OOP24 - RUNWARRIOR

Samuele Bianchedi, Riccardo Cornacchia
Francesca Gatti, Giovanni Maria Rava

27 luglio 2025

Indice

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Il gruppo si pone come obbiettivo quello di realizzare una reinterpretazione del famoso gioco Super Mario Bross del 1986. Il gioco ha come personaggio principale un cavaliere, che tramite l'input dell'utente si muove in una mappa 2D. L'obbiettivo del cavaliere è salvare una principessa tenuta prigioniera da uno stregone, completando diversi livelli che lo condurranno al castello nel quale è rinchiusa. Nel gioco sarà possibile, tramite un negozio, comprare un altro personaggio, se si ha raccolto un numero sufficiente di monete. All'interno del gioco, oltre a diversi ostacoli, sono presenti nemici che il cavaliere può uccidere per rimanere in vita e completare il livello.

Requisiti funzionali

- Il personaggio deve avanzare, indietreggiare e saltare all'interno della mappa.
- Gestione delle collisioni del personaggio con nemici, ostacoli, monete e potenziamenti.
- Il personaggio può ottenere due potenziamenti che lo aiuteranno nella sua avventura.
- Gestione di nemici ed ostacoli diversi in base alla mappa.
- Creazione di un sistema di punteggio, che verrà mostrato al completamento del livello.

Requisiti non funzionali

- Implementazioni di una quarta mappa.
- Gestione di restart e checkpoint.
- Creazione di nemici e ostacoli più complessi.
- Musica e suoni.

1.2 Modello del Dominio

RunWarrior è gioco ambientato in un mondo fantastico in cui il personaggio principale deve affrontare 3 livelli diversi, selezionabili mediante un menù. In questi livelli il personaggio deve portersi muovere per sopravvivere e uccidere i nemici. Il movimento del personaggio è gestito tramite tastiera. All'interno della mappa sono posizionate delle uova che racchiudono al loro interno i 2 possibili powerup, diversi per Warrior e Wizard. Per il completamento del gioco è neccessario sbloccare tutti i livelli in maniera sequenziale, che si considerano terminati tramite l'ingresso in un portale, ad eccezione del terzo che si conclude con il castello della principessa. All'interno di ogni livello possono essere presenti degli ostacoli letali (MapElement) e dei nemici (Enemy) e delle monete (Coin) con il quale il personaggio può collidere. Se ciò accade con un ostacolo o con un nemico perde un potenziamento, nel caso lo avesse, altrimenti la partita finisce. I nemici sono di 5 tipi:

- Goblin
- Snake
- Wizard
- Monkey
- Guard

Gli ostacoli letali sono:

- Fungo
- Cactus
- Camino con fuoco

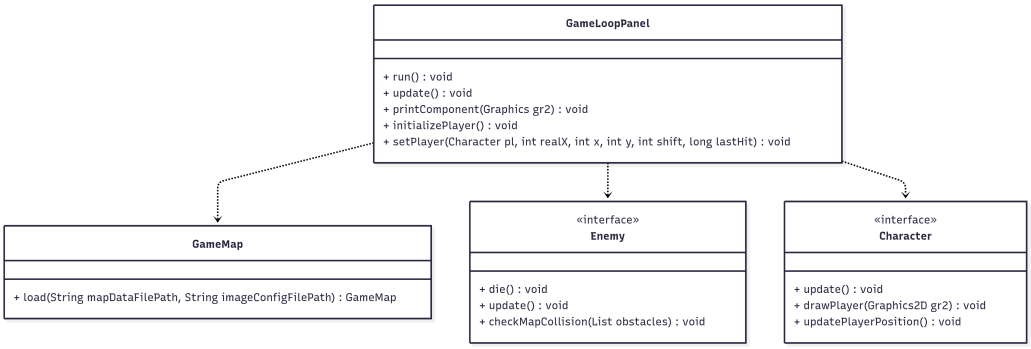

Figura 1.1: UML del modello del dominio

Capitolo 2

Design

2.1 Architettura

L'architettura di RunWarrior segue il pattern architetturale MVC (Model - View - Controller). Il GameLoopController gestisce l'aggiornamento del gioco e di tutte le sue entità a seguito dei diversi eventi che possono capitare durante la sessione. All'interno della classe, dal momento in cui viene acquistato il nuovo personaggio, ricevendo le informazione da GameSaveManager e Shop, viene gestito il cambio skin. Gli input da tastiera per muovere il personaggio vengono gestiti tramite la comunicazione tra le classi Character-Comand e MovementHandlerImpl. Quindi il Character è una entità reattiva che modifica il proprio stato a seguito delle diverse collisioni con le entità, tramite CollisionDetection, KillDetection e PowerUpDetection. Il pattern MVC implementato consente di mantenere lo stato del controller nell'eventualità che si modifichi la view. inserire UML delle classi principali dell'MVC

2.2 Samuele Bianchedi

2.3 Riccardo Cornacchia

2.4 Francesca Gatti

2.5 Giovanni Maria Rava

Problema: I nemici del gioco devono muoversi da soli, invertire direzione appena urtano un ostacolo e restare sincronizzati con lo scorrimento della camera.

