Tommaso Bandini
tommaso.bandini5@studio.unibo.it

Tommaso Goni
tommaso.goni@studio.unibo.it

Luigi Linari
luigi.linari@studio.unibo.it

Claudio Lodi
claudio.lodi@studio.unibo.it

Giugno 2025

Indice

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Il software, mira alla realizzazione di un gioco "Tower Defense" in stile medievale, che prende ispirazione da videogiochi come Bloons TD, Kingdom Rush e simili. Il gioco prevede un percorso prestabilito sul quale orde di nemici cercano di invadere un castello, lo scopo del giocatore è difenderlo scegliendo strategicamente dove posizionare le varie tipologie di torri offerte.

La partita si svolge iterando ondate, esse rappresentano le fasi di avanzamento temporale.

Requisiti funzionali

- La partita si conclude solo se la vita del giocatore scende a 0 o si arriva alla termine dell'ultimo livello.
- Le torri a dispozione devono permettere diverse configurazioni, esse consistono nella scelta della tipologia di proiettile (grande/piccolo) e della tipologia di incantamento (fuoco/ghiaccio).
- Ogni torre ha un sistema di mira a se che le consente di mirare: al nemico più avanti nel percorso, quello più indietro, il più resistente, il più debole o il più vicino alla torre stessa.
- Comporre la mappa di percorsi, punti strategici per il posizionamento delle torri, punti di partenza dei nemici e punto di arrivo (castello).
- I nemici si devono muovere lungo il percorso e, una volta raggiunto il castello, devono arrecare danno al giocatore.

- Il giocatore deve possedere un parametro per i soldi, utilizzabili per acquistare le torri, ed uno per i punti vita.

Requisiti non funzionali

- la finestra deve essere ridimensionatile.
- il gioco deve funzionare correttamente su Windows, Mac e Linux.
- il gioco ha una frequenza di aggiornamento di 20 tick/s, e deve mostrarsi a FPS massimi.

1.2 Modello del Dominio

1.2.1 Mappa

La mappa è costituita da percorsi definiti a priori, anche per il verso di percorrenza, che i nemici rispettano senza eccezioni.

1.2.2 Nemici

L'obiettivo dei nemici è raggiungere il castello per danneggiare il giocatore. Inoltre i nemici hanno diverse proprietà, tra cui:

- punti vita;
- velocità di movimento.

e devono poter essere soggetti ai vari incantamenti delle torri.

1.2.3 Torri

Le torri scelgono i nemici a cui sparare all'interno del loro raggio di azione in base alla priorità di mira scelta dall'utente. Quando una torre viene costruita, le vengono assegnate delle statistiche base. Attraverso gli upgrade è possibile applicare ai proiettili un incantamento a scelta tra:

- Ghiaccio: rallenta i nemici;
- Fuoco: danneggia i nemici.

e una dimensione a scelta tra:

- Grandi: sono più lenti, fanno danno ad area ed hanno una minore velocità di fuoco;
- Piccoli: sono più veloci e con una maggiore velocità di fuoco.

Sia l'incantamento che la dimensione possono essere potenziati di 4 livelli.

1.2.4 Wave

Il gioco si sviluppa in waves. Ogni wave è composta da vari gruppi di nemici che partono dai punti di spawn marciando verso il castello. Esse diventano progressivamente più difficili ed ognuna termina quando tutti i nemici da lei generati vengono sconfitti o raggiungono il castello. Alcune wave provocano un cambiamento strutturale della mappa come:

- un allargamento del campo visibile;
- nuovi percorsi per i nemici;
- nuovi punti di spawn;
- nuove posizioni per le torri.

1.2.5 Player

Il giocatore ha due parametri, oro e vita. L'oro può essere utilizzato per acquistare nuove torri ed effettuare gli upgrade. È possibile ottenerlo attraverso l'eliminazione dei nemici e i reset delle torri. I punti vita diminuiscono ogni volta che un nemico raggiunge il castello. Non è possibile ottenere nuovi punti vita.

1.2.6 Gui

All'apertura del gioco si presenta il menu principale dal quale è possibile accedere alla selezione dei salvataggi e alle opzioni di gioco. In gioco si può accedere alle opzioni, mentre a fine partita verrà visualizzata la schermata di game over con le statistiche e la possibilità di tornare al menu principale.

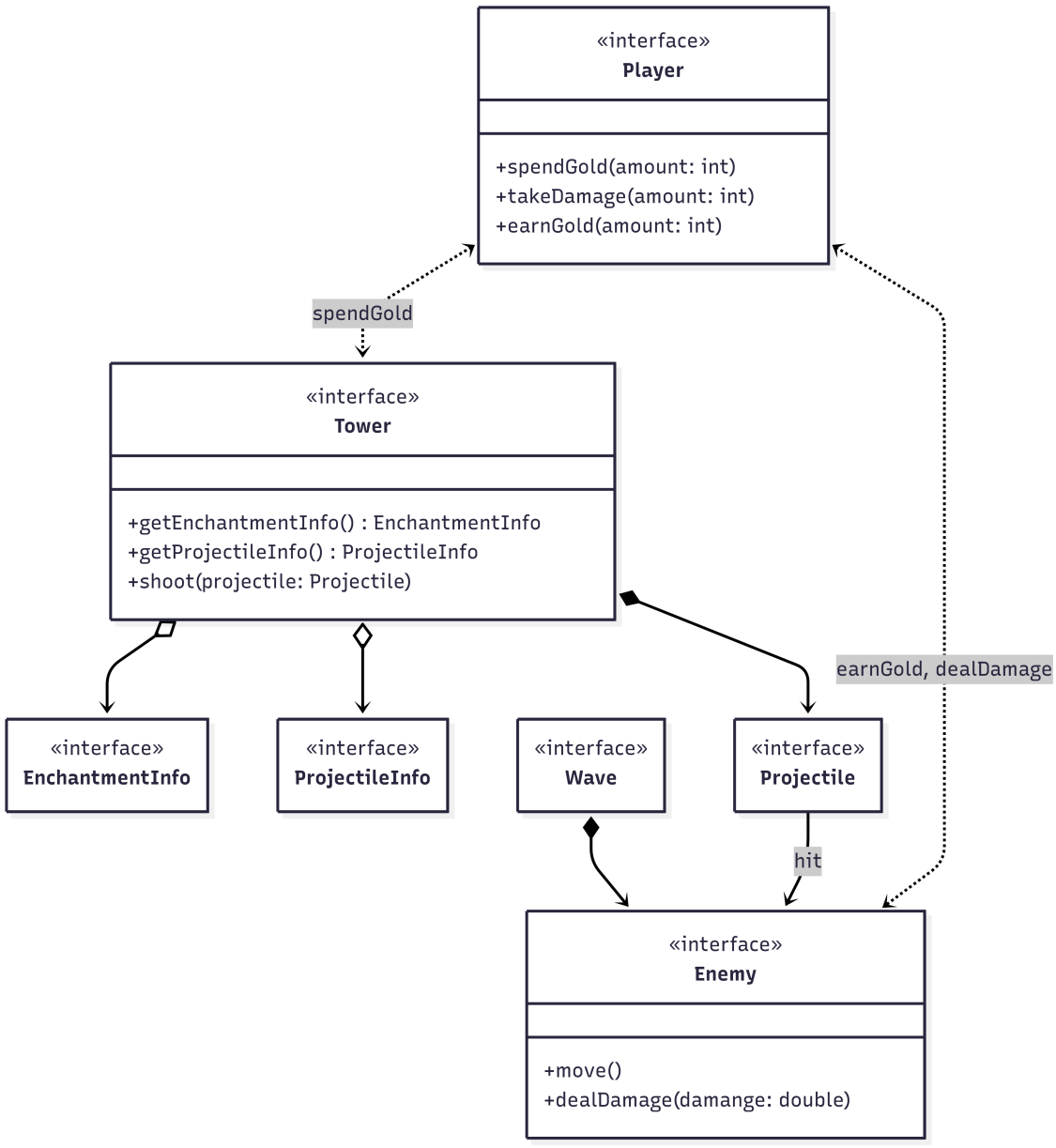

Figura 1.1: Schema UML del dominio, con rappresentate le entità principali ed il rapporto fra loro

