OOP24 - RUNWARRIOR

Samuele Bianchedi, Riccardo Cornacchia
Francesca Gatti, Giovanni Maria Rava

28 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

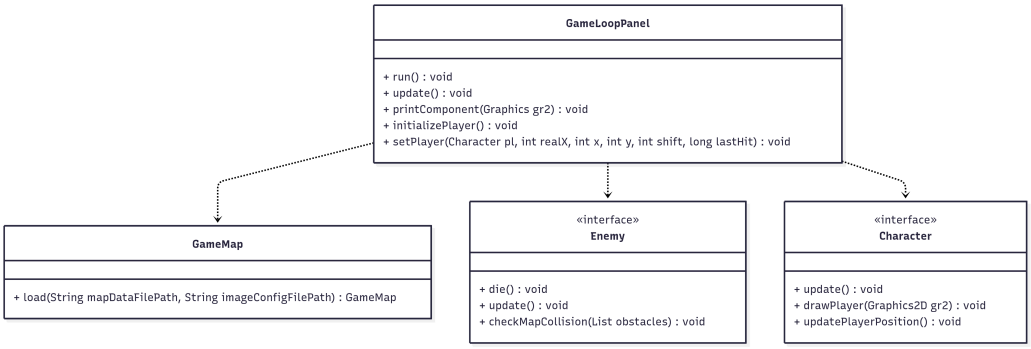

Figura 1.1: UML del modello del dominio

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 CharacterComand 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

Problema: L'implementazioni delle classi del modello soffriva di una criticità legata all'incapsulamento. SptoBugs ha rilevato molti warning che indicavano oggetti mutabili passati e restituiti tramite riferimento diretto. Questo poteva creare un'architettura fragile, incline ad errori e/o modifiche involontarie. **Soluzione**: Per risolvere questo problema e rendere sicuro i dati ho scelto di implementare copie difensive.

- Per oggetti come BufferedImage e int[][]: nei costruttori e nei metodi setter invece di memorizzare il riferimento all'oggetto, ne viene creata una copia completa.

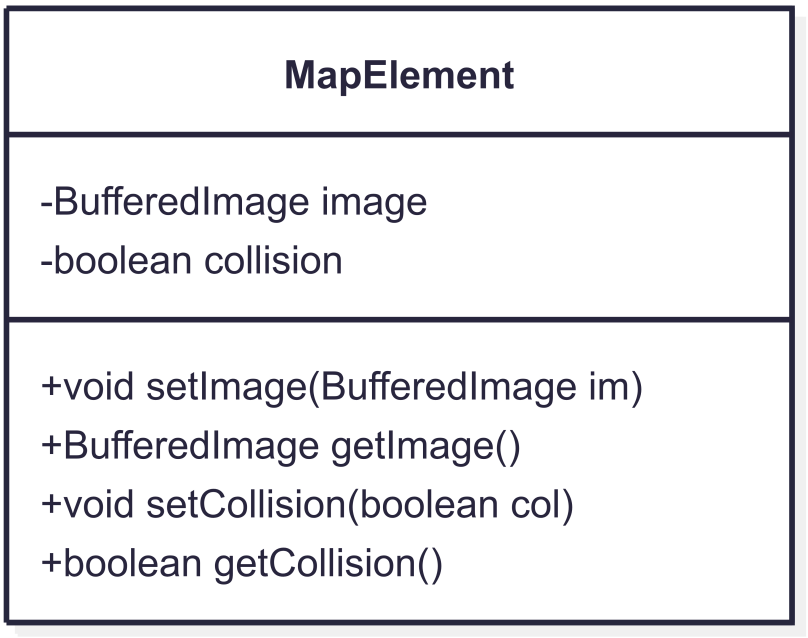

Figura 2.1: UML della classe MapElement nel model

- Nei metodi getter: invece di restituire una riferimento all'oggetti ne viene creato uno nuovo da passare al chiamante
- Per le collezioni: i metodi getter sono stati modificato per restituire una vista immodificabile della collezione.