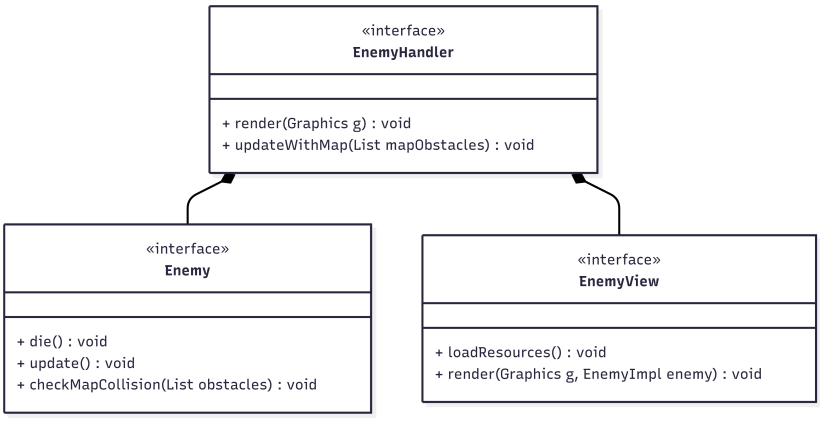

Figura 2.1: UML dell'MVC per la gestione dei nemici

Soluzione: Per gestire il movimento autonomo dei nemici e l'inversione di marcia in presenza di ostacoli ho adottato il pattern *Model-View-Controller*. Il controller centrale è l'*EnemyHandler*, che a ogni ciclo di gioco scorre l'intera lista di nemici: per ciascuno aggiorna prima lo stato logico delegando questa responsabilità all'oggetto model, *EnemyImpl*, poi ne richiede la rappresentazione grafica all'oggetto view specifico, (ad esempio *GoblinView*), per farla disegnare sul pannello di gioco. In questo modo la logica di collisione con la mappa e la scelta del momento in cui invertire la direzione restano incapsulate nel model, mentre il caricamento e il rendering rimangono confinati nella view; il controller non conosce i dettagli interni di nessuna delle due parti, ma si limita a coordinarne l'interazione. Questo assetto garantisce un codice più estendibile e manutenibile: introdurre una nuova tipologia di nemico richiede soltanto di aggiungere una nuova View, senza toccare il controller, e ogni modifica grafica o comportamentale resta circoscritta al relativo livello dell'architettura.

Problema: Gestione il salvataggio del gioco e del caricamento del salvataggio all'apertura del gioco.
Soluzione: Il salvataggio e il ripristino della partita sono gestiti da un'unica classe, GameSaveManager, progettata come singleton per garantire un punto di accesso centralizzato e soprattutto per mantenere una unica istanza del file. Per rendere effettivo ciò, all'interno della classe viene quindi eseguito un controllo specifico, in modo da non creare più file di salvataggio. Quando il giocatore conclude un livello o chiude l'applicazione, i controlle, ScoreController e CoinController invocano i relativi metodi per modificare le variabili e salvare lo stato di gioco, che si compone di:

- Un intero che indica i livelli completati.
- Un intero che indica lemonete raccolte.
- Un booleano che indica l'acquisto della skin secondaria.
- Una stringa che indica la skin scelta.

All'avvio successivo, lo stato viene ricostruito e propagato ai vari model che lo necessitano; in caso contrario viene generata una partita nuova con parametri standard. L'intera logica di I/O è dunque isolata in una sola classe, priva di dipendenze dal motore di gioco; il resto del codice che dipende da un'interfaccia minimale, Questa architettura rende effettivo il singleton e semplifica i test automatizzati.

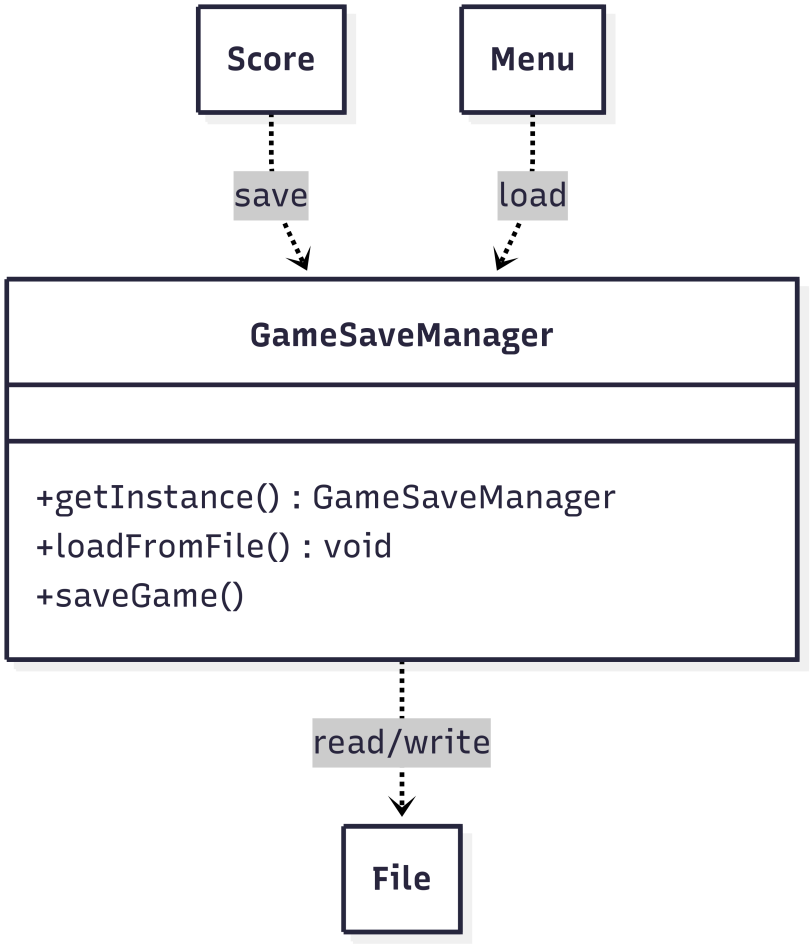

Figura 2.2: UML dell'MVC per la gestione dei nemici