# 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 i piani di un dungeon circolare alla ricerca di tesori, affronta nemici e procede al piano successivo, culminando in un epico scontro con un boss finale. La struttura dei livelli, con meccaniche ispirate al gioco Ring of Pain, garantisce partite sempre diverse.

### Requisiti funzionali

- L'area di gioco è circolare ed è composta da diverse caselle contenenti elementi che possono avere effetti positivi o negativi sul giocatore.
- Gli elementi di gioco includono tesori (che possono fornire punti esperienza o armi), trappole (che riducono i punti vita) e scale (che permettono al giocatore di salire al piano successivo).
- Il giocatore può muoversi liberamente all'interno del livello, combattere contro i nemici e interagire con gli elementi del gioco.
- Ogni scontro o attraversamento di una trappola determina la perdita di punti vita; al raggiungimento dello zero, la partita termina e si torna alla schermata iniziale.
- La generazione casuale della torre garantisce variabilità e nuove sfide ad ogni partita.

### Requisiti non funzionali

- All'ultimo piano il giocatore affronterà un boss finale dotato di comportamenti unici e di una difficoltà superiore rispetto ai nemici ordinari.
- La difficoltà si adatta dinamicamente in base al percorso del giocatore.
- È prevista una musica di sottofondo e feedback sonori per ogni azione significativa.
- Sono disponibili diverse tipologie di armi, la cui efficacia varia in base al tipo di nemico affrontato.

## 1.2 Modello del Dominio

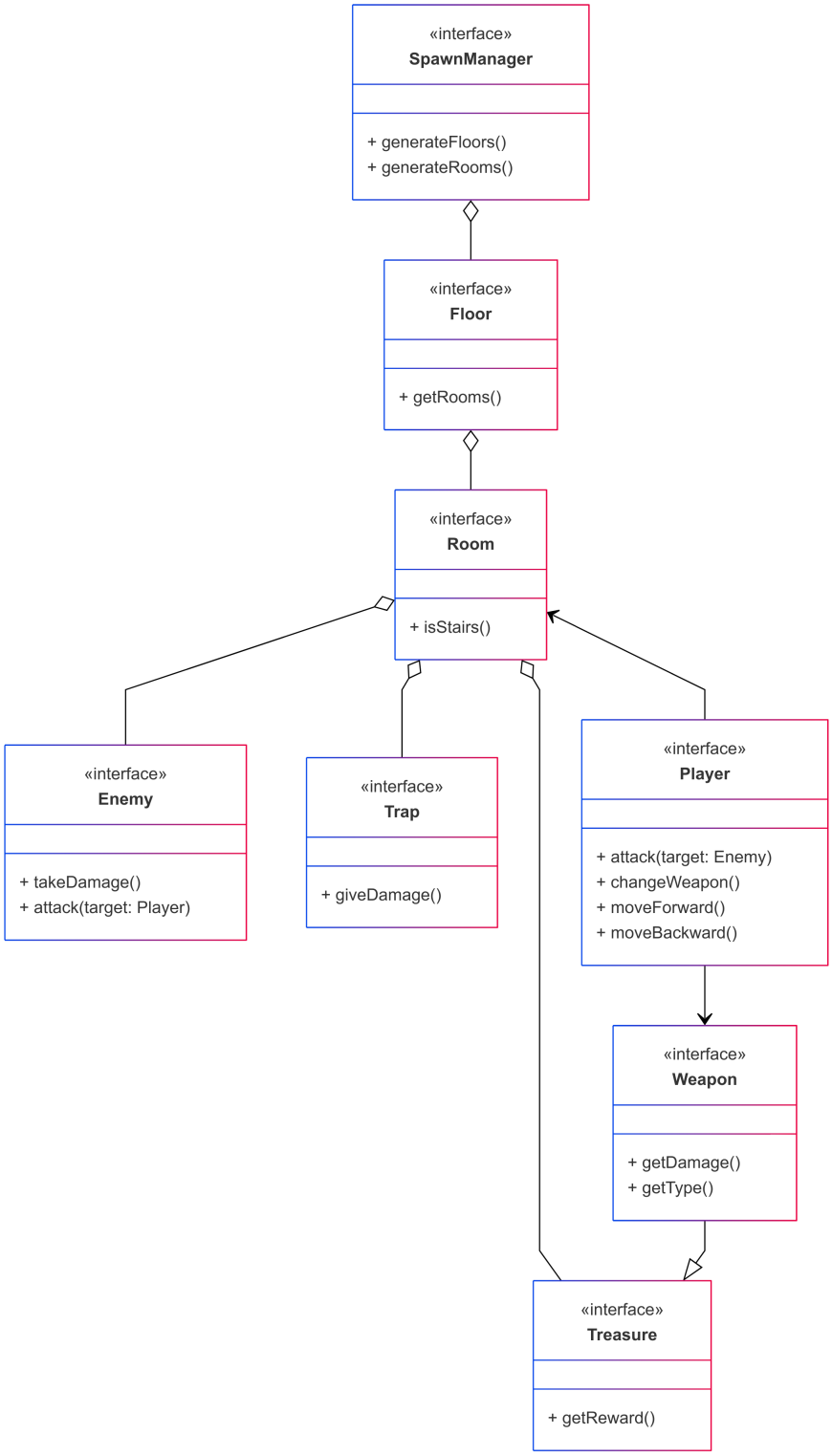
**Temple Tower** verrà rappresentato da una torre composta da un certo numero di piani (*Floor*), ciascuno dei quali ospiterà stanze interconnesse. Ogni stanza potrà contenere una trappola, un nemico, un tesoro e ospitare il giocatore.

