Wild Encounter

Andrea Maria Castronovo
Leonardo Mengozzi
Lorenzo Mazzini
Kleo Rama

15 febbraio 2026

Indice

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Wild Encounter è un gioco 2D bullet-heaven[1] ispirato a Vampire Survivors ma ambientato nel mondo Pokémon. Il giocatore si trova sempre al centro dello schermo e deve sopravvivere alle orde di Pokémon selvatici che arrivano da ogni lato. Man mano che avanza il tempo i nemici diventano sempre più forti e numerosi, aumentando così il grado di sfida. Affrontando i nemici si guadagna esperienza che permette di salire di livello e di sbloccare nuove MT[2]. Ogni livello appare una scelta di 3 MT che serviranno per fronteggiare i Pokémon selvatici sempre più potenti e numerosi. Ogni MT può salire di livello in base a quante volte è scelta al level-up, fino ad arrivare ad un livello massimo.

1.1.1 Requisiti funzionali

- Muovere il proprio personaggio all'interno della mappa infinita;
- Presenza di Pokémon che cercano di avvicinarsi e danneggiare il giocatore;
- Combattere i Pokémon
- Aumentare il proprio livello guadagnando esperienza sconfiggento i Pokémon selvatici;
- Scegliere tra diversi tipi di MT ad ogni livello;

[1] Categoria di giochi in cui i nemici convergono sul giocatore
[2] Macchine Techine, ovvero mosse usabili dai Pokémon

- Possibilità di combattere i Pokémon selvatici tramite MT trovate salendo di livello.

1.1.2 Requisiti non funzionali

- Progredendo nel gioco saranno presenti a schermo quantità di Pokémon elevate, le prestazioni dovranno però restare accettabili;
- Possibilità di tenere conto di quanti Pokémon sono stati visti in ogni partita in un Pokédex.

1.2 Modello del Dominio

La *Mappa* di gioco conterrà vari *Oggetti* dotati di collisioni, come le varie *Entità* animate (il *Giocatore* ed i *Nemici*) o oggetti inanimati come i *Collezionabili*. Durante la partita appariranno casualmente diversi tipi di nemici che arriveranno dai bordi dello schermo, inseguendo il giocatore cercando di danneggiarlo. Sia il giocatore che i nemici posseggono delle *Armi*, ciascuna dotata di statistiche diverse, come ad esempio portata, danno e velocità di attacco, con cui attaccheranno passivamente durante tutta la partita. Per garantire una corretta progressione del gioco, i nemici, una volta eliminati, rilasceranno punti esperienza, monete o bacche per recuperare vita che possono essere collezionati dal giocatore (Collezionabili). Accumulando punti esperienza il giocatore sale di livello, il quale tornerà a 0 ad ogni partita. Salire di livello permette al giocatore di scegliere delle armi che lo aiuteranno ad affrontare i nemici sempre più forti e numerosi. Se il giocatore sceglie un'arma che possiede già potrà potenziarla aumentandone le statistiche fino ad un certo massimo. La Figura 1.1 mostra le relazioni tra gli oggetti descritti.

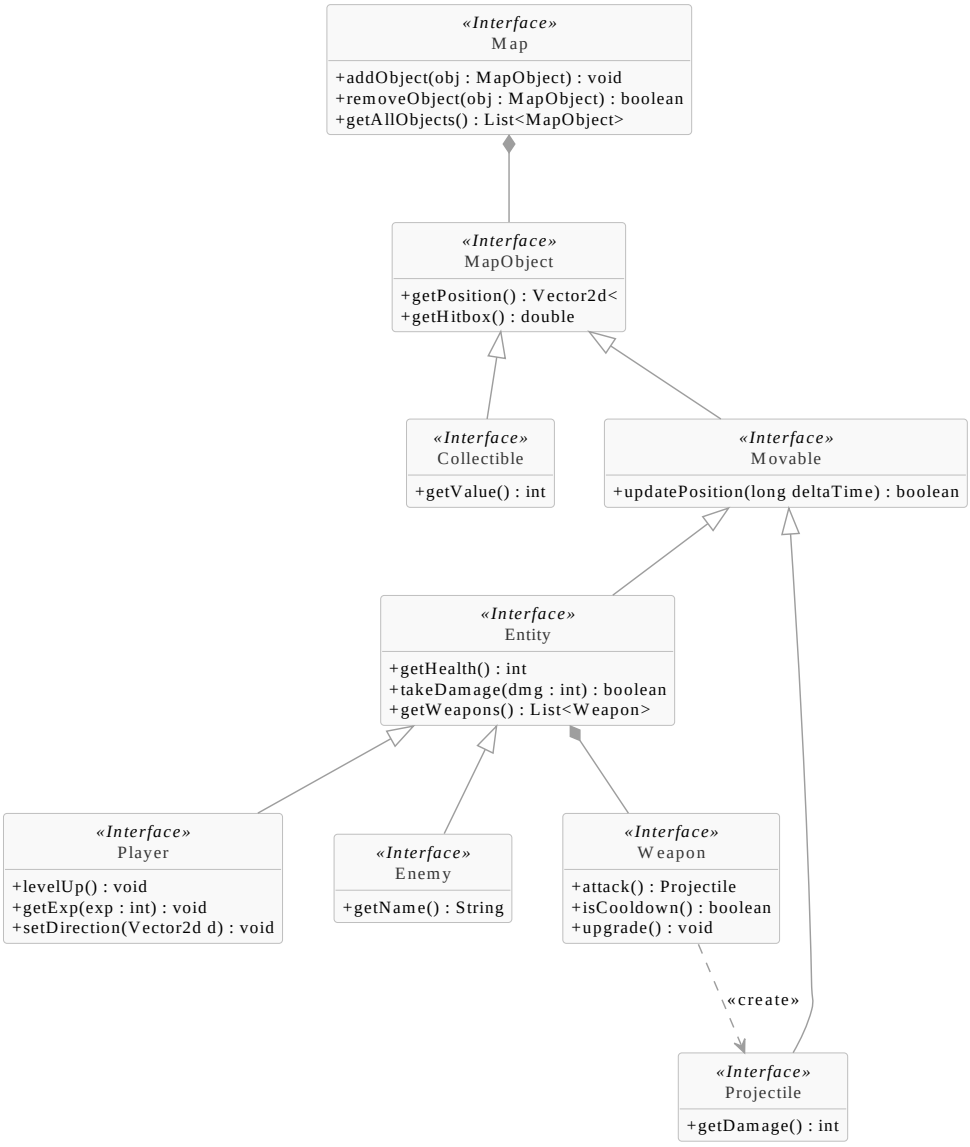

Figura 1.1: Schema UML dell'analisi del dominio, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

L'architettura di Wild Encounter segue il pattern MVC. Il motore di gioco (ossia il Controller) dovrà essere in grado di registrare gli input del giocatore e di associarli alle azioni delle varie entità presenti nel gioco che costituiscono il Model, come ad esempio il movimento delle entità Movable o l'interazione con i Collectible. In quanto tale il motore di gioco si limiterà ad orchestrare i vari movimenti o eventi di gioco, gestiti poi singolarmente da ogni Entità del model. Il motore di gioco dovrà gestire il loop principale che, ad ogni iterazione, dirà al model come aggiornare i valori in base ai comandi impartiti. Nel gioco sono presenti dei nemici che dovranno avere dei comportamenti autonomi, questi saranno calcolati tramite degli algoritmi all'interno del sistema di movimento dei nemici nel model. Sarà presente una fase di Menù statica, che non interesserà il game loop (gestito all'interno del Motore di Gioco principale) in quanto non vi sono entità animate da gestire; durante questa fase sarà possibile avviare la partita, controllare il proprio Pokédex e, in futuro, acquistare potenziamenti permanenti tramite le monete guadagnate nelle varie partite. La View sarà rimpiazzabile facilmente implementando tutto ciò che è richiesto dall'app per funzionare, così facendo sarebbe possibile in futuro anche l'implementazione dello stesso gioco per altre piattaforme come telefoni o schermi touchscreen.

Nella Figura 2.1 è illustrata ed evidenziata l'architettura della fase di gioco appena descritta.