Capitolo 2

Design

2.1 Architettura

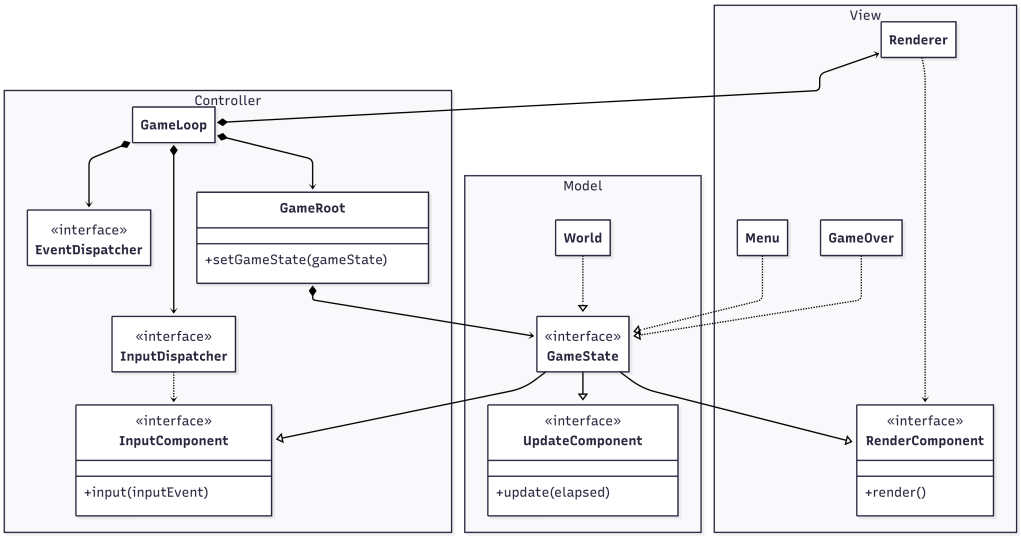

Figura 2.1: Schema UML dell' architettura

Il sistema adotta un'architettura MVC con pattern component-based[1]. Ciascuna entità di gioco mantiene autonomamente il proprio stato (model), mentre le operazioni di aggiornamento logico e rendering grafico sono delegate ai propri componenti: implementazioni di *UpdateComponent* e *RenderComponent*.

[1]https://gameprogrammingpatterns.com/component.html

L'architettura adotta una struttura ad albero per gestire input, aggiornamenti e rendering. Gli impulsi del game loop fluiscono dalla radice (*GameRoot*) verso i nodi figli attraverso tre canali indipendenti:

- Input: L'*InputDispatcher* inietta gli input nella radice. Ogni nodo decide se consumare l'input, propagarlo ai figli o ignorarlo.
- Update: Il *GameLoop* manda gli impulsi di aggiornamento alla radice che verranno propagati in profondità attraverso tutto l'albero.
- Rendering: Per ogni frame, il *GameLoop* manda un impulso di rendering alla radice, i nodi dell'albero interessati ad essere renderizzati notificano il *Renderer* che alla fine della propagazione dell'impulso avrà collezionato e interpretato tutte le richieste per la formazione del frame.

Questa suddivisione favorisce l'intercambiabilità della libreria grafica in quanto andrebbero apportate modifiche ai soli render component senza andare ad intaccare controller e modello.

2.2 Design dettagliato

2.2.1 Tommaso Bandini

Caricamento degli sprite

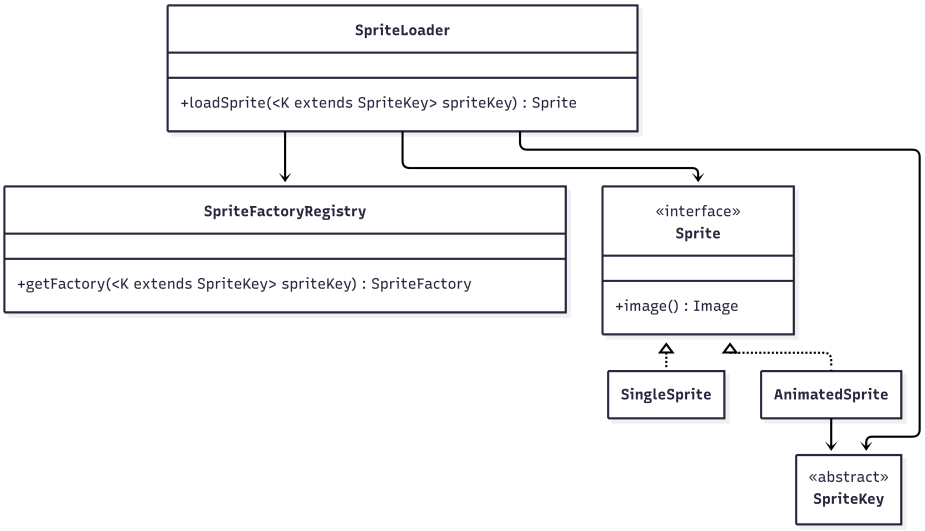

Figura 2.2: Schema UML dello SpriteLoader

Problema Nel gioco gli sprite rappresentano elementi visivi come nemici, torri, proiettili, GUI o altri asset grafici animati o statici. Un caricamento ridondante di questi oggetti o una gestione poco efficiente di essi causerebbe rallentamenti e un utilizzo non necessario della memoria.

Soluzione Per risolvere il problema ho creato una classe centralizzata per il caricamento di ogni tipo di sprite. Quando viene richiesto uno sprite, essa verifica se è già stato caricato in precedenza. In tal caso, lo prende dalla sua cache interna; altrimenti, lo carica una volta e lo conserva nella cache per utilizzi futuri.

Ogni sprite viene caricato tramite una factory specifica per quel tipo, ognuna di esse viene registrata allo sprite loader tramite lo *SpriteFactoryRegistry*. Questo approccio consente di mantenere lo sprite loader semplice e allo stesso tempo di poter aggiungere con facilità nuove tipologie di sprite.