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

Il giocatore potrà utilizzare armi per affrontare i nemici. Poiché gli attacchi avvengono a turni, è fondamentale bilanciare le statistiche per evitare vittorie "matematiche" dovute a disparità nell'efficacia delle armi. Il giocatore e i nemici disporranno di diverse tipologie di mosse d'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 disporranno di barre di stato: **vita**.

- **Barra della vita**: si riduce subendo danni da nemici o trappole. Quando raggiunge lo 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 d'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**:
  - La view è gestita da un `SceneManager` che, mediante l'implementazione di un pattern Factory, si occupa del cambio della vista in base alla situazione attuale.

  - Grazie al manager è possibile decentralizzare le responsabilità della view in diverse classi, ciascuna responsabile della gestione degli eventi della propria interfaccia.

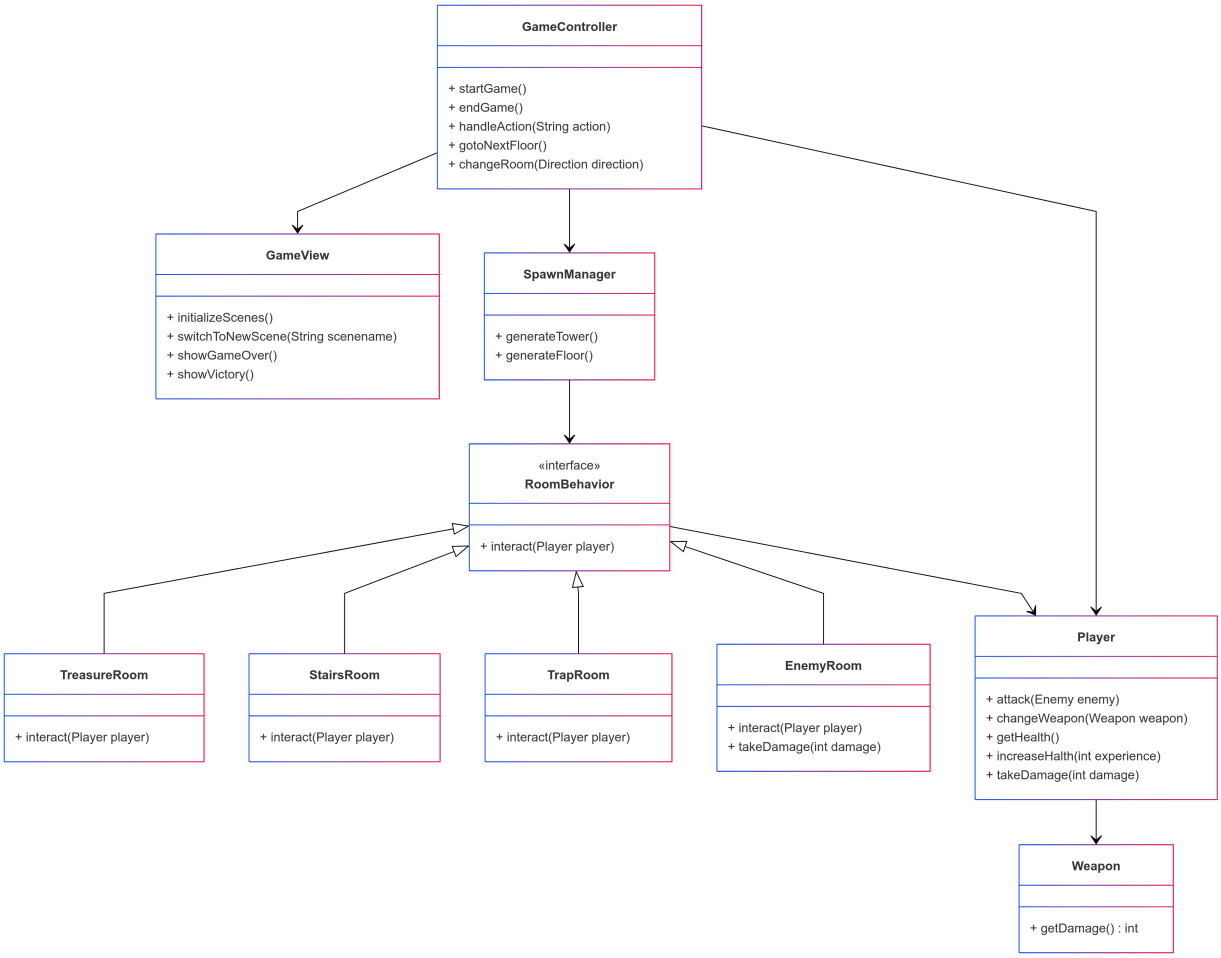
- **Controller**:
  - La classe `GameController` funge da intermediario tra il modello e la vista, orchestrando il flusso degli eventi nel gioco.
  - Gestisce le azioni dell'utente, permettendo al giocatore di muoversi tra le stanze (metodo `changeRoom`) o salire al piano successivo (metodo `gotoNextFloor`), e 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 modificare singole componenti senza influenzare l'intero sistema.