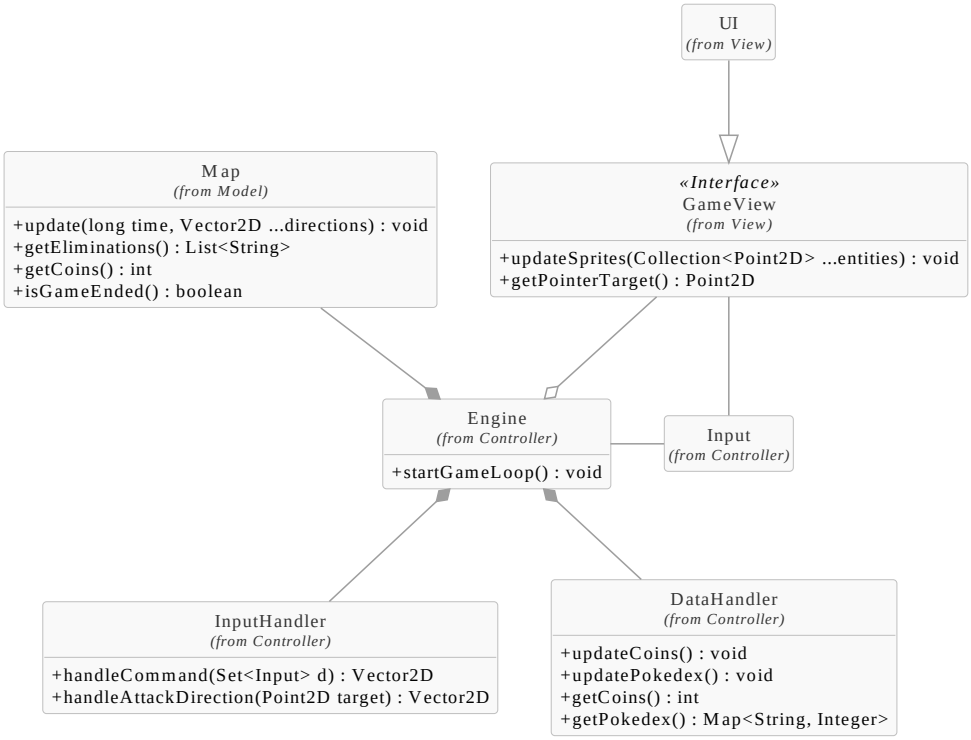

Figura 2.1: Schema UML che evidenzia la struttura MVC del progetto

2.2 Design Dettagliato

2.2.1 Andrea Maria Castronovo - Gestione mappa ed entità

Oggetti di gioco

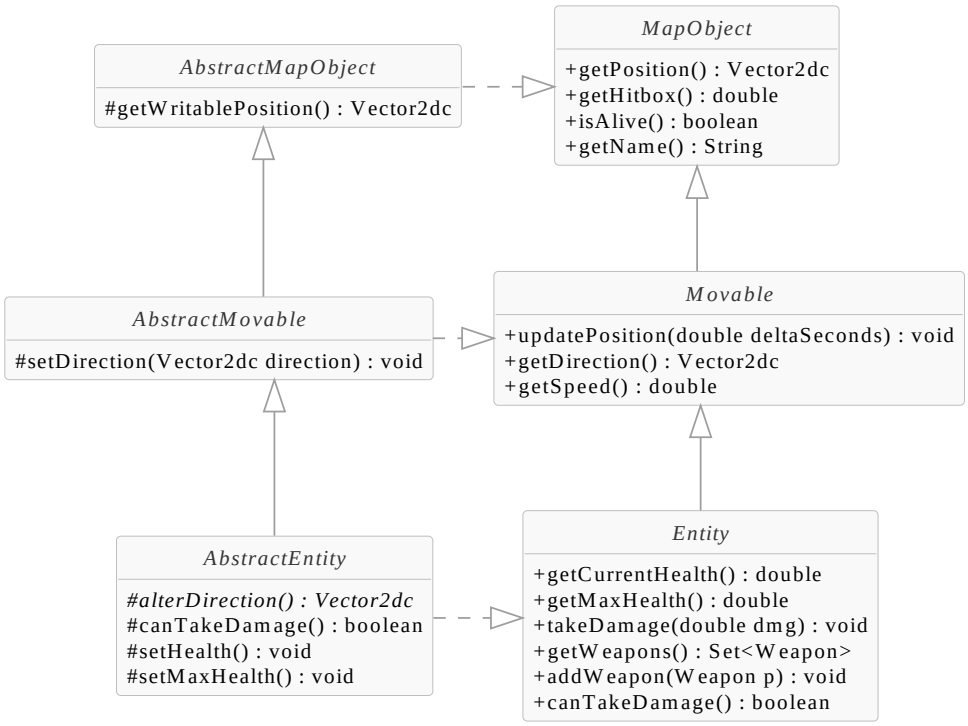

Figura 2.2: Oggetti di gioco

Problema: Devono esitere degli oggetti di gioco che possono o non possono muoversi.
Soluzione: Per gestire correttamente tutti gli oggetti del gioco con una singola logica è stata creata una gerarchia di oggetti astratti che vanno a specializzarsi man mano. Nella gerarchia è presente un metodo astratto che consente di mantenere un comportamento controllato della classe, ma anche adattabile alle varie esigienze delle classi implementanti; ciò è reso possibile applicando il *Template Pattern* sul metodo *alterDirection* di AbstractEntity.

La mappa

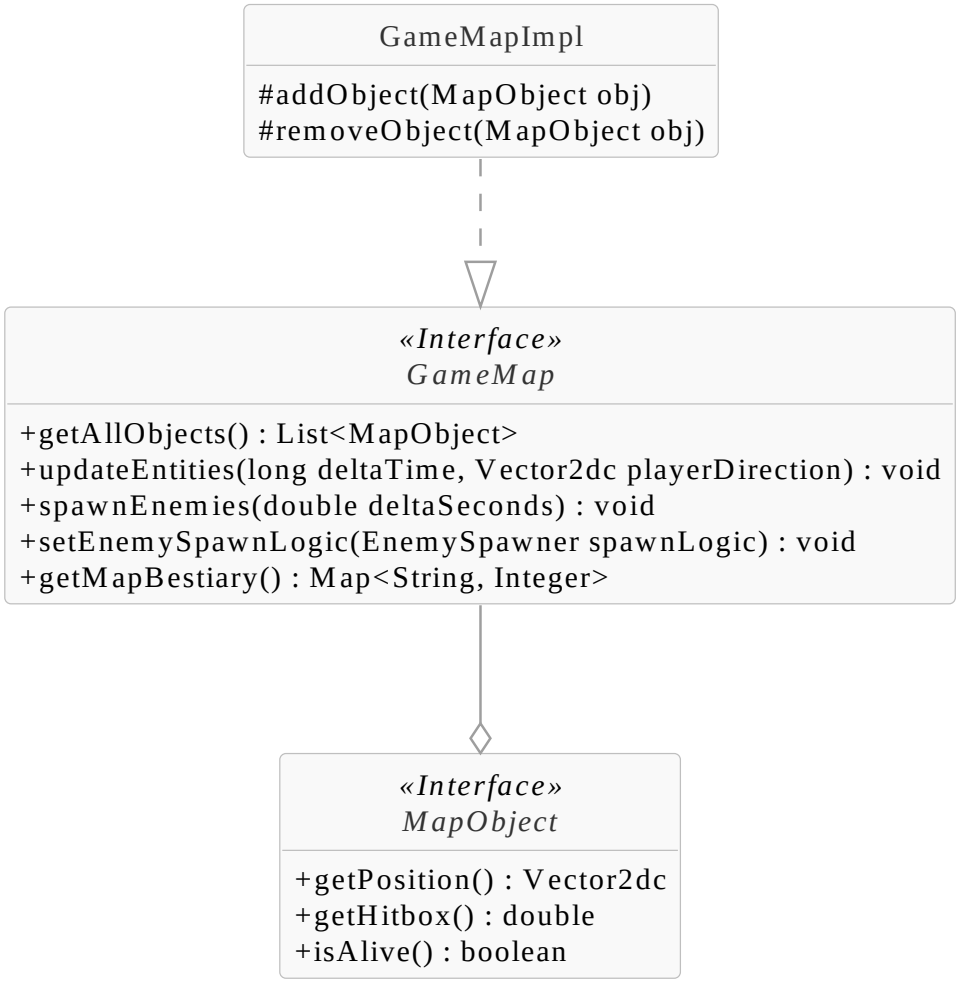

Figura 2.3: Mappa di gioco

Problema: Deve esserci una mappa su cui gli oggetti devono coesistere ed interagire.
Soluzione: La *GameMap* è la principale responsabile della gestione degli oggetti di gioco, è lei che gestirà i movimenti, le collisioni dei proiettili e si occuperà di spawnare i nemici. La mappa è pensata in relazione one-to-many con tanti *MapObject*, in modo che possa contenerli e gestirli tutti. Lo spawning dei nemici è gestito assieme al movimento delle vari entità, ma la logica secondo cui questi spawnano è stata scorporata grazie all'interfaccia *EnemySpawner* fornita da Leonardo Mengozzi. Ciò consente in futuro di