Player

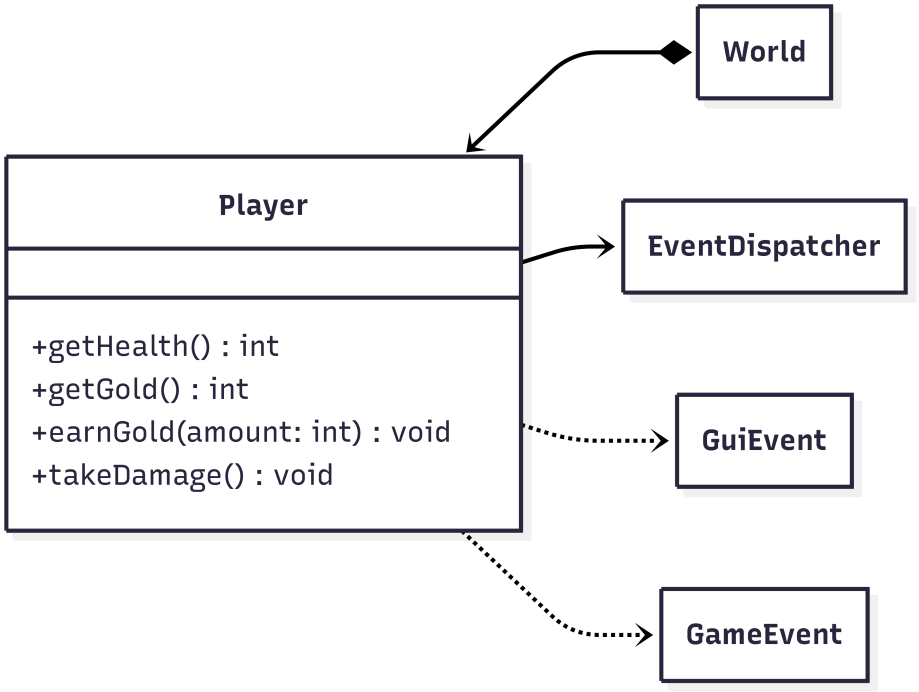

Figura 2.3: Schema UML del player

Problema Il giocatore deve interagire con il mondo di gioco in modo reattivo e indipendente dai dettagli interni: costruire o potenziare torri, subire danni e innescare la fine della partita senza inserire la logica di gioco direttamente nelle entità del mondo.

Soluzione La classe *Player* incapsula stato e comportamenti del giocatore (punti vita e oro) e si affida al sistema a eventi per comunicare con il resto dell'applicazione. Questo approccio realizza il pattern Observer: Player è un listener registrato presso l'*EventDispatcher* e reagisce automaticamente ai messaggi di interesse. Grazie a questo disaccoppiamento, la logica di acquisto, danno e fine partita è confinata al *Player*, mentre le parti di rendering, fisica e gestione delle torri rimangono autonome.

GUI

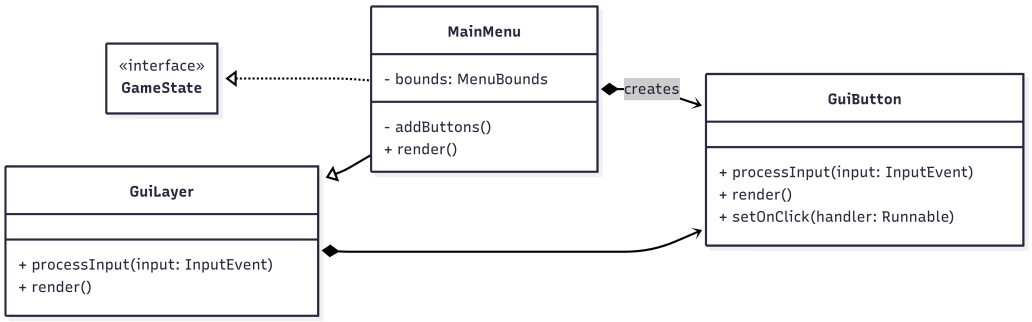

Figura 2.4: Schema UML delle GUI, in questo caso specifico mi era comodo specificare per il MainMenu, l'approccio è il medesimo per le altre GUI

Problema Nel flusso di gioco è necessario gestire i cambi di "stato" dettati dalla navigazione tramite le GUI. Le GUI richiedono una struttura capace di raggruppare elementi grafici interattivi, di processare i click dentro aree precise e di disegnare elementi statici e dinamici in modo modulare.

Soluzione Per risolvere il problema ho reso *GameRoot* responsabile della gestione del flusso di gioco, delegando la logica di ciascuna schermata a ogni specifico stato. In particolare ogni videata implementa l'interfaccia *GameState*, definendo quindi come gestire Input, Update e Render. L'aggiunta dei pulsanti avviene in fase di *render* permettendo cambi di layout nel caso sia necessario. La gestione degli input è gestita internamente da *GuiLayer*, i click vengono prima filtrati contro i confini del layer e poi inoltrati a ciascun *GuiButton* che a sua volta scatena l'evento specifico tramite l'*EventDispatcher*.

Audio

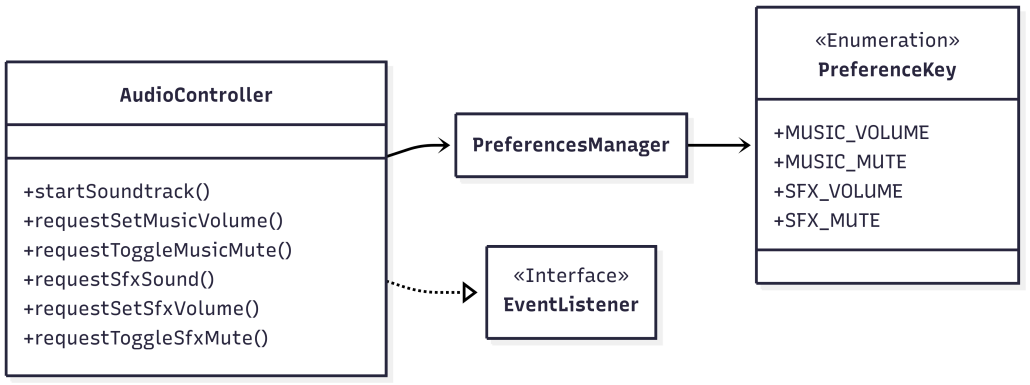

Figura 2.5: Schema UML del AudioController

Problema Sistema di gestione audio con regolazione indipendente di musica ed effetti sonori dalle opzioni di gioco e persistenti anche dopo la chiusura e la riapertura del gioco.

Soluzione L'*AudioController* implementa un sistema audio dove musica ed effetti sonori sono gestiti separatamente attraverso l'integrazione del *PreferencesManager* per memorizzare volume/stato muto in preferenze persistenti e modifiche in tempo reale tramite il pattern *Observer*.

2.2.2 Tommaso Goni

Rendering

Problema Il rendering del gioco richiede la gestione di diversi elementi (sfondo, entità, GUI, ecc.) che devono essere disegnati in un ordine specifico per garantire corrette sovrapposizioni visive (es. GUI sopra le entità). È necessario un sistema flessibile che permetta l'aggiunta dinamica di operazioni di rendering e che ne disaccoppi la logica dalle varie invocazioni.

Soluzione In figura 2.6 è schematizzata la soluzione adottata: è stato implementato un sistema task-based con priorità configurabili.