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 netta 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 della riproduzione, gestisce il ciclo di vita del clip audio e lo stato interno del sistema.
- **Controller (MusicController)**: Gestisce le operazioni sulla riproduzione musicale, fungendo 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 si occupa dei dettagli tecnici mentre il MusicController gestisce il flusso principale.

**Contro**: Richiede una buona coordinazione tra le classi.

**Diagramma UML**

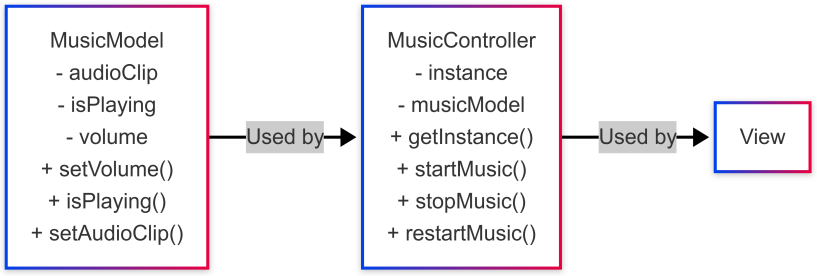



Figura 2.1: Diagramma UML del sistema di gestione audio

### Pattern Singleton

Il pattern Singleton è stato utilizzato nella classe `MusicController` per garantire l'unicità dell'istanza del controller musicale durante l'esecuzione dell'applicazione, evitando conflitti e garantendo un punto d'accesso globale alla gestione audio.

### Metodo Template: getInstance()

**Motivazione**: Assicura l'unicità dell'istanza di `MusicController`, facilitando la gestione centralizzata della riproduzione musicale. Il pattern Singleton è stato implementato correttamente, considerando che la gestione centralizzata richiede una sola istanza.

### 2.2.2 Montanari

**Gestione dei Popup tramite Factory Method**

**Problema**
Nel gioco esistono diversi tipi di finestre di dialogo (popup) per 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 duplicazioni e a una minore manutenibilità. Inoltre, l'assenza di un'adeguata astrazione comporterebbe la necessità di riscrivere la struttura per ogni nuovo popup, aumentando il rischio di incoerenze.