applicare logiche di spawn diverse alle varie mappe, in modo da creare nuove modalità di gioco o anche boss battle seguendo il principio OCP.

Gestione della fase di gioco

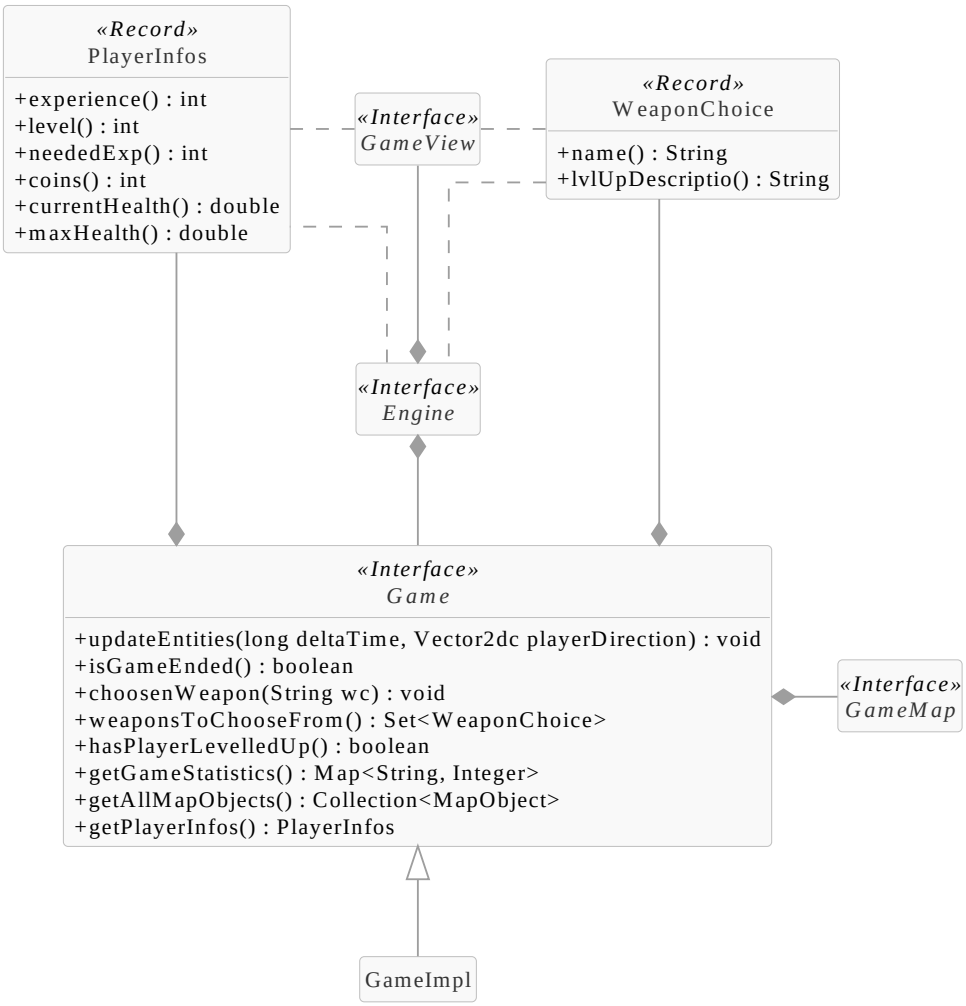

Figura 2.4: Applicazione del pattern *Facade* per proteggere la *GameMap*.

Problema: Il gioco deve essere comandato da un controller, deve quindi esporre determinate informazioni senza però alterare la sicurezza e l'encapsulamento dei dati.
Soluzione: È stata introdotta l'interfaccia *Game*. Il suo scopo è quello di fornire all'esterno ciò che serve per interagire con il gioco; deve, ad esempio, occuparsi di ricevere informazioni riguardo al personaggio scelto e di

generarlo per la mappa, deve inoltre fornire le armi che saranno sbloccate durante la progressione del gioco, ma anche comunicare alla mappa quando è il momento di aggiornare le entità. Tale flusso di informazioni è stato gestito applicando il pattern *Facade*: L'interfaccia *Game* è un Facade per *GameMap*, in quanto si occupa di impacchettare e gestire i dati nei record, così facendo è possibile gestire gli oggetti della mappa in modo sicuro.

2.2.2 Lorenzo Mazzini - Gestione armi e proiettili

Introduzione: All'interno del gioco dovranno essere disponibili varie tipologie di armi, ciascuna capace di generare diverse tipologie di attacchi con i quali sarà possibile colpire i nemici. Tali attacchi dovranno variare non solo dal punto di vista delle statistiche (quali danno, velocità...), ma anche dal punto di vista del loro comportamento relativo al movimento e alla generazione.

Armi e proiettili

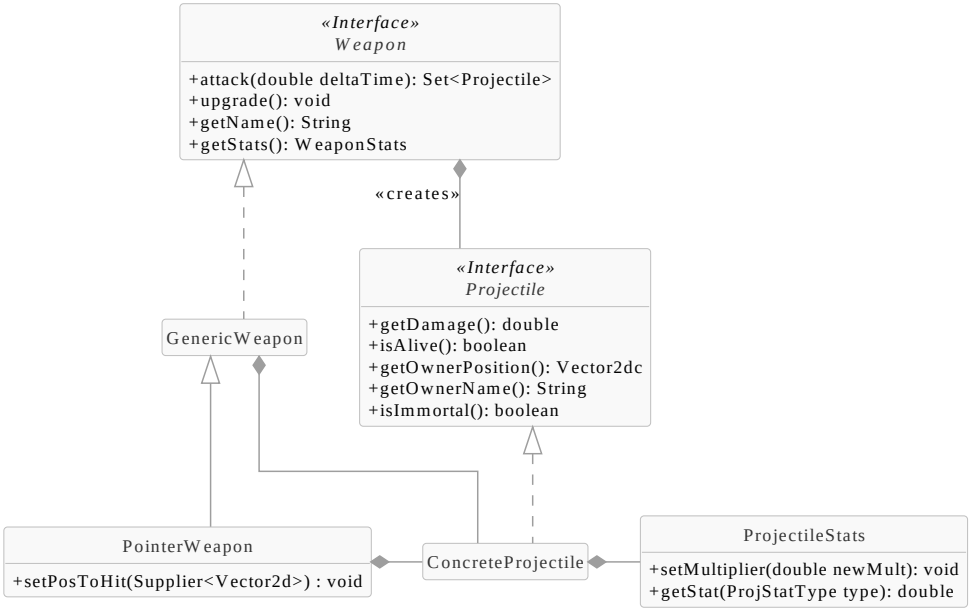

Figura 2.5: Schema UML rappresentante il funzionamento delle Weapon come factory di Projectile

Problema 1: ogni arma dovrà essere differenziabile, sia dal punto di vista di come attacca, sia dal punto di vista di quando attacca.