I pro sono ovviamente la sicurezza dei dati che sono protetti da modifiche esterne. Di contro abbiamo una riduzione della performance, in quanto copiare oggetti innumerevoli volte costa abbastanza e in alcuni casi ha portato a "freeze" dell'applicazione. Nel seguente schema UML si mostra come i campi siano privati e l'accesso avvenga tramite metodi pubblici che lavorano restituendo copie. Il pattern utilizzato è stato Immutable Object, un idioma fondamentale di OOP. Quest'ultimo spiega gli oggetti immutabili, oggetti che non si possono veder modificato lo stato dopo la loro creazione. Sebbene gli oggetti nel progetto non siano perfettamente immutabili a causa dei setter, quello a cui si aspirava era un oggetto non modificabile lontano da errori imprevedibili.

2.3 Riccardo Cornacchia

2.4 Francesca Gatti

Problema: Nel gioco è previsto un sistema di personalizzazione del personaggio tramite l'acquisto e la selezione di una nuova skin. Il problema principale affrontato è stato quello di separare la gestione della parte logica del negozio, ovvero il controllo delle monete, delle skin e la selezione, dalla sua interfaccia grafica, mantenendo la possibilità di modificarlo in futuro. **Soluzione**: Per affrontare questo problema è stato adottato il pattern Model-View-Controller (MVC), suddividendo le responsabilità tra i tre livelli:

- Model: rappresentato dalla classe Shop, gestisce le skin disponibili, quella selezionata e lo stato di sblocco della skin (unlock). Nel model sono inoltre presenti le due classi Skin e Score che permettono la gestione degli oggetti skin e punteggio da parte dello Shop.

- View: la classe ShopView è responsabile della visualizzazione delle informazioni (monete, stato skin) e della gestione dei pulsanti che consentono l'acquisto e la selezione della nuova skin. L'utente è messo al corrente delle azioni che può fare nello shop grazie a messaggi di dialogo che appaiono una volta cliccato il pulsante desiderato.

- Controller: ho implementato l'interfaccia ShopController che è responsabile della logica legata all'acquisto e alla selezione delle skin e all'acquisizione dei dati e delle informazioni richieste dalla View. Inizialmente avevo implementato un'unica classe all'interno del Model (la classe Shop), ma il codice risultava molto pesante e difficile da gestire.

L'introduzione del pattern MVC ha permesso di ottenere una struttura più chiara, estendibile e orientata al riuso.

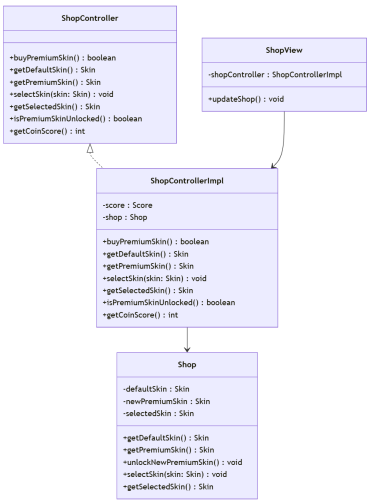

Figura 2.2: UML dell'MVC per la gestione dello shop

Problema: per lo sviluppo del gioco è stato necessario creare un sistema che gestisse la visualizzazione e la raccolta delle monete. Le diverse funzionalità implementate sono il caricamento da file, il disegno delle monete sulla mappa, il rilevamento delle collisioni e l'aggiornamento di un punteggio. **Soluzione:** ho creato questo sistema facendo una separazione logica delle responsabilità, suddividendo il sistema in più componenti specializzate:

- La classe Coin rappresenta l'oggetto moneta, con le informazioni sulla posizione, il suo stato di raccolta e l'immagine.
- L'interfaccia CoinController (impelementata da CoinControllerImpl) gestisce il caricamento delle monete, la logica di visualizzazione e il conteggio.
- L'interfaccia CoinDetection (implementata da CoinDetectionImpl) che si occupa del rilevamento delle collisioni tra player e moneta.