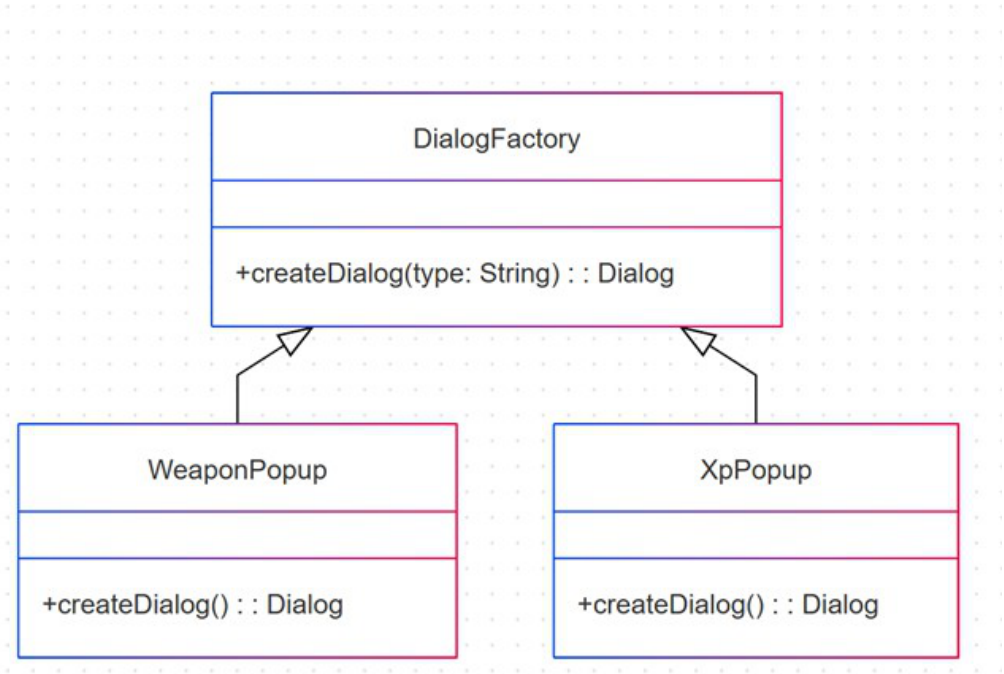
**Soluzione**
È stato adottato il Factory Method per delegare la creazione dei popup a una classe centralizzata (`DialogFactory`). Questa classe definisce il metodo `createDialog` che, in base al tipo di popup richiesto, restituisce un oggetto `Dialog` con il contenuto e il comportamento appropriati.

Questa soluzione offre diversi vantaggi:

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

Un'alternativa considerata era l'uso di una classe `DialogUtil` con metodi statici per ogni tipo di popup, ma tale approccio avrebbe reso più difficile l'estensione senza modifiche dirette, violando il principio Open/Closed.

**Schema UML**




**Applicazione del Pattern Factory Method**

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

Questa implementazione migliora la separazione delle responsabilità e facilita l'aggiunta di nuovi tipi di popup in futuro.

### Gestione della Sincronizzazione tra Attacco, Punti Vita e UI

**Problema**
Nella gestione della scena di combattimento è necessario sincronizzare correttamente l'attacco del giocatore, la riduzione dei punti vita dell'avversario e l'aggiornamento dell'interfaccia utente (UI). Una gestione errata potrebbe causare ritardi o incoerenze visive, impattando negativamente il gameplay.