Soluzione: il concetto di arma è stato generalizzato in modo che ogni implementazione differisca sulla base di come l'arma si deve comportare; nello specifico è stata creata una classe *GenericWeapon*, la quale ottiene le informazioni relative al suo comportamento nel momento in cui viene istanziata: in questo modo è ampiamente permessa la personalizzazione dei vari attacchi. Ogni *Weapon* è una *factory* di *Projectiles*; la tipologia di proiettile che l'arma sarà in grado di generare è specificato alla costruzione dell'arma stessa tramite le sue statistiche, nella forma di una classe *ProjectileStats*. Ogni proiettile generato dall'arma otterrà tali statistiche, le quali definiscono sia come il proiettile è fatto concretamente, ma anche come esso deve muoversi. In questo modo si scorpora la dipendenza tra arma e proiettile dopo che il proiettile è stato generato. Oltre al tipo di proiettile, l'arma dovrà ottenere anche altre informazioni, come il nome dell'arma, il suo tempo di ricarica e il numero di proiettili sparati alla volta. Ultima, fondamentale, componente di cui l'arma ha bisogno è il metodo di generazione dei proiettili: ogni arma genererà i proiettili per mezzo di una classe *AttackContext*, la quale contiene tutte le informazioni utili al proiettile per esitere (posizione, direzione, velocità e posizione da seguire).
Armi così definite vengono specializzate per mezzo del *design pattern* Abstract Factory: ogni Factory produce una specifica tipologia di arma, di cui cambiano solamente i parametri riguardanti le statistiche.

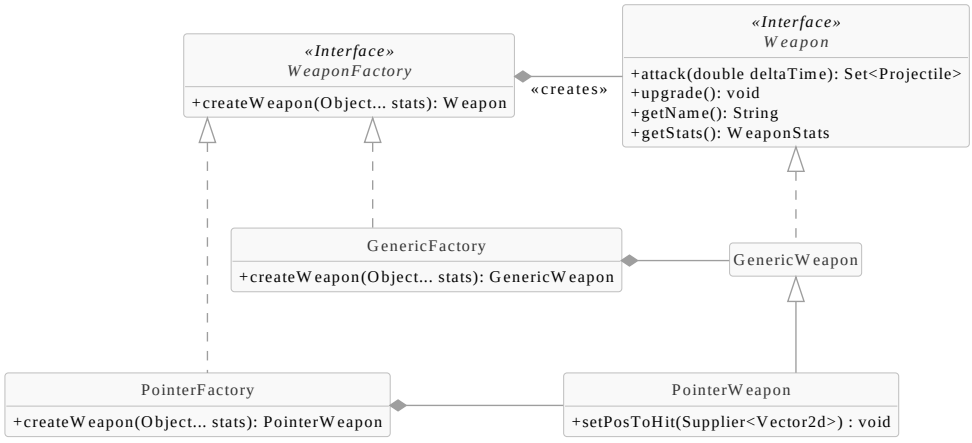

Figura 2.6: Schema UML rappresentante il funzionamento delle WeaponFactory. In questo caso, GenericFactory e PointerFactory sono due esempi di implementazione

Problema 2: esistono delle armi che devono seguire la posizione del mouse al fine di selezionare il punto in cui i proiettili andranno.

Soluzione: è stata creata una sottoclasse di *GenericWeapon*, la quale è *PointerWeapon*. Tale tipologia di arma è in grado di cambiare la posizione a cui "si sta mirando", di fatto scegliendo dove i proiettili finiranno. All'interno del controller, ad ogni *PointerWeapon* verrà associata la posizione "bersaglio".

Salvataggio dei dati

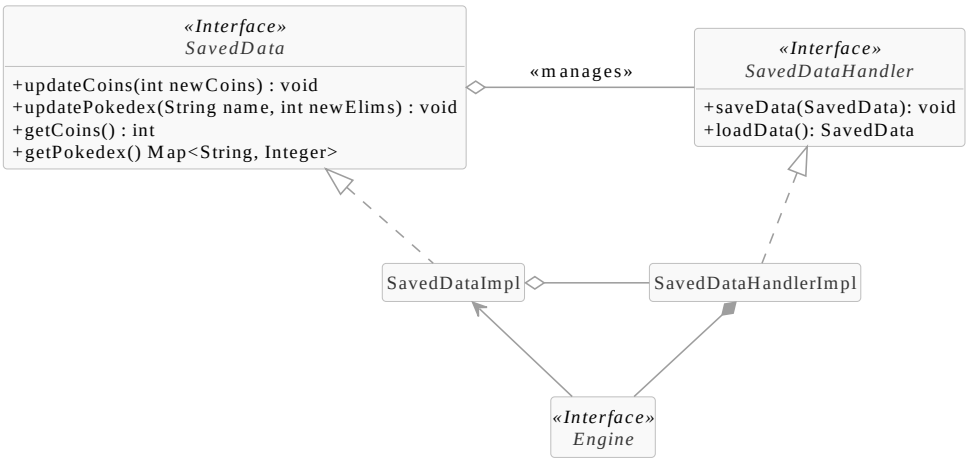

Figura 2.7: Schema UML rappresentante il modulo di salvataggio dati. Si noti come Engine, ossia il controller, possieda sia i dati salvati sia la componente per caricare/salvare i dati stessi.

Problema: I dati relativi ai *Pokemon* eliminati e le monete ottenute durante le varie partite devono essere salvati anche per le future partite.
Soluzione: Sono state realizzate le classi *SavedData* e *SavedDataHandler* al fine di gestire i dati salvati. In particolare *SavedData* mantiene il registro dei Pokemon eliminati in tutte le partite fatte, oltre al conteggio delle monete; *SavedDataHandler* ha il compito di salvare i dati quando una partita termina o quando il gioco viene chiuso, e ricaricarli nel gioco quando esso viene aperto.

Rendering di Sprite

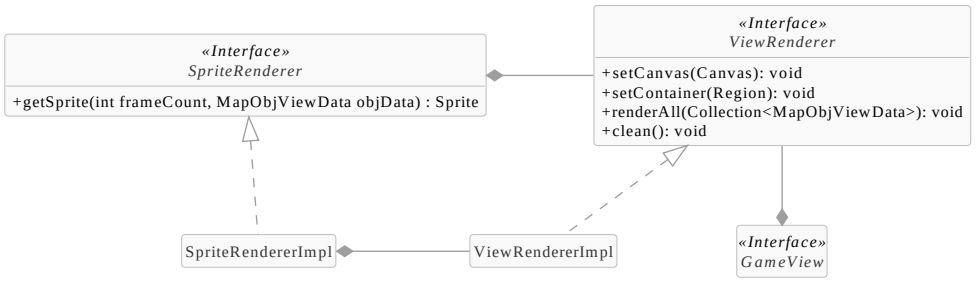

Figura 2.8: Schema UML rappresentante il modulo di caricamento/rendering di Sprite e immagini.

Problema: ogni entità di gioco, possiede un aspetto diverso, che necessita di essere visualizzato correttamente.
Soluzione: ogni *MapObject*, quando deve essere processato all'interno della view, viene tradotto in un record *MapObjViewData*, il quale contiene l'essenzialità del *MapObject* di partenza: ID, posizione, direzione e grandezza della hitbox. Internamente alla view è stata creata una classe *SpriteRenderer*, la quale, dato un *MapObjectViewData*, compone un oggetto di tipo *Sprite*, il quale contiene i dati necessari per mostrare l'immagine corrispondente a schermo. Tale immagine viene animata anche sulla base della direzione seguita dall'oggetto. Inoltre, la classe processa anche lo sfondo di gioco, permettendo un effetto a scorrimento.