La classe *RenderTask* incapsula un'operazione di rendering e una priorità globale (*RenderPriority*), determinata dalla categoria di appartenenza

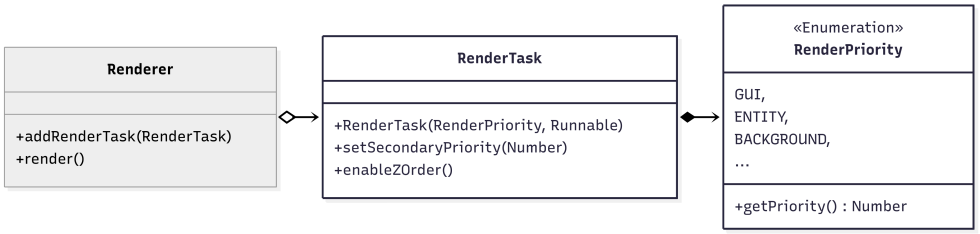

Figura 2.6: Schema UML: Renderer

dell'operazione (ad esempio, rendering della GUI). Per garantire una maggiore flessibilità, *RenderTask* offre inoltre due metodi opzionali per risolvere le situazioni di pari priorità tra operazioni appartenenti alla stessa categoria.

Il *Renderer* è incaricato di collezionare i *RenderTask* e di eseguirli nell'ordine corretto facendo da tramite con la libreria grafica.

Vantaggi La soluzione garantisce un disaccoppiamento architetturale tra le componenti di gioco e il sottosistema di rendering: gli elementi definiscono esclusivamente cosa renderizzare, delegando al Renderer la gestione dell'ordinamento e dell'esecuzione. Questo approccio garantisce un buon livello di estensibilità, consentendo l'introduzione di nuove categorie di rendering e criteri di ordinamento secondari attraverso modifiche localizzate.

Visualizzazione statistiche della torre

Problema Quando l'utente seleziona una torre, il gioco deve mostrare in modo chiaro e immediato le sue statistiche (come *Velocità di fuoco*, *Danno*, *Raggio d'azione*, ecc.) e, ogni volta che l'utente apporta modifiche, deve mettere a confronto le nuove e le vecchie statistiche per evidenziare le variazioni in modo intuitivo.

Esempio L'utente seleziona una torre e vuole valutare se valga la pena effettuare un upgrade: tramite il sistema di visualizzazione delle statistiche osserva che l'upgrade andrebbe a potenziare il *Raggio d'azione* di 3 punti, che il *Danno* non verrebbe modificato e che si aggiungerebbe una nuova statistica *Effetto: Brucia*.

Soluzione Si è voluto ottenere un sistema che riceva in ingresso una lista di tutte le statistiche da smistare, ogni statistica marcata come base della comparazione o come statistica di confronto. Come output deve essere fornita

una lista strutturata pronta ad essere rappresentata visualmente nel gioco come posto nel problema.

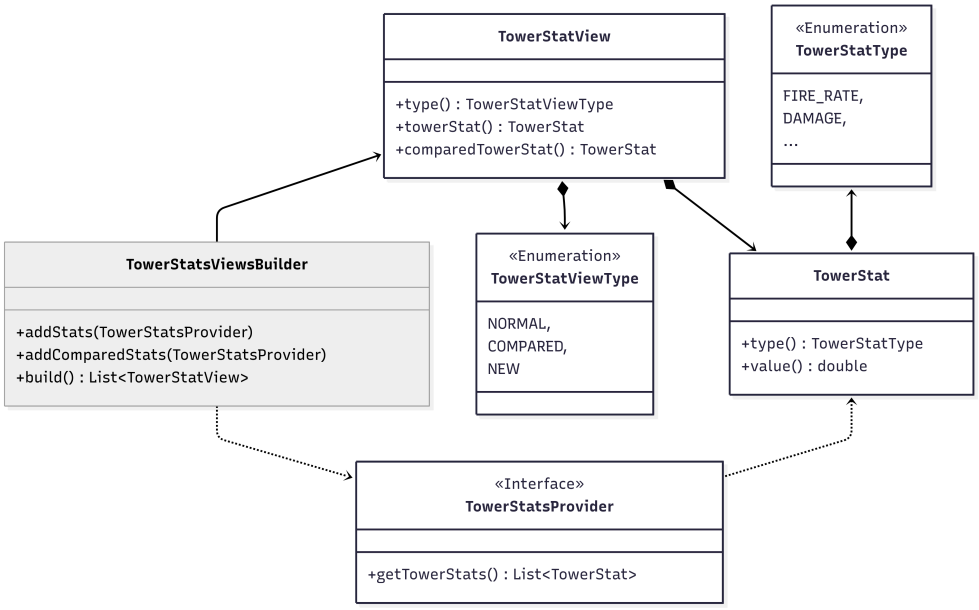

Figura 2.7: Schema UML: Sistema delle statistiche della torre.

In figura 2.7 una schematizzazione della soluzione adottata: la classe *TowerStat* modella le statistiche e *TowerStatView* modella come una statistica può venire rappresentata: come invariata, nuova o con il valore alterato (comparata).

È stato scelto il **Builder** pattern implementato dalla classe *TowerStatsViewsBuilder*. Questo pattern permette di costruire progressivamente la lista di viste, offrendo un sistema di utilizzo estremamente semplice.

Tramite l'interfaccia *TowerStatsProvider* il builder è stato disaccoppiato dal resto del gioco: all'utilizzatore basta fornire un'implementazione dell'interfaccia per poter utilizzare il sistema.

Vantaggi Questa architettura rende facile apportarvi modifiche, offre una rigida separazione dei compiti e ne garantisce il riutilizzo.

Alternative considerate Inizialmente era stato considerato il calcolo diretto in UI, escluso perché avrebbe violato il *principio di singola responsabilità*, assegnando alla componente di presentazione compiti di elaborazione dati.

Strategie di Mira

Problema Le torri richiedono diverse strategie di *targeting* (es. "primo nemico lungo il percorso", "nemico più forte", "bersaglio più vicino"). È necessario un sistema flessibile che permetta l'aggiunta di nuove strategie senza modificare il codice esistente e che separi la logica di targeting dalla logica della torre.

Soluzione In figura 2.8 una schematizzazione della soluzione adottata. È stato implementato il pattern **Template Method** tramite la classe astratta *AimStrategy*.

Il metodo template `getOrder()` implementa la struttura fissa dell'algoritmo, mentre il metodo astratto `compare(Enemy, Enemy)` definisce il cuore variabile della strategia, implementato dalle sottoclassi concrete rappresentanti diverse strategie di mira.

È rispettato il *Dependency Inversion Principle*: le strategie di mira low-level e le torri dipendono dall'astrazione (*AimStrategy*). Le torri dipendono esclusivamente dall'astrazione *AimStrategy* e ricevono via iniezione (*Dependency Injection*) l'implementazione concreta della strategia di mira, anziché conoscere classi specifiche. Ciò garantisce basso accoppiamento, facile estensione con nuove strategie e testabilità isolata del comportamento di targeting.

