a1799027d7d685569e15316a32e6394632ce71/alchemist-incarnation-protelis/src/main/java/it/unibo/alchemist/model/ProtelisIncarnation.java#L388-L440`

Capitolo 4

Commenti finali

4.1 Vignali

Penso che la parte scritta da me sia stata fatta al meglio per il tempo a disposizione, e il fatto di aver passato molto tempo iniziale a pensare su come far funzionare la questione dei Json è servita, alcuni dati che inizialmente avevo pensato di inserire nella mia parte che era principalmente la struttura dati li ho dovuti rimuovere per non appesantire troppo il lavoro ai miei compagni, mentre alcuni non sono stati utilizzati anche se inseriti, tutto sommato sono soddisfatto della mia parte e del mio codice, la feature più grande che ho dovuto rimuovere per mancanza di tempo è la creazione tramite interfaccia di torri personalizzate, probabilmente sarebbero servite 15-20 ore extra per implemtarla e sarebbe stato molto interessante svilupparla quindi potrebbe essere una dei lavori futuri

4.2 Mularoni

Considerando il tempo a disposizione e gli impegni derivanti da lavoro e altri esami, reputo questo progetto "accettabile" ma con un grande potenziale inespresso, dovuto sia alla scarsa divisibilità del progetto a livello di idea (è difficile che tutti lavorino a compartimenti stagni) che di organizzazione generale.

4.3 Montanari

Durante lo sviluppo del progetto, mi sono occupato principalmente del sistema di combattimento, della creazione delle varie view di gioco e del controller

in modo da far comunicare tutte le classi tra di loro. Ritengo che i miei punti di forza siano stati la capacità di problem-solving, la scrittura di codice efficiente e la collaborazione con il team. Tuttavia, ho riscontrato alcune difficoltà, in particolare nella gestione del tempo e nel debugging di problemi complessi.

Ruolo all'interno del gruppo

All'interno del team, il mio ruolo è stato quello di sviluppatore principale per il sistema di combattimento e le interfacce di gioco, oltre a occuparmi della logica di comunicazione tra le classi. Ho contribuito a garantire che tutte le componenti funzionassero correttamente insieme, cercando di mantenere un buon livello di collaborazione con gli altri membri.

Lavori futuri

Se il progetto dovesse essere portato avanti, credo che si potrebbe migliorare in diverse direzioni. In particolare, suggerirei di ottimizzare le prestazioni del sistema di combattimento, migliorare l'interfaccia utente e aggiungere nuove funzionalità per rendere il gioco più coinvolgente. Inoltre, potrebbe essere utile impiegare il progetto come base per un gioco più ampio o come portfolio personale per dimostrare le competenze acquisite.

Nel complesso, questa esperienza mi ha permesso di migliorare le mie competenze in programmazione, gestione delle comunicazioni tra classi e sviluppo di interfacce di gioco, e sono soddisfatto dei progressi fatti.

4.4 Cobo

Mi sono occupato della gestione dell'audio e dell'integrazione del sistema musicale nel progetto. Ho implementato il controllo della musica di sottofondo, il sistema di regolazione del volume e la gestione della riproduzione attraverso il pattern Singleton. Inoltre mi sono occupato della gestione delle scene e dell'interfaccia utente, implementando i menu, ad esempio quello delle impostazioni che ha pulsanti per il controllo del volume e per la navigazione tra schermate.

Punti di forza

Utilizzo avanzato di Java Sound API per il controllo dell'audio. Implementazione del pattern Singleton con doppia verifica per garantire efficienza e

sicurezza. Utilizzo di JavaFX per un'interfaccia grafica responsive e dinamica. Creazione di pulsanti grafici con immagini per un'interfaccia più intuitiva. Organizzazione chiara del codice, rendendo i menu facilmente estendibili.

Punti di debolezza

La gestione del volume potrebbe essere migliorata con un'interfaccia utente più intuitiva. Alcune transizioni tra scene potrebbero essere più fluide e animate. L'aspetto visivo del menu potrebbe essere più curato con CSS.

Lavori futuri

Se il progetto venisse portato avanti, si potrebbe migliorare il sistema audio offrendo un maggiore controllo e una qualità audio migliore. Inoltre si potrebbe aggiungere un file di configurazione per memorizzare le preferenze dell'utente e un design UI più accattivante con animazioni e transizioni fluide. Durante il corso mi sono trovato molto bene con i docenti, sia per la chiarezza delle spiegazioni sia per l'efficacia dei metodi di insegnamento adottati. Il materiale fornito e l'approccio didattico hanno reso l'apprendimento strutturato e accessibile, facilitando la comprensione anche di concetti avanzati.

Appendice A

Guida utente

Abbiamo cercato di rendere l'applicazione il più facile possibile da utilizzare. Una volta avviata all'utente basterà cliccare il bottone su schermo per arrivare al menu principale, a questo punto serve importare una qualunque torre tramite il Modding Menu, e selezionarla per poter giocare, la tower si troverà nella repository oppure può essere scaricata da questo link (Tower Download). La tower può essere selezionata in una qualunque posizione all'interno del proprio pc, una volta selezionata si può iniziare a giocare, cliccando il bottone centrale, si consiglia di utilizzare la difficoltà facile per motivi di testing, a questo punto per giocare si può girare a destra oppure sinistra con i due bottoni in basso e cliccando il bottone centrale si può entrare in una delle stanze, una volta entrato basterà seguire le istruzioni su schermo;

Appendice B

Esercitazioni di laboratorio

B.0.1 mattia.mularoni@studio.unibo.it

- Laboratorio 07: `https://virtuale.unibo.it/mod/forum/discuss.php?d=177162#p246190`
- Laboratorio 08: `https://virtuale.unibo.it/mod/forum/discuss.php?d=178723#p247234`
- Laboratorio 09: `https://virtuale.unibo.it/mod/forum/discuss.php?d=179154#p247924`
- Laboratorio 10: `https://virtuale.unibo.it/mod/forum/discuss.php?d=180101#p249553`
- Laboratorio 04: `https://virtuale.unibo.it/mod/forum/discuss.php?d=12345#p123456`
- Laboratorio 06: `https://virtuale.unibo.it/mod/forum/discuss.php?d=22222#p222222`
- Laboratorio 09: `https://virtuale.unibo.it/mod/forum/discuss.php?d=99999#p999999`

B.0.2 davide.vignali4@studio.unibo.it

- Laboratorio 09: `https://virtuale.unibo.it/mod/forum/discuss.php?d=179154#p248027`

B.0.3 nicolas.montanari3@studio.unibo.it

- Laboratorio 09: `https://virtuale.unibo.it/mod/forum/discuss.php?d=179154`
- Laboratorio 10: `https://virtuale.unibo.it/mod/forum/discuss.php?d=180101`