**Soluzione**

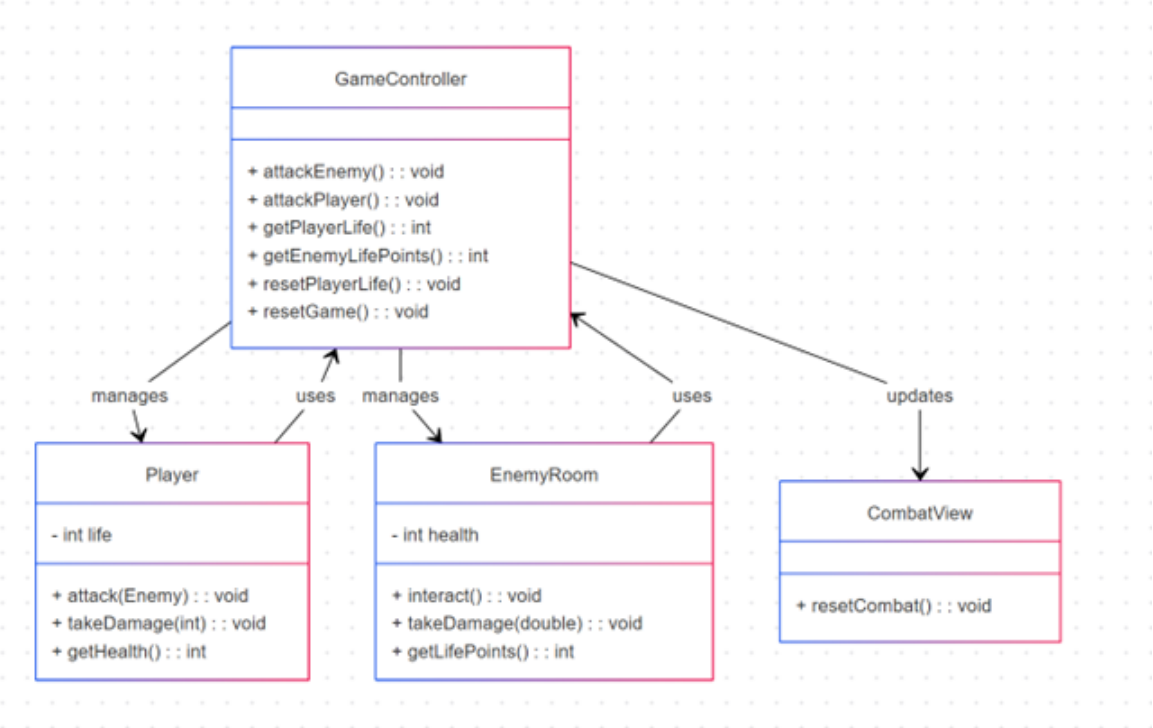
La soluzione prevede una gestione sequenziale delle azioni tramite:

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

Il flusso esecutivo prevede:

1. Esecuzione dell'attacco tramite animazione (ad esempio, una fiamma o un movimento in avanti del giocatore).
2. Al termine dell'animazione, applicazione del danno riducendo i punti vita dell'avversario.
3. Aggiornamento della UI, modificando la barra della salute (`ProgressBar`) e il valore numerico degli HP.
4. Se il nemico è ancora in vita, attivazione di un ritardo tramite `PauseTransition` per simulare il tempo di reazione.
5. Esecuzione del contrattacco nemico con conseguente aggiornamento della UI.
6. Verifica dello stato del combattimento:
7. In caso di vittoria del giocatore, il pulsante d'attacco viene disabilitato e la salute ripristinata.
8. In caso di sconfitta, viene mostrato un popup che reindirizza al menu principale.

**Schema UML**




**Benefici ottenuti**

Questa soluzione garantisce una transizione fluida tra attacco, difesa e aggiornamento della UI, prevenendo problemi di desincronizzazione. L'uso di `Platform.runLater` assicura aggiornamenti sicuri nel thread principale di JavaFX, migliorando la responsività dell'applicazione. Inoltre, le pause controllate migliorano il feedback visivo per il giocatore.

### 2.2.3 Vignali

**Modular Game Data Loading and Tower Configuration**

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

**Soluzione** Sono stati valutati due approcci:

1. Un sistema di caricamento decentralizzato in cui ogni componente gestisce i propri dati;

2. Un gestore centralizzato che coordina l'intero caricamento.

È stata scelta la seconda opzione, implementando GameDataManagerImpl come Singleton per garantire un punto d'accesso unico. Pur considerando l'uso della Dependency Injection, la natura globale dei dati e la necessità di mantenerli coerenti hanno reso il Singleton l'opzione migliore.

Il sistema utilizza record immutabili (FloorData, Enemy, Weapon) e custom deserializer GSON per garantire type-safety e validazione durante il caricamento. Questa soluzione separa nettamente i dati dalla logica, agevolando 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) da cartelle o file ZIP, gestendo validazione, conflitti di nomi e isolamento tra mod.

**Soluzione** Sono state considerate due architetture:

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

È stata scelta la seconda, che offre una separazione più chiara delle responsabilità e una gestione dello stato più prevedibile. Il pattern Observer è preferito rispetto ai callback per la facilità di estensione senza modifiche al codice esistente.

È stato inoltre implementato il pattern Strategy per supportare nuovi formati di mod, oltre a ZIP e cartelle.