Vantaggi Questo pattern rende facile apportarvi modifiche, aggiungere nuovi sistemi di mira, offre una rigida separazione dei compiti e ne garantisce il riutilizzo. Per l'utilizzatore finale è facile scambiare i vari sistemi di mira in quanto i dettagli implementativi di ogni strategia di mira sono nascosti e si interagisce su un'interfaccia comune.

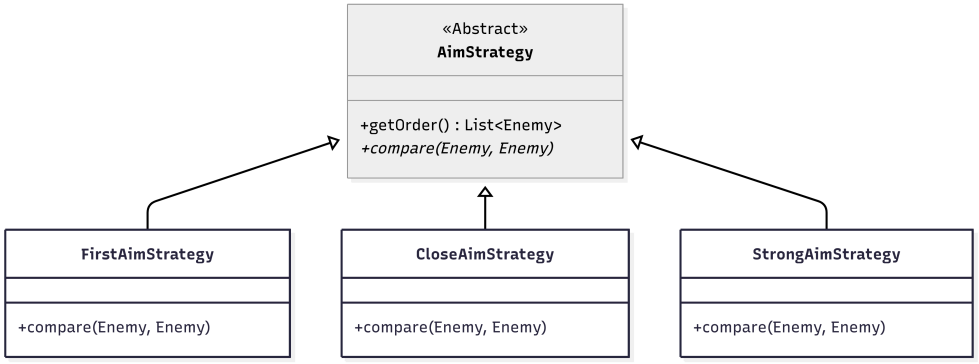

Figura 2.8: Schema UML: Sistema di mira delle torri.

2.2.3 Claudio Lodi

Nemici

Problema All'interno del gioco saranno presenti vari tipi nemici come *Ogre* e *Pig*, è quindi necessario avere una struttura che riutilizzi il codice e renda possibile un'estensione futura.

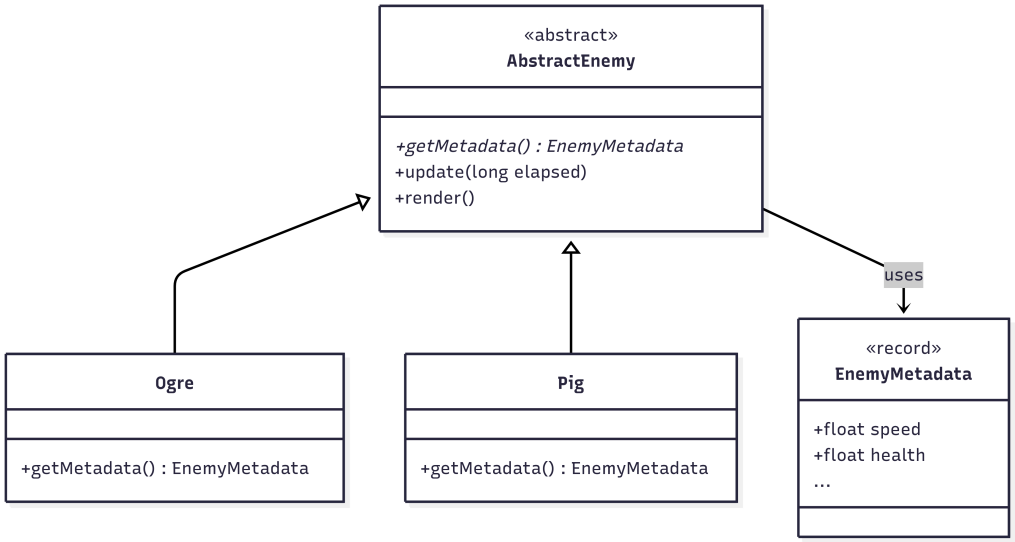

Figura 2.9: Schema UML: struttura per i diversi tipi di nemici.

Soluzione Nonostante un nemico nel gioco può essere di vario tipo, lo stesso però la logica che descrive il suo comportamento è la medesima. Per questo ho deciso di accomunare questo aspetto di ogni nemico in una classe astratta *AbstractEnemy* la quale implementerà appunto il comportamento del nemico basandosi sul contratto *getMetadata()*. Ogni tipo di nemico implementerà quel metodo specificando quindi i valori specifici ai parametri come velocità, vita, ecc...

Problema La creazione di nuovi nemici dovrebbe essere incapsulata, in modo che chi li istanzia non debba conoscere i dettagli interni delle loro classi.

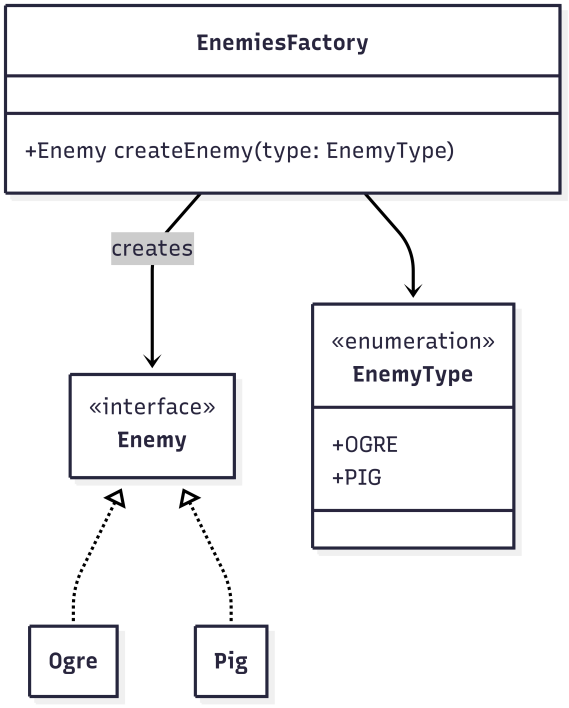

Figura 2.10: Schema UML: funzionamento dell'*EnemiesFactory*.

Soluzione Per affrontare questo problema ho utilizzato il pattern *factory*. L'*EnemiesFactory* si occupa di tener conto dei tipi di nemici instanziabili ed esporre un metodo per creare nuovi nemici in base al tipo richiesto. Grazie a questo pattern, la creazione di nuovi nemici diventa semplice e facilmente estendibile.

Problema Il movimento, non essendo una cosa intrinseca del nemico, c'è bisogno che venga ricavato da un'entità esterna. Ci vuole quindi un contratto che determini come avvengono queste interazioni.

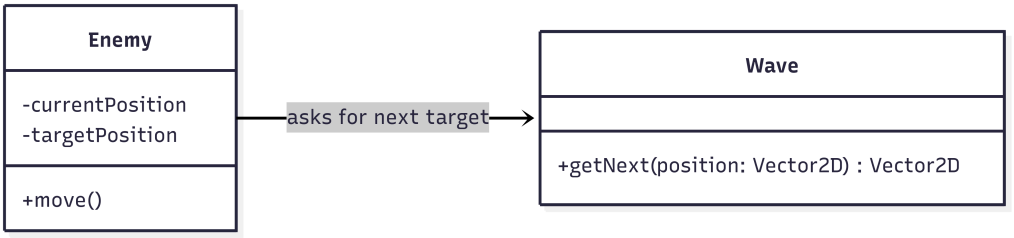

Figura 2.11: Schema UML: movimento nemici.