Caricamento statistiche da file

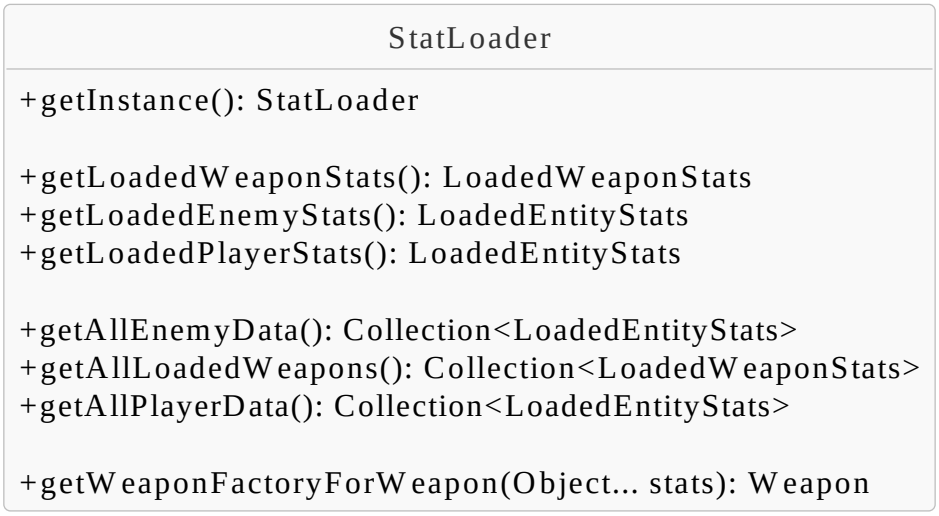

Figura 2.9: Schema UML che rappresenta il Singleton StatLoader.

Problema: vogliamo che le statistiche delle entità di gioco siano separate dal codice del gioco stesso, permettendo così la modifica di entità già esistenti o la creazione di nuove entità in modo semplice.
Soluzione: inizialmente, la gestione delle statistiche dei personaggi, dei nemici e delle armi era completamente interna al codice. Al fine di separare le statistiche dalle implementazioni è stata utilizzata una classe *StatLoader*, che sfrutta il *design pattern* Singleton. Tale classe, carica da file in formato JSON le statistiche relative ad armi ed entità, memorizzandole nei record *LoadedWeaponStats* e *LoadedEntityStats*. Al momento della generazione di armi o entità, la classe può restituire questi record al fine di permettere la generazione tramite i dati letti da file. In questo modo è resa facile l'implementazione da parte di utenti esterni di nuove entità o nuove armi, specificandone semplicemente le statistiche (per le weapon va anche specificata la Factory con la quale vengono prodotte).

2.2.3 Leonardo Mengozzi - Gestione nemici e Controller

Abstract Factory di enemy

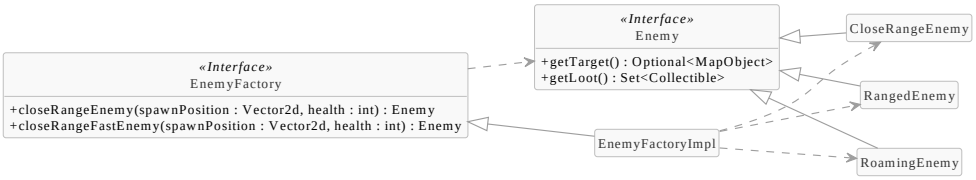

Figura 2.10: UML rappresenta Abstract factory, con metodi d'esempio, per gli enemys

Problema: Wild Encounter ha diverse tipologie di nemici, la cui creazione richiede parametri simili e l'assegnamento delle armi deve essere fisso dopo la creazione delle istanze.

Soluzione: Per rendere facile e sicura la creazione delle istanze di nemici si è adottato un *Abstract factory* che espone un metodo con parametri necessari a ogni nemico per essere istanziato.

Applicazione template method

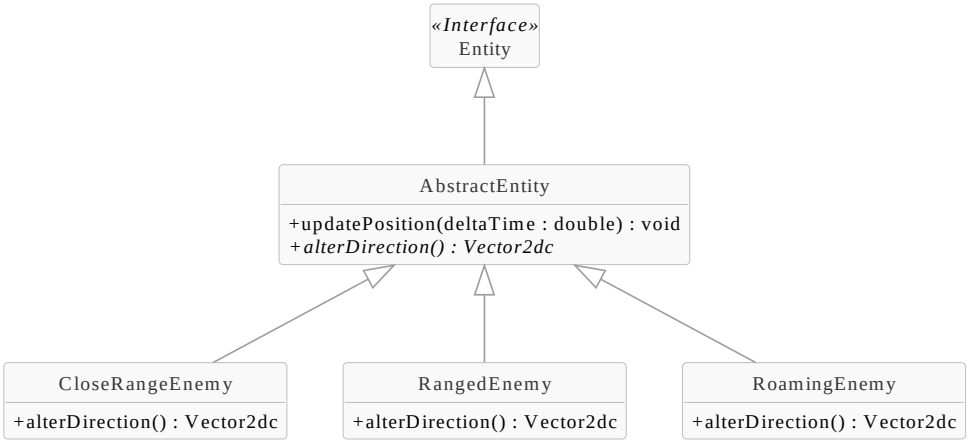

Figura 2.11: UML rappresenta Abstract factory per gli enemys

Problema: Le implementazioni dei nemici differiscono solamenete per la specializzazione della logica di movimento e se possono o meno subire danno.

Soluzione: Concretizzo i metodi astratti alterDirection e canTakeDamage, creati seguendo il design pattern *Template method* nella Abstract Entity class, per specificare il movimento specifico del nemico e per il Roaming enemy che può acquisire danno solo dopo un tempo specifico.

Aggiornamento della grafica

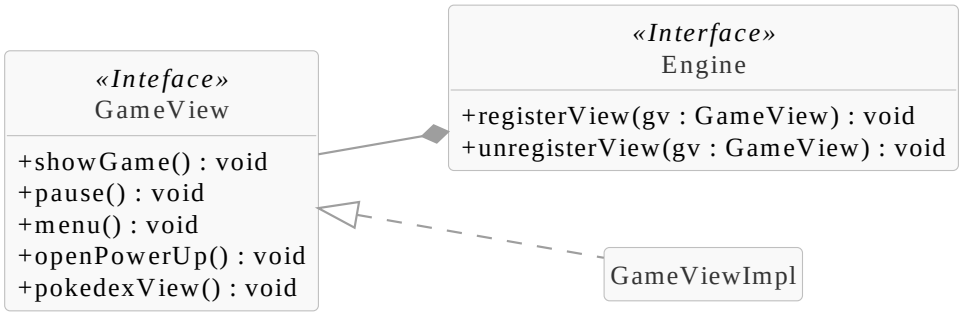

Figura 2.12: UML rappresenta osberver controller e view

Problema: Nel caso un'istanza dell'applicazione disponga di più GUI o CLI queste devono coerentemente rispondere alla unica logica.

Soluzione: Applico il design pattern *Observer* per notificare in modo coerente tutte le view registrate quando lo stato del gioco cambia. Seguendo l'architettura MVC, il controller (Engine) funge da Subject/Observable, mentre le view si registrano come Observer e vengono aggiornate dal controller durante il game loop. L'implementazione differisce dal modello standard per l'aggiunta della possibilità di disiscrivere un Observer dall'Observable e diversi metodi di notifica.

2.2.4 Kleo Rama - Gestione Player, Audio, Input movimento e UI

Il contributo si è focalizzato sulla definizione dell'entità principale di gioco, sulla gestione degli input utente per il movimento, sul sistema audio, sulle interfacce di overlay e sui collezionabili.