Schema UML:

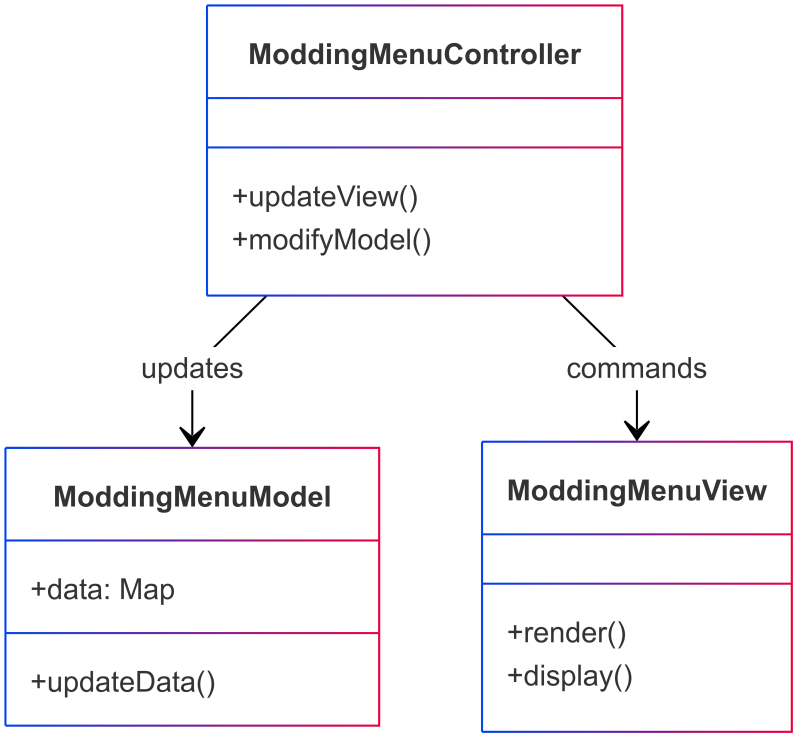



Figura 2.3: Modding Menu UML: MVC and Observer pattern overview.

### Gestione dello Spawn System

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

**Soluzione** Sono stati valutati due approcci:

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

È stato scelto il secondo approccio, implementando un Template Method in SpawnManagerImpl che standardizza il processo di generazione, permettendo variazioni nel comportamento specifico. Il metodo template si articola in 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**

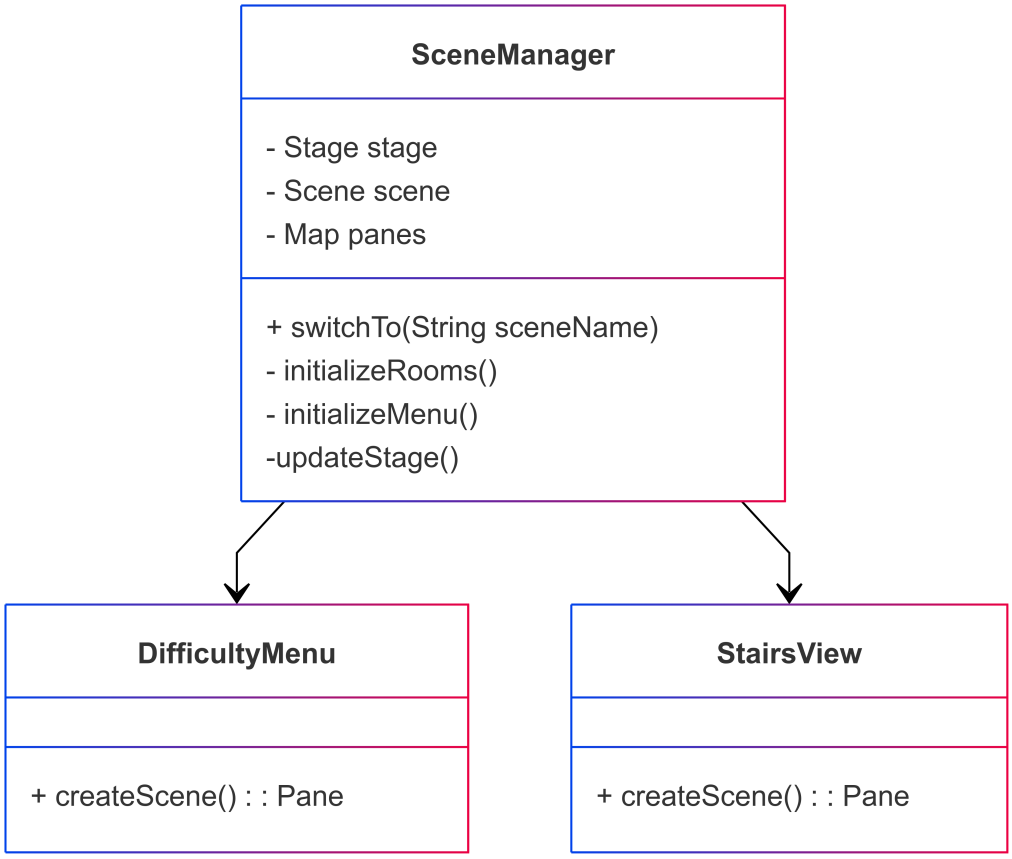



Figura 2.4: Rappresentazione UML del pattern Factory per la gestione delle viste (esempio con 2 viste)