Soluzione Il nemico si muove sulla mappa attraversando una sequenza di sentieri da punto A a punto B in linea retta, questo significa che una volta raggiunto il punto B sarà necessario chiedere all'oggetto *Wave* la prossima destinazione. Facendo così il nemico non si deve preoccupare di come funziona per esempio lo smistamento nelle intersezioni di più percorsi, disaccoppiando la logica del movimento dall'entità che lo esegue.
Questo approccio segue il principio di separazione delle responsabilità e rende il sistema più estendibile, ad esempio se sarà necessario cambiare la logica di smistamento, il codice del nemico rimarrebbe invariato.

Proiettile

Problema Una volta atterrato, il proiettile ha bisogno di sapere chi ha colpito e interagire di conseguenza. Ci vorrebbe un sistema che disaccoppi il ciclo di vita del proiettile da ciò che accade quando colpisce un nemico.

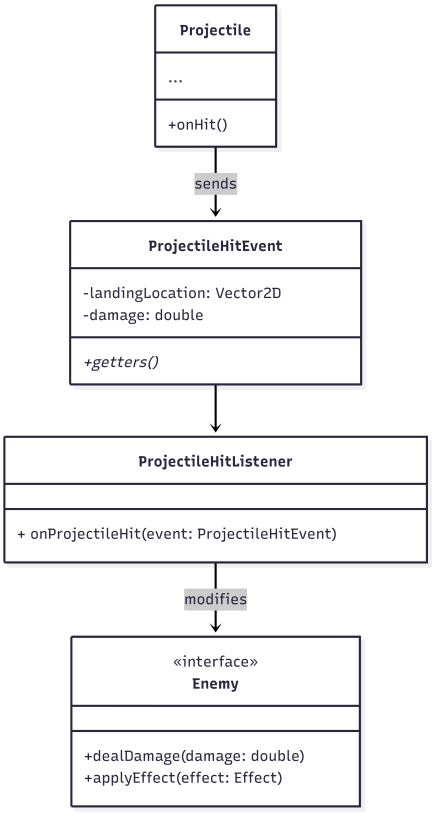

Figura 2.12: Schema UML: hit dei proiettili.

Soluzione Il proiettile all'atterraggio crea un evento *ProjectileHitEvent* il quale conterrà le informazioni del proiettile e dove è atterato. Ci sarà quindi un listener che si preoccuperà di determinare che nemico è stato colpito e interagirci in funzione delle informazioni del proiettile.

2.2.4 Luigi Linari

Mappa

Problema Il percorso va fornito dalla mappa al nemico sotto forma di una serie di nodi, che rappresentano la sequenza di punti che il nemico deve visitare. Inoltre deve distribuirli in modo deciso a priori dallo sviluppatore.

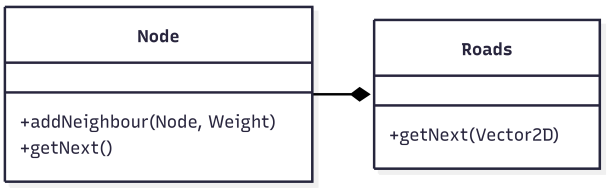

Figura 2.13: schema UML delle classi dei percorsi.

Soluzione Ho modellato ciò attraverso due classi che fungono insieme da grafo di tutti i possibili percorsi di una determinata wave. La classe *Roads* contiene infatti tutti i *Node* di cui sono formati i percorsi, in relazione alla loro posizione. Inoltre ognuno di essi mantiene una lista di altri *Node* a cui è collegato, e verso cui a rotazione direziona i nemici. Attraverso l'assegnazione di un peso ad ogni *Node* di questa lista è quindi possibile modulare il flusso di nemici, a discrezione del programmatore.

Waves

Problema L'organizzazione in wave del gioco: all'avanzare delle wave, infatti, i percorsi possono cambiare, così come anche il modo in cui vengono distribuiti i nemici o il tipo di nemici stessi che entrano sul campo di battaglia.

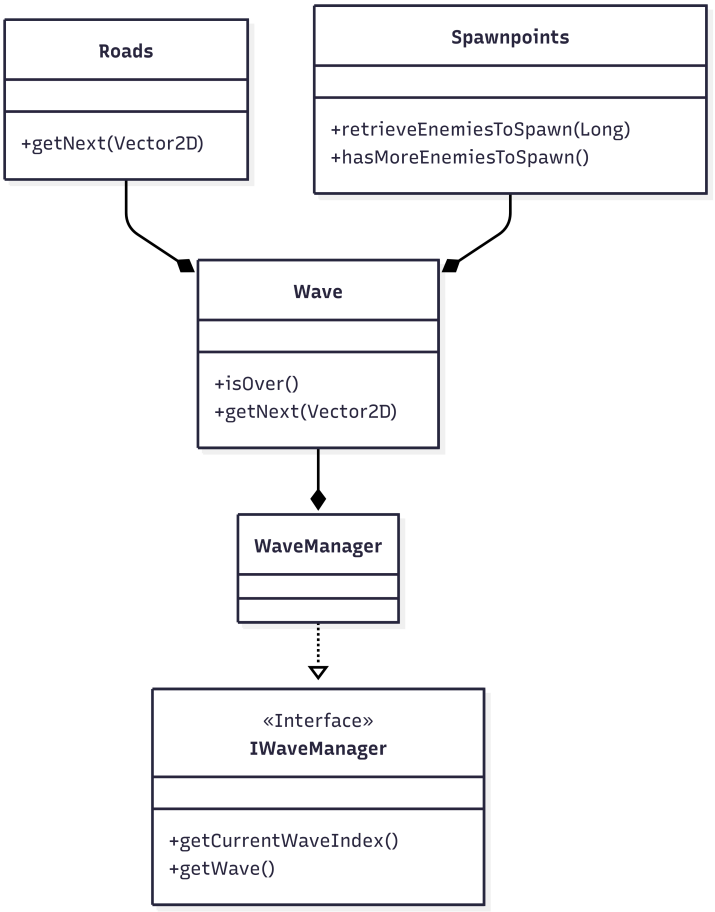

Figura 2.14: schema UML delle waves.

Soluzione Ho modellato ciò raggruppando le classi che garantiscono queste due funzionalità, *Roads* e *Spawnpoints*, dentro una classe *Wave*, che contiene tutto ciò che dipende dalla wave. A loro volta le *Wave* sono contenute dentro un manager, che si occupa di gestirne la progressione, insieme ad altre funzionalità come notificare il game over alla classe preposta in caso di vittoria.

Statistiche

Problema Raccogliere alcune semplici statistiche del gioco, come le wave completate o il danno totale inflitto dalle torri ai nemici, senza sporcare il codice delle classe già esistenti, e con libertà in merito a future aggiunte di altre statistiche.

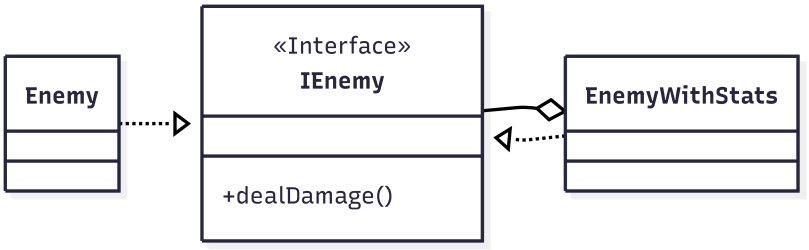

Figura 2.15: schema UML esempio per i decoratori delle statistiche.