Le classi sono collegate a Score, ScoreController e GameSaveManager per il salvataggio del punteggio totale. All'interno del CoinController è stato necessario aggungere il metodo updatePlayer che si occupa dell'aggiornamento delle monete in base ai cambiamenti subiti dal player durante il gioco, quali i potenziamenti. **Problema:** nel gioco era necessario monitorare il tempo di gioco trascorso, per poterlo visualizzare durante la partita e avere un resoconto finale del tempo impiegato per completare il livello. **Soluzione:** Per affrontare il problema è stata creata l'interfaccia Chronometer che si occupa della gestione del tempo tramite un oggetto di tipo javax.swing.Timer. Il

timer aggiorna periodicamente il tempo trascorso in millisecondi rispetto al momento di avvio. La classe fornisce metodi per avviare, fermare e ottenere il tempo in formato numerico e in formato stringa (HH::mm::ss::d). Infine, il tempo viene disegnato a schermo tramite la classe GameLoopPanel che recupera il valore dal Chronometer e lo mostra in tempo reale durante il gioco.

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.

Soluzione: Per gestire il movimento autonomo 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. 2.3

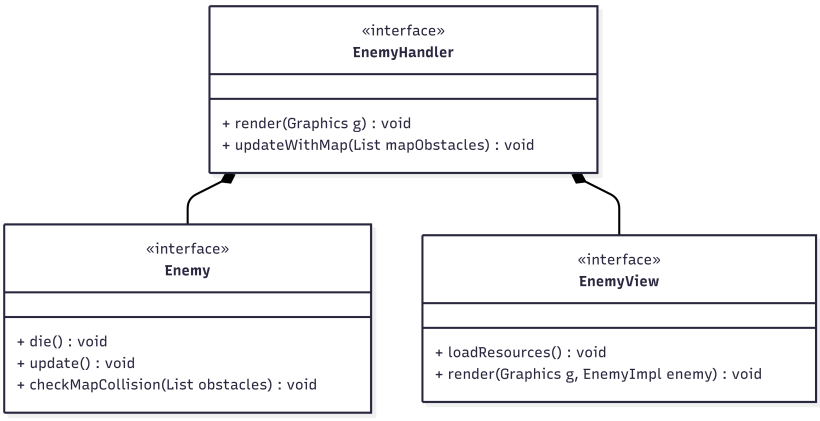

Figura 2.3: UML dell'MVC per la gestione dei nemici

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 controller, *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; Questa architettura rende effettivo il singleton e semplifica i test automatizzati. 2.4

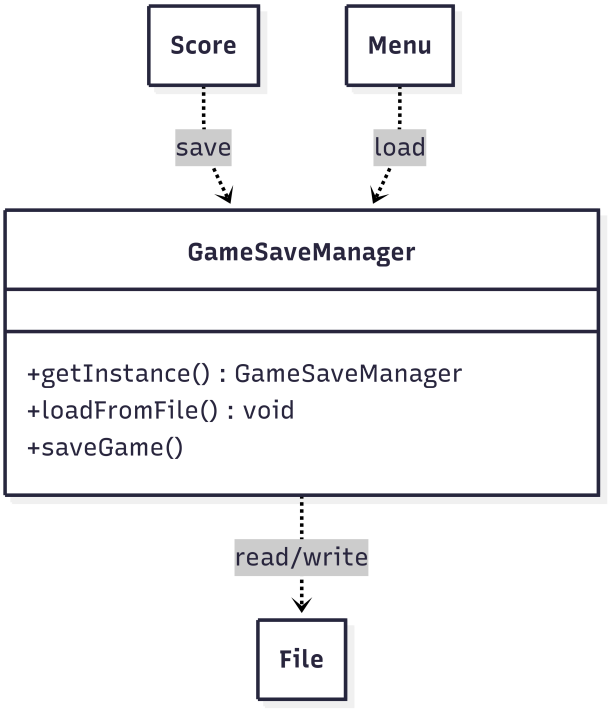

Figura 2.4: UML dell'MVC per la gestione dei nemici

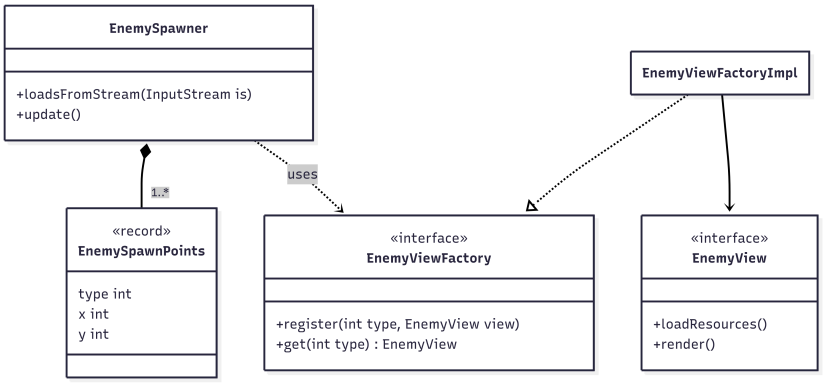

Figura 2.5: UML gestione spawing nemici

Problema Nel livello di gioco occorre generare molti nemici in posizioni predefinite: ognuno deve comparire al momento giusto, nella coordinata corretta e con la propria grafica, ma senza mescolare la logica di lettura dei dati con la scelta delle view.