Gestione Input e Movimento (WASD)

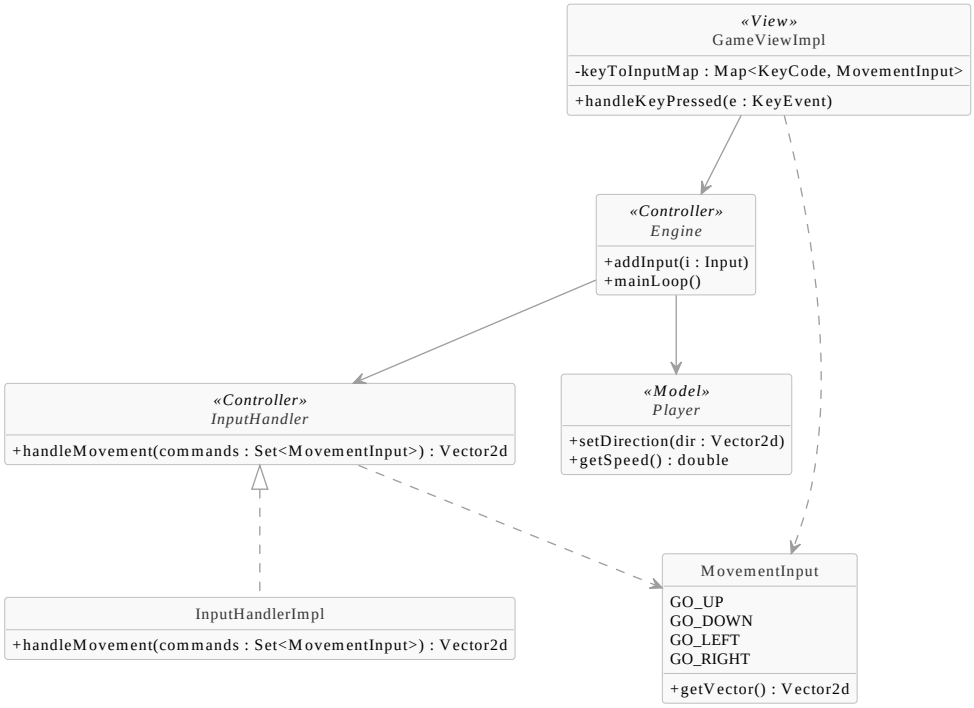

Figura 2.13: UML rappresenta la gestione del movimento

Problema: Era necessario implementare un sistema di controllo del personaggio che fosse reattivo e completamente disaccoppiato dalla libreria grafica (JavaFX), permettendo al *Controller* di gestire la logica di movimento senza conoscere i dettagli dell'evento tastiera.
Soluzione: È stata definita l'interfaccia `InputHandler` e la sua implementazione `InputHandlerImpl`. Il sistema utilizza un approccio basato su enumerazioni (`MovementInput`) per astrarre i comandi grezzi (tasti W, A, S, D).

- **View:** La classe `GameViewImpl` intercetta gli eventi `KeyEvent` di JavaFX e li traduce in comandi logici (es. `KeyCode.W` → `MovementInput.GO_UP`) che vengono inviati all'Engine.
- **Model:** Il vettore calcolato viene applicato all'entità `Player`, che aggiorna la sua posizione in base alla velocità corrente.

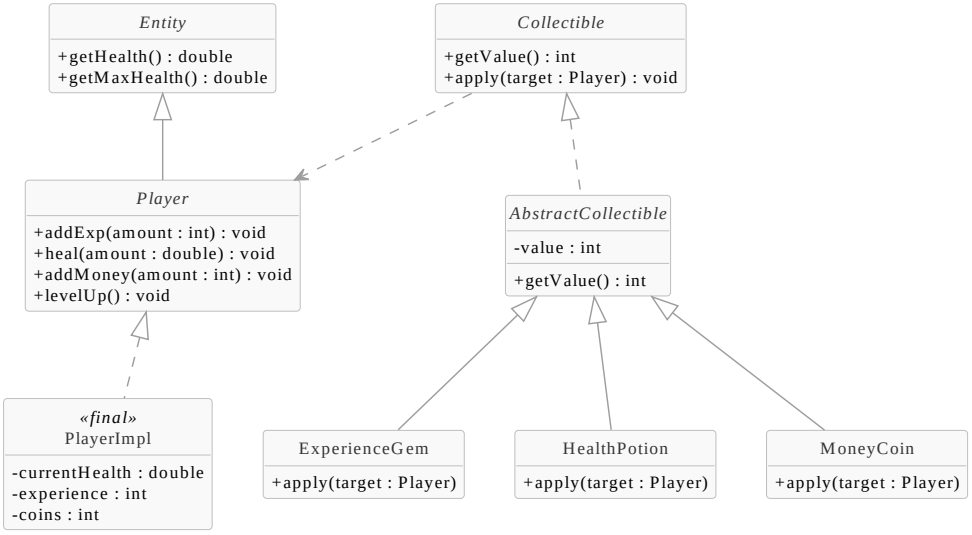

Figura 2.14: UML rappresenta il giocatore e i collezionabili

Modellazione dell'Entità Giocatore

Problema: Era necessario definire un'entità principale capace di evolvere nel tempo, gestendo statistiche dinamiche come l'esperienza e la salute, e che fosse robusta rispetto a modifiche esterne.
Soluzione: La classe `PlayerImpl` estende `AbstractEntity` e implementa l'interfaccia `Player`. Per garantire la stabilità del codice e prevenire estensioni non sicure la classe è stata dichiarata `final`. La logica di progressione è stata divisa in due fasi per permettere al Controller di gestire l'interazione con l'utente:

- `canLevelUp()`: Verifica semplicemente se l'esperienza accumulata supera la soglia necessaria.
- `levelUp()`: Effettua concretamente il passaggio di livello, incrementando le statistiche (HP massimi, velocità) e resettando l'esperienza.

È stata inoltre implementata una suite di test unitari (`PlayerTest`) che simula il ciclo di vita del giocatore, verificando casi limite come il guadagno multiplo di livelli in un singolo aggiornamento.

Sistema di Collezionabili (Collectibles)

Problema: Il gioco richiede oggetti interagibili che forniscano benefici immediati al giocatore (esperienza, cura, monete) una volta raccolti, e un sistema

flessibile per generarli alla morte dei nemici.
Soluzione: È stata creata una gerarchia basata sull'interfaccia `Collectible` e sulla classe astratta `AbstractCollectible`. Le implementazioni concrete includono: Il comportamento specifico di ogni oggetto è incapsulato nel metodo `apply(Player target)`. Questo design permette al sistema di collisioni di interagire con qualsiasi oggetto raccolto senza conoscerne la natura specifica:

- `ExperienceGem`: Nel metodo `apply`, invoca `addExp()` sul giocatore per conferire esperienza.
- `HealthPotion`: Nel metodo `apply`, invoca `heal()`, ripristinando la salute fino al massimo consentito.
- `MoneyCoin`: Nel metodo `apply`, invoca `addMoney()` incrementando il contatore delle monete.

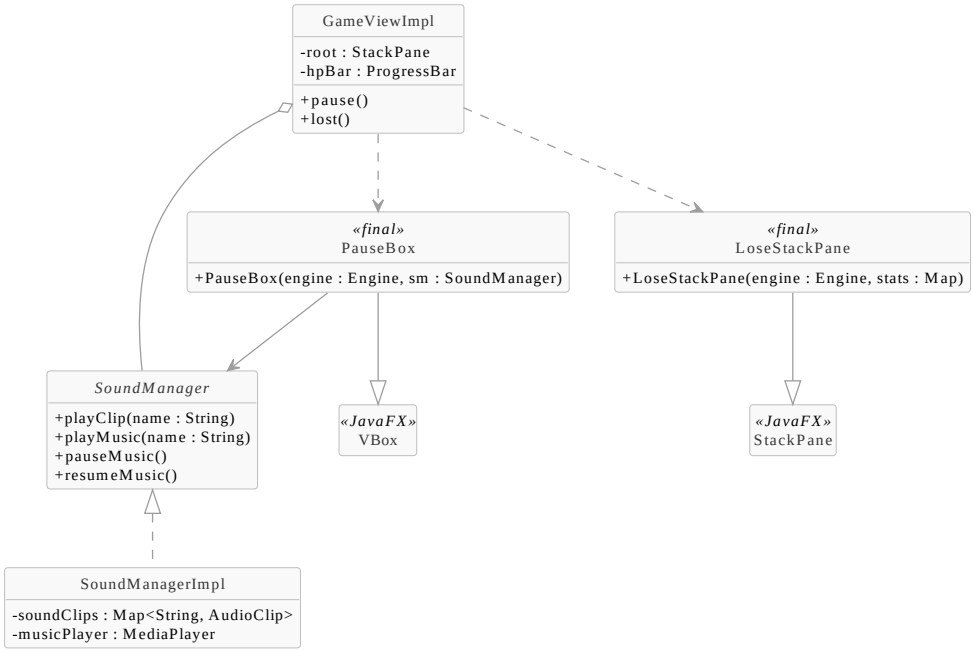

Figura 2.15: UML rappresenta la gestione degli effetti sonori e degli overlay di gioco

Gestione Audio (SoundManager)

Problema: Gestire la riproduzione concorrente di effetti sonori (brevi e frequenti) e musica di sottofondo (loop continuo) senza bloccare il thread principale.
Soluzione: Il componente `SoundManagerImpl` incapsula la logica delle librerie JavaFX `Media`. La classe distingue due tipologie di riproduzione:

- **Effetti Sonori:** Utilizza `AudioClip` per suoni a bassa latenza che possono sovrapporsi (es. raccolta oggetti, levelUp).
- **Musica:** Utilizza `MediaPlayer` per le tracce di sottofondo, gestendo il loop continuo e permettendo funzioni di pausa/ripresa quando il gioco viene interrotto.