**Problema** Il gioco presenta diverse viste (menu iniziale, viste delle stanze, ecc.) e occorre gestire la visualizzazione in modo coerente.

**Soluzione** Il sistema utilizza il pattern Factory: le classi che implementano le viste (`NomevistaView`) vengono registrate nel mapping interno del `SceneManager`, il quale si occupa di:

- Inizializzare le viste relative alle schermate iniziali all'avvio del gioco.
- Inizializzare le viste degli elementi di gioco dopo il caricamento della torre.

- Caricare e visualizzare il nuovo Pane tramite il metodo `switchTo`.

Questa soluzione consente di passare agevolmente da una vista all'altra, facilitando l'aggiunta di nuove interfacce.

### Stanze del gioco

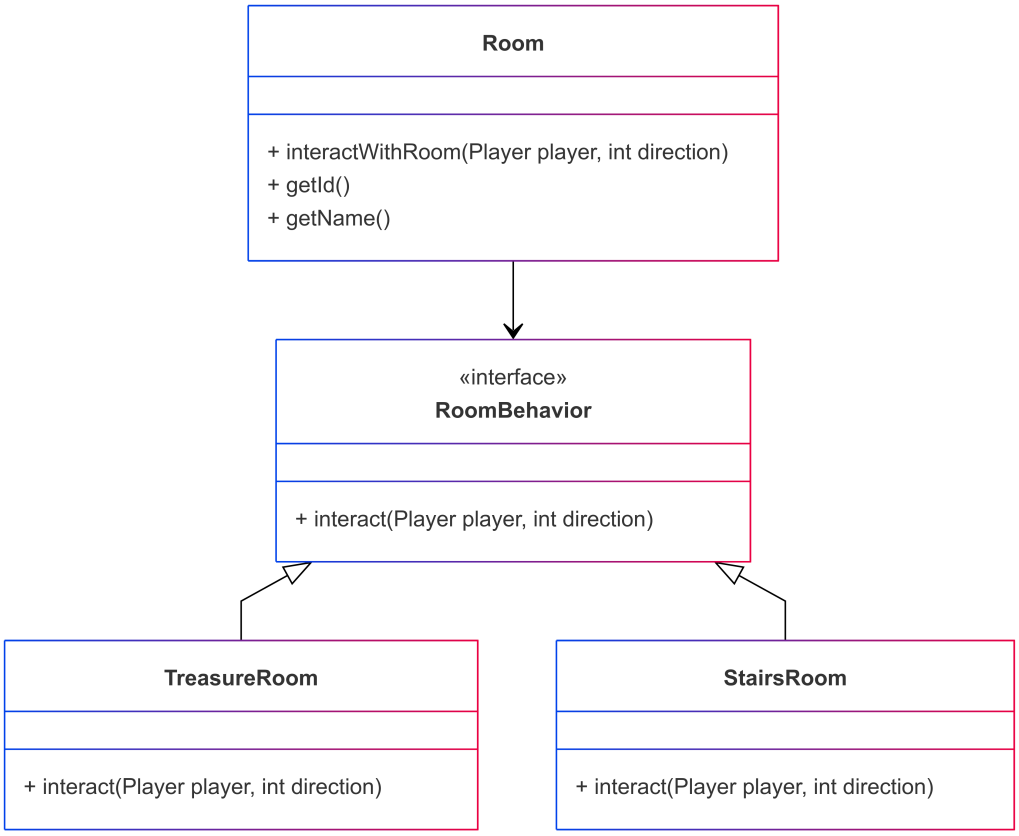



Figura 2.5: Rappresentazione UML del pattern Strategy per le stanze del gioco

**Problema** Temple Tower prevede diverse tipologie di stanze, ciascuna con contenuti differenti.

**Soluzione** Il sistema usa il pattern Strategy per modellare le stanze: le implementazioni di `RoomBehavior` possono essere facilmente sostituite. Ad esempio, il metodo `interact()` nella classe `Trap` sottrarrà punti vita al giocatore, mentre nella stanza `Stairs` il metodo cambierà il piano.