Soluzione: Il file *enemies*.txt*, letto da EnemySpawner, elenca riga per riga tipo di nemico e coordinate. Ogni riga viene trasformata in un oggetto immutabile EnemySpawnPoints, che serve come contenitore dei dati di spawn. Durante l'esecuzione EnemySpawner rimane in ascolto dello scorrimento della mappa di gioco: quando un punto di spawn entra nell'area visibile, istanzia il relativo modello di nemico e, anziché occuparsi direttamente della grafica, delega la creazione della vista a EnemyViewFactory. Quest'ultima contiene un metodo che riceve l'identificativo numerico del nemico e restituisce la classe di rendering adeguata; la sua implementazione concreta, EnemyViewFactoryImpl, contiene la mappatura fra tipologia e view corrispondente (as esempio 1 → GuardView). In questo modo la lettura del file, la gestione della memoria e il caricamento delle immagini restano ben separati: per aggiungere un nuovo tipo di nemico è sufficiente aggiungere una riga al file di mappa e una voce nella factory, senza toccare né il game-loop né lo spawner, mantenendo il sistema estendibile e pulito. 2.5

Capitolo 3

Sviluppo

3.1 Testing automatizzato

I test sono stati scritti con JUnit 5, quindi Gradle li esegue automaticamente e l'intera suite gira senza alcun intervento manuale. Ogni caso di prova carica risorse dedicate, in particolare mappe di test, copie ridotte e semplificate di quelle di produzione, così l'esito non risente di eventuali modifiche ai livelli reali. La combinazione fra esecuzione automatizzata e risorse isolate garantisce che l'intero motore grafico, la gestione dei nemici e la correttezza del sistema di salvataggio vengano controllati con rapidità, senza interferenze con il materiale definitivo del gioco.

Per quanto riguarda i nemici, i test verificano la corretta restituzione del proprio Rectangle, assicurano che le collisioni con gli altri elementi della mappa siano gestite a dovere, e controllano l'inversione della velocità in caso di impatto con un ostacolo. Con la medesima logica, la classe TestGoblinView verifica il caricamento corretto delle immagini.

GameSaveManagerTest accerta che il pattern singleton sia rispettato, impedendo la creazione di più istanze del gestore dei salvataggi. Controlla il corretto funzionamento della scrittura e lettura tramite file.

Il test riguardante il player è TestPlayerCollision, che ha lo scopo di verificare con 4 test le possibili interazioni del player con le altre entità. Si è settata uan dimensione di default dei blocchi della mappa. Quindi vengono verificati i metodi principali delle classi "Detection". Nel caso dei blocchi di mappa e delle uova, vengono verificate le direzioni delle collisioni. Si verifica poi se il player raccoglie in modo corretto il powerup e se cambia skin. Il cambio di skin viene verificato anche quando il player collide con un nemico; se la collisione avviene dall'alto il nemico invece muore. Infine si verifica se le monete vengono raccolte.

3.2 Note di sviluppo

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Samuele Bianchedi

4.1.2 Riccardo Cornacchia

4.1.3 Francesca Gatti

4.1.4 Giovanni Maria Rava

Appendice A

Guida utente

Per il movimento del player si usano 4 frecce della tastiera:

- **Freccia destra**: per muoversi verso destra
- **Freccia sinistra**: per muoversi verso sinistra
- **Freccia in alto**: per saltare, più la si preme più il salto sarà alto. Mentre si è in volo il player si può muovere a destra e sinistra.
- **Shift**: per attaccare con spada o bastone nel caso in cui si è giunti al secondo potenziamento. Attenzione, mentre si è in movimento il player può attaccare una volta ogni secondo.