Soluzione Per affrontare questa problematica ho deciso di applicare il pattern Decorator alle classi da cui raccolgo delle statistiche.

Quindi ad esempio per il nemico si ha un interfaccia IEnemy che viene implementata sia dalla classe base, Enemy, che dal decoratore EnemyWithStats. Racchiudendo Enemy dentro quest'ultimo è facile intercettare il danno che i nemici ricevono, e salvarlo nell'apposita classe Statistics, che funge da contenitore per tutte le statistiche raccolte.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

I test automatizzati sono stati fatti utilizzando il framework JUnit per le asserzioni e la libreria Mockito per la creazione di oggetti mock, nel caso classi avessero dipendenze.

Enemy

Per quanto riguarda la parte dei nemici vengono testati: il movimento con e senza rallentamento, il metodo *dealDamage* e il comportamento sotto effetto di *SlowEffect* e *BurnEffect*.

SpatialHashGrid

Per la struttura dati *SpatialHashGrid* sono stati effettuati i test sulle varie operazioni offerte: add, remove e update. Questi test vengono fatti generando randomicamente una quantità di nemici e, ad ogni operazione, effettuare un test d'integrità rispetto ad una normale lista.

Le operazioni randomiche sono naturalmente seedate in modo da poter riprodurre un possibile test fallito.

Tower

Delle torri vengono testate: le funzionalità per il cambiamento del tipo di torre, il fatto che il fire rate sia effettivamente quello nominale e l'aggiunta effettiva del proiettile alla pool in seguito ad uno sparo.

Roads

Il testing della classe *Roads* è stato fatto verificando la correttezza del funzionamento del grafo dei percorsi, testando anche possibili corner case.

Spawnpoints

Il testing della classe *Spawnpoints* si è concentrato sul controllo della nascita dei nemici, testando che corrispondano al tipo e al tempo prestabiliti.

EventDispatcher

Il testing della classe `EventDispatcher` si è concentrato sulla verifica del corretto funzionamento del sistema di gestione degli eventi, con particolare attenzione a: Registrazione e deregistrazione dinamica dei listener; Dispatching selettivo basato sul tipo di evento; Gestione di gerarchie di eventi (eventi base e sottotipi); Validazione dei listener (controllo firme dei metodi); Comportamento in edge case (eventi null, listener null).

3.2 Note di sviluppo

3.2.1 Tommaso Bandini

Utilizzo di `lambda`

Utilizzate in vari punti. Un esempio è `https://github.com/TommyT0mmY/OOP24-Emberline/blob/7b389c4efcf2454de23ca42e67d35aa365a5a332/src/main/java/dev/emberline/core/graphics/SpriteLoader.java#L38`

Progettazione con `generici`

Permalink: `https://github.com/TommyT0mmY/OOP24-Emberline/blob/7b389c4efcf2454de23ca42e67d35aa365a5a332/src/main/java/dev/emberline/core/graphics/SpriteFactoryRegistry.java`

Utilizzo della libreria `JavaFX`

Utilizzate in vari punti. Un esempio è `https://github.com/TommyT0mmY/OOP24-Emberline/blob/f08baf9acac0643c78161844b7ae8dc7988d0456/src/main/java/dev/emberline/core/sounds/AudioController.java`

3.2.2 Claudio Lodi

Utilizzo di `Stream`

Permalink: `https://github.com/TommyT0mmY/OOP24-Emberline/blob/7b389c4efcf2454de23ca42e67d35aa365a5a332/src/main/java/dev/emberline/game/world/entities/projectiles/projectile/ProjectileUpdateComponent.java#L174-L177`

Utilizzo di `lambda`

Utilizzate in vari punti. Un esempio è `https://github.com/TommyT0mmY/OOP24-Emberline/blob/7b389c4efcf2454de23ca42e67d35aa365a5a332/src/main/java/dev/emberline/game/world/entities/enemies/EnemiesFactory.java#L33-L34`

Utilizzo di `Function`

Permalink: `https://github.com/TommyT0mmY/OOP24-Emberline/blob/7b389c4efcf2454de23ca42e67d35aa365a5a332/src/main/java/dev/emberline/game/world/entities/projectiles/projectile/ProjectileUpdateComponent.java#L97`

Utilizzo della libreria `Mockito`

Utilizzata in vari punti. Un esempio è `https://github.com/TommyT0mmY/OOP24-Emberline/blob/7b389c4efcf2454de23ca42e67d35aa365a5a332/src/test/java/dev/emberline/game/world/entities/enemies/enemy/EnemyTest.java#L50-L58`

Utilizzo della libreria `JavaFX`

Utilizzata in vari punti. Un esempio è `https://github.com/TommyT0mmY/OOP24-Emberline/blob/7b389c4efcf2454de23ca42e67d35aa365a5a332/src/main/java/dev/emberline/game/world/entities/enemies/enemy/EnemyRenderComponent.java#L65-L70`

3.2.3 Tommaso Goni

Utilizzo di annotazioni

Permalink: `https://github.com/TommyT0mmY/OOP24-Emberline/blob/7b389c4efcf2454de23ca42e67d35aa365a5a332/src/main/java/dev/emberline/core/event/EventHandler.java`

Progettazione con generics con vincoli incrociati per type-safety

Permalink: https://github.com/TommyT0mmY/OOP24-Emberline/blob/7b389c4efcf2454de23ca42e67d35aa365a5a332/src/main/java/dev/emberline/game/model/UpgradableInfo.java

Utilizzo di generic methods con bounded type parameters annidati

Permalink: https://github.com/TommyT0mmY/OOP24-Emberline/blob/7b389c4efcf2454de23ca42e67d35aa365a5a332/src/main/java/dev/emberline/gui/towerdialog/TowerDialogLayer.java#L260-L261

Utilizzo di Reflections

Utilizzate all'interno del sistema di event dispatching; un esempio è: https://github.com/TommyT0mmY/OOP24-Emberline/blob/7b389c4efcf2454de23ca42e67d35aa365a5a332/src/main/java/dev/emberline/core/event/EventDispatcher.java#L153-L161

Utilizzo di Lambda

Sono state utilizzate in gran parte del progetto; un esempio è: https://github.com/TommyT0mmY/OOP24-Emberline/blob/7b389c4efcf2454de23ca42e67d35aa365a5a332/src/main/java/dev/emberline/game/world/buildings/tower/TowerRenderComponent.java#L86-L99

Utilizzo di Stream

Permalink: https://github.com/TommyT0mmY/OOP24-Emberline/blob/7b389c4efcf2454de23ca42e67d35aa365a5a332/src/main/java/dev/emberline/gui/towerdialog/stats/TowerStatsViewsBuilder.java#L184-L209

Utilizzo della libreria Jackson

Utilizzata per il caricamento delle configurazioni da risorse `.json`; un esempio è: https://github.com/TommyT0mmY/OOP24-Emberline/blob/7b389c4efcf2454de23ca42e67d35aa365a5a332/src/main/java/dev/emberline/core/config/ConfigLoader.java