Questo pattern consente di ampliare i tipi di stanze con minime modifiche al codice esistente.

# Capitolo 3

# Sviluppo

## 3.1 Testing automatizzato

Per il testing generale di Temple Tower è stata utilizzata la suite JUnit.

### Controller

Nel controller sono state testate tutte le classi che vi interagiscono, oltre alla view:

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

## 3.2 Note di sviluppo

### 3.2.1 Cobo

#### Uso di parti della libreria JDK non spiegate a lezione

Uso di `Clip` per gestire la riproduzione audio permalink.

#### Uso di librerie di terze parti

Uso di SLF4J per il logging permalink.

### 3.2.2 Vignali

#### Utilizzo della libreria gson

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

#### Utilizzo della libreria JavaFX

Utilizzato all'interno di ModdingMenuView insieme a codice CSS: permalink.

#### Utilizzo della libreria SLF4J

Impiego prevalente per la gestione dei log, sia nei test che nel codice: permalink.

#### Utilizzo della libreria Apache Commons IO

Impiegata per la gestione dei file, in combinazione con java.util, durante l'importazione: permalink.

#### Utilizzo della libreria Java util zip

Utilizzata per la decompressione dei file: permalink.

#### Utilizzo di stream e lambda expressions

Impiego in diverse parti del codice – esempio: permalink.

#### Utilizzo di Optional

Utilizzato in varie parti del codice – esempio: permalink.

### 3.2.3 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.

### 3.2.4 Montanari

#### Uso di SLF4J per il logging

Il progetto utilizza SLF4J per la gestione dei log, garantendo un monitoraggio efficace degli eventi di gioco.
Esempio:

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

Link al codice su GitHub.

#### Uso di Lambda Expressions

Per gestire gli 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` garantisce operazioni sicure.
Esempio:

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

Link al codice su GitHub.

### Utilizzo di Timeline e KeyFrame per le animazioni

Il codice impiega `Timeline` e `KeyFrame` per creare effetti visivi durante attacchi e riduzioni della vita.
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 UI è costruita utilizzando StackPane, HBox, VBox e BorderPane.
Link al codice su GitHub.

# Capitolo 4

# Commenti finali

## 4.1 Vignali

Rileggendo il mio contributo, ritengo di essere riuscito a ottenere un buon equilibrio tra la parte strutturale e quella relativa alla gestione dei dati JSON. Alcuni dati precedentemente previsti sono stati omessi per non appesantire il lavoro dei compagni, mentre altre parti sono state trattate in maniera sintetica ma efficace. La feature relativa all'interfaccia per torri personalizzate, ad esempio, è stata rimandata per mancanza di tempo, ma rappresenta un potenziale sviluppo futuro.

## 4.2 Mularoni

Considerando il tempo a disposizione e gli impegni, reputo il progetto "accettabile", sebbene con un grande potenziale inespresso, sia per la scarsa divisibilità del lavoro che per l'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

L'applicazione è stata progettata per essere intuitiva e di facile utilizzo. Di seguito, una breve guida per iniziare a giocare:

1. **Avvio e Menu Iniziale:** All'avvio, verrà mostrata la schermata iniziale. Cliccare sul pulsante centrale per accedere al menu principale.
2. **Importazione della Torre:**
   - Dal menu principale, aprire il *Modding Menu* per importare una torre.
   - È possibile caricare una torre presente nella repository del progetto o scaricarla tramite il seguente link Tower Download.
   - Ricordarsi di cliccare sul tasto Select per selezionare la torre.
3. **Inizio della Partita:**
   - Dopo aver scelto la torre, premere il pulsante centrale per avviare il gioco.
   - Si consiglia di impostare la difficoltà su *facile* per facilitare il testing e l'apprendimento delle dinamiche di gioco.
4. **Navigazione Durante il Gioco:**
   - Utilizzare i pulsanti in basso per muoversi a destra o a sinistra nelle stanze.
   - Cliccare il pulsante centrale *Enter* per entrare in una stanza. Una volta all'interno, seguire le istruzioni visualizzate per interagire con gli elementi della stanza.

**Nota:** Assicurarsi che la torre selezionata sia accessibile (basta permesso di lettura) e che l'applicazione abbia il permesso di creare una cartella user.

# 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`