Interfaccia Utente: Overlay di Gioco

Problema: Visualizzare schermate informative sopra la vista di gioco principale senza interrompere il rendering sottostante o creare conflitti tra i vari componenti JavaFX.

Soluzione: Sono stati realizzati componenti grafici personalizzati estendendo i container di JavaFX.

- **PauseBox**: Un menu a comparsa (`VBox`) che permette di riprendere il gioco o tornare al menu principale. La classe è stata resa `final` per risolvere vulnerabilità legate alla chiamata di metodi sovrascrivibili (`getChildren`) all'interno del costruttore.
- **LoseStackPane**: Una schermata di "Game Over" che presenta le statistiche finali della partita. Utilizza mappe chiave-valore per mostrare le etichette dei punteggi e offre opzioni di navigazione.

Entrambi i componenti sono gestiti dalla `GameViewImpl` che ne controlla la visibilità e il focus, garantendo che le interazioni da tastiera (es. tasto ESC) siano correttamente indirizzate.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Sono stati sviluppati dei test automatizzati per assicurare il funzionamento e la qualità del codice nel tempo, utili soprattutto in caso di operazioni di refactoring di alcune parti del codice (avvenute e future). È stata presa la scelta di non testare view e controller, ma di concentrarsi a fare solidi test del model in modo da garantire un corretto funzionamento delle fondamenta del software. A tale scopo è stata utilizzata la libreria di testing *JUnit*.

- Implementazioni delle armi
- Implementazione dei proiettili
- Implementazione dei nemici.
  I nemici close ranged devono avvicinarsi al player fino a toccarlo (entrare nella hitbox). I ranged devono stare fermi dentro una fascia (distanza minima e massima) dal player. I roaming devono vagare per la mappa senza dirigersi verso il player.
- Implementazione del player
- Implementazione dell'ambiente di gioco (la mappa)
  Sono stati testati vari compiti della mappa quali la gestione delle collisioni, l'aggiunta di nuovi elementi e la pulizia di elementi non più utili alla mappa.

3.2 Librerie esterne

Al fine di facilitare lo sviluppo sono state ricercate ed implementate delle librerie esterne per semplificare determinati compiti.

3.2.1 JUnit

Già esposto precedentemente, è stato utilizzato per stilare i test automatici del software.

3.2.2 JOML

La gestione delle posizioni di ogni elemento è stata facilitata di molto grazie alla libreria JOML, la quale si occupa di tutto ciò che riguarda il lato matematico del movimento di oggetti, ciò grazie ai suoi Vector2d.

3.2.3 Log4J

È stato necessario l'uso di un logger per evitare l'uso di `System.log.println` e simili. Si è preferito utilizzare Log4J in quanto consigliato a lezione.

3.2.4 Jackson

Libreria necessaria per la lettura di file .json, rendendoli di fatto oggetti concreti e permettendone il raggruppamento di tali all'interno di mappe.

3.2.5 JavaFX

La grafica è stata creata grazie a JavaFX, la quale si occupa di fornire un moderno API per lo sviluppo di interfacce grafiche in Java.

3.3 Note di sviluppo

Andrea Maria Castronovo

- Gestione delle collisioni tramite Stream: (permalink 1), (permalink 2)
- Optional: (permalink)
- Method reference lambda: (permalink)
- Utilizzo libreria JOML per calcoli con vettori: (permalink)

Leonardo Mengozzi

- Gestione della concorrenza con semafori per stoppare il gameLoop: (permalink 1) (permalink 2)
- Esempio di uso di optional:

  (permalink)
- Uso di generic:

  (permalink)
- Uso di stream:

  (permalink 1)

  (permalink 2)
- Esempio di uso di lambda:

  (permalink)

Lorenzo Mazzini

- Utilizzo di Supplier, Consumer e Function per la gestione delle armi: upgrade (permalink), generazione proiettili (permalink), esempio d'uso pratico (permalink).
- Utilizzo di Serializable per salvataggio dati: permalink.
- Uso di stream in varie occasioni: permalink 1, permalink 2, permalink 3.
- Uso della libreria Jackson per la lettura da .json: permalink.

Kleo Rama

- Gestione della view thread safe e lambda: (permalink)
- Utilizzo libreria JavaFX per gestione audio e caricamento da file: (permalink)

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Andrea Maria Castronovo

Mi sono occupato della parte di gestione della mappa e delle entità fornite dai miei colleghi. Dal punto di vista architetturale sono abbastanza sicuro di aver fatto una buona scelta organizzando i vari oggetti presenti nel gioco specializzandoli man mano; così facendo la gestione dei singoli oggetti risulta più immediata. Il problema che mi si è subito presentato però è stato quello della differenziazione tra i tipi effettivi: per esempio, dopo che un'entità ha sparato un proiettile, mi serviva un modo per identificare chi lo avesse sparato, così da poter ignorare la collisione tra proiettile e attaccante. Ho optato per differenziare ogni oggetto in base all'interfaccia, ad esempio se l'oggetto "Movable", presente in una collisione, implementava Player o Enemy tramite la keyword "instanceof" di Java. Questo ha portato a del codice non particolarmente pulito.

4.1.2 Leonardo Mengozzi

Mi sono occupato delle entità nemiche, della logica di spawn, del game loop del controller e alcune interfacce grafiche e la logica di interfacciamento al controller. Il risultato finale mi soddisfa in quanto è una buona base solida per possibili sviluppi futuri già in parte abbozzati, come lo shop e un sistema di valute. Una sezione migliorabile è la logica di spawn dei nemici che, per quanto sia già stata perfezionata, rimane acerba e rifinibile su aspetti come i parametri su cui viene scelto il nemico da spawnare e probabilità di spawn ad hoc per ogni nemico.

4.1.3 Lorenzo Mazzini

Mi sono occupato della creazione delle armi per le entità e qualche altra funzione utile al progetto. Posso ritenermi soddisfatto del lavoro fatto, sia perché l'obiettivo finale è stato correttamente raggiunto, ma anche perché, sulla base di un architettura corretta, mi sono accorto che aggiungere nuove funzionalità al gioco è risultato piuttosto facile e spesso anche intuitivo. Durante la realizzazione sono sorti diversi problemi per ciò che riguarda le armi: primo di tutti è la creazione di diverse tipologie di armi, su cui è stato necessario effettuare diverse considerazioni partendo dal modello iniziale. Inoltre anche le armi a puntamento sono state particolarmente difficili da gestire, in quanto è la view a deciderne la direzione: è per questo nata la classe PointerFactory. Infine, ho trovato piuttosto difficile rendere il gioco compatibile per ogni sistema operativo, spesso optando per codice sicuramente non ottimale.

4.1.4 Kleo Rama

Il mio lavoro si è focalizzato sulla definizione dell'entità protagonista (Player), sulla gestione dell'Input per il movimento e sull'interfaccia utente (Audio e Overlay).
Dal punto di vista architetturale ritengo che la scelta di mantenere separata la gestione degli input dalla libreria grafica sia stata vincente: l'introduzione di un livello intermedio di traduzione dei comandi ha mantenuto il Controller pulito e indipendente da JavaFX.
Per quanto riguarda la progettazione, avrei voluto utilizzare un numero maggiore di Design Pattern. In altre aree come il sistema audio o la gestione delle schermate UI ho optato per soluzioni più dirette per non complicare eccessivamente la struttura dato il tempo a disposizione. A posteriori, riconosco che un uso più esteso di pattern strutturali avrebbe potuto rendere alcune componenti ancora più eleganti.