Utilizzo della libreria AssertJ

Permalink: `https://github.com/TommyT0mmY/OOP24-Emberline/blob/7b389c4efcf2454de23ca42e67d35aa365a5a332/src/test/java/dev/emberline/core/event/EventDispatcherTest.java#L113-L120`

3.2.4 Luigi Linari

Uso di `Optional`

Permalink: `https://github.com/TommyT0mmY/OOP24-Emberline/blob/6d9ab78dbed71f0ce7f47690fe948c956a6f161e/src/main/java/dev/emberline/game/world/roads/Roads.java#L62`

Uso di `Stream`

Permalink: `https://github.com/TommyT0mmY/OOP24-Emberline/blob/6d9ab78dbed71f0ce7f47690fe948c956a6f161e/src/main/java/dev/emberline/game/world/buildings/TowersManager.java#L66-L73`

Uso di `Vector2D` dalla libreria Commons Geometry

Permalink: `https://github.com/TommyT0mmY/OOP24-Emberline/blob/6d9ab78dbed71f0ce7f47690fe948c956a6f161e/src/main/java/dev/emberline/utility/Coordinate2D.java#L10`

Uso di una libreria esterna per identificare il percorso di salvataggio della partita in base al sistema operativo

Permalink: `https://github.com/TommyT0mmY/OOP24-Emberline/blob/6d9ab78dbed71f0ce7f47690fe948c956a6f161e/src/main/java/dev/emberline/game/serialization/Serializer.java#L20-L22`

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Tommaso Bandini

Lo sviluppo del progetto ha richiesto un impegno costante, sia sul piano tecnico che organizzativo. Tra le maggiori difficoltà c'è sicuramente: conciliare il progetto con gli altri impegni universitari, gestire la voglia di aggiungere nuove feature sopratutto quando non necessarie ed una stima del tempo richiesto per ogni nuova feature o cambiamento. Inoltre essendo il mio primo progetto di questa portata è stata molto complicata la progettazione non essendo a conoscenza della difficoltà di implementazione di molti dettagli. A progetto concluso mi sento molto soddisfatto del risultato ma non totalmente del mio lavoro svolto, perchè a riguardare i sistemi di GUI penso che avrei potuto avere un organizzazione migliore che potesse astrarle meglio, mentre sono molto contento della riuscita dello SpriteLoader perchè lo vedo assolutamente riutilizzabile in altri contesti. Il mio ruolo nel progetto è relativo alle GUI, al caricamento degli Sprite, gestione audio ed eventi principali dedicati al comportamento del giocatore.

L'esperienza è stata estremamente formativa chiaramente per tutte le riflessioni sopra citate, inoltre ho avuto modo di imparare e poi utilizzare software di terze parti per la realizzazione di quasi tutte le pixel art (statiche e animate) presenti nel gioco.

L'esperienza di progettare un videogioco dall'inizio alla fine mi ha entusiasmato moltissimo e sarei molto contento di poter continuare lo sviluppo fino a quanto immaginato inizialmente, anche se ovviamente sarebbe da rivedere la gestione del tempo.

4.1.2 Claudio Lodi

Nel complesso mi ritengo abbastanza soddisfatto di come è riuscito il progetto, considerando anche che è la prima volta che mi trovo a lavorare in gruppo. Io principalmente mi sono occupato della parte relativa ai nemici e ai proiettili ma ho anche strutturato come sarebbero state contenute le varie entità di gioco all'interno del mondo. La parte relativa all'engine l'abbiamo discussa assieme concentrandoci in particolar modo sul come interagire con la libreria grafica JavaFX.

Mi trovo però a dover fare una critica sul come è uscita la classe del nemico, in particolare la gestione degli stati e le conseguenti interazioni con le entità esterne. Questo è dovuto dal fatto che in fase di progettazione non sono stati ben definiti gli attori e come avrebbero interagito, così in fase di sviluppo mi sono trovato a fare degli accorgimenti poco eleganti.

Come esperienza penso sia molto formativa in quanto ti spinge fuori dalla comfort zone, spesso infatti mi sono trovato in situazioni completamente nuove e mi ha dato anche un'idea di come potrebbe essere un contesto lavorativo.

L'idea del progetto secondo me era molto interessante, solo che molti aspetti non li abbiamo approfonditi. Se in futuro ci sarà interesse a estendere quanto già realizzato, ne sarei entusiasta, soprattutto considerando l'esperienza acquisita.

4.1.3 Tommaso Goni

Mi ritengo complessivamente soddisfatto del mio contributo al progetto, nonostante le inevitabili sfide tecniche e organizzative. Mi sono concentrato principalmente sulla progettazione delle fondamenta del gioco (Game Loop, Rendering, Event Dispatching, ecc.), sulla progettazione del sistema delle statistiche delle torri e su alcuni aspetti di GUI.

Riflettendo sul lavoro svolto, riconosco che alcuni aspetti dell'integrazione tra GUI e logica di business avrebbero potuto beneficiare di una maggiore astrazione, un'area su cui continuerò a lavorare per il futuro. L'esperienza è stata estremamente formativa, specialmente nella gestione dell'architettura e nella coordinazione con il team.

L'idea del progetto è molto stimolante, e sarei interessato a continuarne lo sviluppo per apportare migliorie alla struttura già presente visto che numerose volte sono state prese scorciatoie progettuali a causa del vincolo temporale della consegna.

4.1.4 Luigi Linari

Nonostante le molte difficoltà incontrate, e il non essere riuscito a fare tanto quanto inizialmente mi ero immaginato, sono abbastanza soddisfatto del risultato ottenuto dalla mia parte di lavoro. Infatti la nascita e l'indirizzamento dei nemici attraverso l'uso delle classi *Spawnpoints* e *Roads* è riuscito molto bene, così come altre cose quali la gestione delle *Wave*.

D'altro canto penso che mi sarei dovuto concentrare di più sull'architettura generale del gioco, che invece hanno elaborato altri membri del gruppo, perchè la trovo altrettanto interessante e mi sembra una mancanza significativa.

Anche se sarebbe bello migliorare il gioco, aggiungendo nuovi livelli, apportando bilanciamenti più equi, ma anche adottando soluzioni migliori per tutti quei problemi che abbiamo risolto in modo non ottimale, non penso che avremo modo di portarlo avanti. Tuttavia è stata comunque un'esperienza molto costruttiva e sono contento di averla potuta affrontare.

Appendice A

Guida utente

Il gioco si sviluppa intorno al principio del punta e clicca, quindi non è necessaria la tastiera per giocare ma solo il mouse. Per posizionare le torri è sufficiente cliccare sopra ai cartelli e confermare l'acquisto dall'apposita finestra. Similmente cliccando sopra alle torri si entra nel loro menu, e si esce cliccando al di fuori di esso. Sempre un click basta per cambiare la **priorità di mira** o acquistare/cancellare i **potenziamenti**. I cambiamenti sono mostrati in modo più preciso direttamente nella GUI delle torri, inoltre facendo l'hover su un pulsante di upgrade/reset è possibile visualizzare i cambiamenti in tempo reale prima di confermare il cambiamento.