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Andrea Maria Castronovo

Lavorare in un team è stata per me una cosa nuova: tutti i piccoli progetti che ho affrontato finora avevano in comune il fatto che qualsiasi decisione spettasse unicamente a me. Fin dalle scuole superiori ho sempre avuto difficoltà a fare lavori di gruppo con dei colleghi, per questo motivo ho trovato

davvero interessante la presenza di un progetto di gruppo obbligatorio in questo corso, soprattutto considerando i livelli di attenzione e dettaglio richiesti. Grazie a questo progetto ho potuto esercitarmi molto per quanto riguarda l'organizzazione e la coordinazione con gli altri, e credo che riuscire ad allenare questa "soft skill" prima di entrare nel mondo del lavoro sia una grande fortuna.

4.2.2 Leonardo Mengozzi

Questo è stato il primo grande progetto. Mi sono trovato a lavorare e sincronizzare quantità di istruzioni mai affrontate prima e sopprattuto a interfacciarmi in un progetto di cui non avevo il pieno controllo. L'abilità nel confronto, l'ascolto, e le altre numerose soft skill che questo progetto hanno richiesto e che mi ha portato ad'affinare si riveleranno utili in futuro.

4.2.3 Lorenzo Mazzini

Essendo il mio primo, grande progetto a seguito di molti altri molto più piccoli, è stata sicuramente propedeutica per tutti i lavori che farò in futuro: è stato necessario un livello di dettaglio nella scrittura del codice che non avevo mai affrontato prima, ma che mi aiuterà sicuramente in futuro per scrivere programmi in modo migliore. Ho fatto anche esperienza nel lavorare in un team, anche questa cosa che non avevo mai fatto prima; seppur si debba mantenere un livello di coordinazione e di ordine per quanto riguarda l'organizzazione dei lavori, risulta anche più facile sviluppare nuove funzioni e risolvere problemi in quelle già scritte.

4.2.4 Kleo Rama

Affrontare un progetto di queste dimensioni in gruppo è stato un passo fondamentale per la mia crescita come sviluppatore. Ho compreso sul campo quanto sia vitale scrivere codice che non sia solo "funzionante", ma anche leggibile e manutenibile, cosa che nessun altro corso ci aveva mai insegnato prima di ora.
In particolare, questo percorso mi è stato fondamentale per apprendere l'uso corretto di Git, che prima di questo corso non avevo mai utilizzato. Uno strumento che inizialmente rappresentava una sfida alla fine si è rivelato indispensabile per gestire il flusso di lavoro condiviso e che ora considero una competenza acquisita essenziale.
Infine, l'esperienza è stata resa estremamente positiva dai miei compagni: la loro costante collaborazione ed efficace comunicazione hanno permesso di

risolvere velocemente le criticità e di lavorare in modo produttivo, insegnandomi il valore del confronto nel lavoro di squadra.

Appendice A

Guida utente

Figura A.1: Schermata principale del gioco.

All'avvio dell'applicazione si può selezionare il personaggio da giocare e avviare una nuova partita o avviare la partita con il personaggio selezionato di default. Dalla stessa interfaccia si può anche andare nel Pokédex, chiccando il rispettivo menu, per visualizzare i risultati totali delle partite precedenti. Al primo avvio il Pokédex si presenterà vuoto non avendo ancora effettua alcuna partita.

Avviato il gioco si potrà muovere il personaggio con "WASD" e direzionare gli attacchi con il cursore del mouse.

Durante la partita a ogni nuovo livello del player apparità un'interfaccia per selezionare un power-up. Si potranno usare le frecette SU e GIÙ per

selezionare il potenziamento e premere INVIO per applicarlo oppure con un doppio click del muose.

Sempre durante la partita premendo ESC è possibile mettere il gioco in pausa e dalla interfaccia che appare si può riprendere la partita o tornare al menu principale.

A fine partita si visualizza l'interfaccia per tornare al menu di gioco o chiudere l'applicazione, in entrambi i casi si effettua il salvataggio dei risultati della parita, i cui dati verranno memorizzati nel Pokédex.

La chiusura della finestra in partita non comporta il salvataggio dei dati della partita.

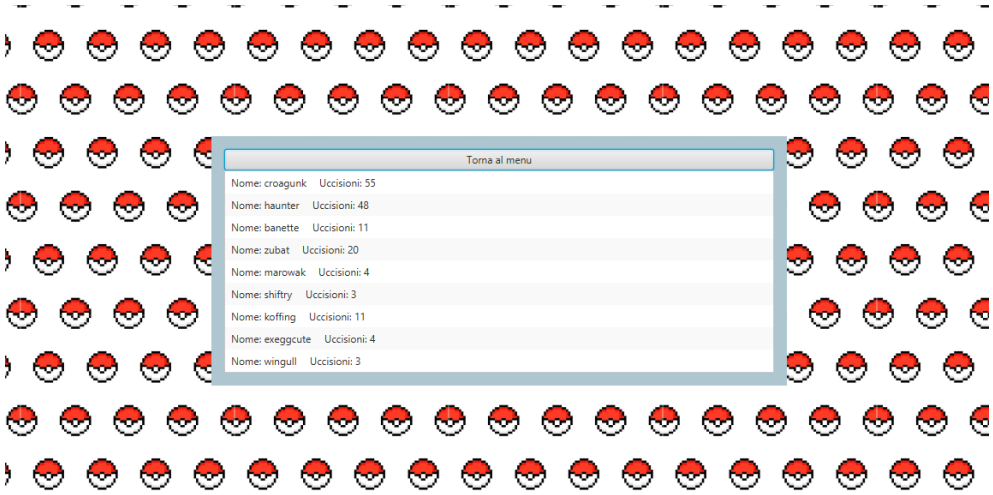

Figura A.2: Schermata del Pokedex, dov'è possibile vedere quali e quanti nemici sono stati eliminati nelle partite.

Appendice B

Esercitazioni di laboratorio

Andrea Maria Castronovo

- Laboratorio 6 : permalink lab 6
- Laboratorio 7 : permalink lab 7
- Laboratorio 8 : permalink lab 8
- Laboratorio 9 : permalink lab 9
- Laboratorio 10 : permalink lab 10
- Laboratorio 11 : permalink lab 11
- Laboratorio 12 : permalink lab 12

Leonardo Mengozzi

- Laboratorio 6:
  https://virtuale.unibo.it/mod/forum/discuss.php?d=206731#p284046
- Laboratorio 7:
  https://virtuale.unibo.it/mod/forum/discuss.php?d=207193#p284727
- Laboratorio 8:
  https://virtuale.unibo.it/mod/forum/discuss.php?d=207921#p285429
- Laboratorio 9:
  https://virtuale.unibo.it/mod/forum/discuss.php?d=208718#p286620

- Laboratorio 10:
  https://virtuale.unibo.it/mod/forum/discuss.php?d=209589#p288298
- Laboratorio 11:
  https://virtuale.unibo.it/mod/forum/discuss.php?d=210617#p289204
- Laboratorio 12:
  https://virtuale.unibo.it/mod/forum/discuss.php?d=211539#p290657

Lorenzo Mazzini

- Laboratorio 6 : https://virtuale.unibo.it/mod/forum/discuss.php?d=206731#p283985
- Laboratorio 7 : https://virtuale.unibo.it/mod/forum/discuss.php?d=207193#p284642
- Laboratorio 8 : https://virtuale.unibo.it/mod/forum/discuss.php?d=207921#p286086
- Laboratorio 9 : https://virtuale.unibo.it/mod/forum/discuss.php?d=208718#p286659
- Laboratorio 10 :
- Laboratorio 11 : https://virtuale.unibo.it/mod/forum/discuss.php?d=210617#p288950
- Laboratorio 12 : https://virtuale.unibo.it/mod/forum/discuss.php?d=211539#p290624