

OOP24 - RUNWARRIOR

Samuele Bianchedi, Riccardo Cornacchia
Francesca Gatti, Giovanni Maria Rava

29 luglio 2025

Indice

1	Analisi	2
1.1	Descrizione e requisiti	2
1.2	Modello del Dominio	3
2	Design	5
2.1	Architettura	5
2.2	Samuele Bianchedi	5
2.3	Riccardo Cornacchia	7
2.4	Francesca Gatti	11
2.5	Giovanni Maria Rava	14
3	Sviluppo	19
3.1	Testing automatizzato	19
3.2	Note di sviluppo	20
3.3	Samuele Bianchedi	20
3.4	Riccardo Cornacchia	20
3.5	Francesca Gatti	20
3.6	Giovanni Maria Rava	20
4	Commenti finali	21
4.1	Autovalutazione e lavori futuri	21
4.1.1	Samuele Bianchedi	21
4.1.2	Riccardo Cornacchia	21
4.1.3	Francesca Gatti	21
4.1.4	Giovanni Maria Rava	21
A	Guida utente	22
B	Bibliografia	23

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Il gruppo si pone come obbiettivo quello di realizzare una reinterpretazione del famoso gioco Super Mario Bros 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 portarsi 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 necessario 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

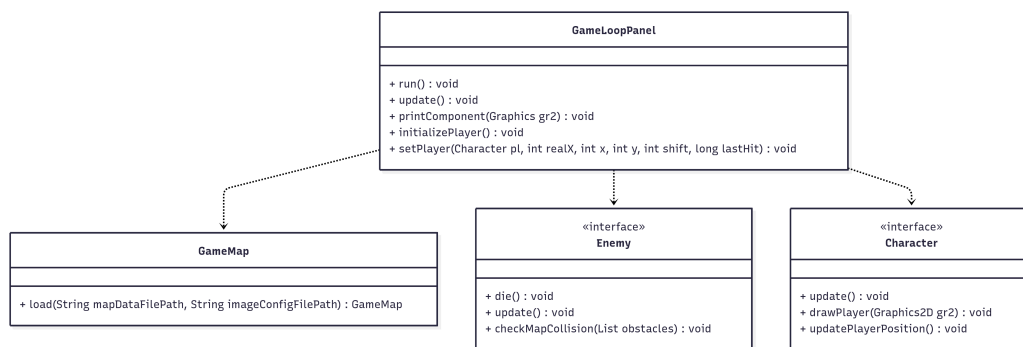


Figura 1.1: UML del modello del dominio

Capitolo 2

Design

2.1 Architettura

L'architettura di RunWarrior segue il pattern architetturale MVC (Model - View - Controller). Il GameController 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 informazioni 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 CharacterCommand 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'implementazione 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 imm modificabile 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

Problema: si deve gestire la creazione del player considerando che le 2 entità warrior e wizard possono cambiare in modo attivo 3 skin (senza armatura, con armatura, con armatura e spada/bastone) durante la sessione di gioco.

Soluzione: all'interno dell'architettura MVC, ho realizzato una classe astratta `AbstractCharacterImpl` che implementa l'interfaccia `Character` per creare l'entità player, nel model del progetto. In questa classe vengono definiti metodi standard che il player esegue sempre e i campi che ogni entità player deve avere. Tale classe astratta viene estesa da 6 classi concrete (`NakedWarrior`, `NakedWizard`, `ArmourWarrior`, `ArmourWizard`, `SwordWarrior`, `StickWizard`). Ogni sottoclasse fa `Override` del metodo astratto `playerImage()`, mentre `SwordWarrior` e `StickWizard` fanno `Override` del metodo `updateAttackCollision()` che setta l'eventuale area dell'arma. `AbstractCharacterImpl` implementa il metodo `update()`, che richiama i metodi `playerImage()` e `updatePlayerPosition()`, modificati ad hoc da ciascuna sottoclasse. Tramite questa architettura viene sfruttato il Template pattern che stabilisce lo scheletro dell'operazione principale (`update`) nella classe astratta e delega alle sottoclassi la definizione di uno o più aspetti specifici dell'algoritmo (`playerImage` e `updateAttackCollision`). Il Template method è l'`update()` che definisce l'algoritmo principale; esso infatti richiama i due metodi definiti dalle sottoclassi.

Problema: gestione dell'animazione, ovvero il cambio continuo dei frame per ogni specifica azione del personaggio; gestione del movimento del player, collegato ad input da tastiera.

Soluzione: all'interno dell'architettura MVC, ho realizzato due interfacce `CharacterAnimationHandler` e `CharacterMovementHandler`, rispettivamente implementate da `CharacterAnimationHandlerImpl` e `CharacterMovementHandlerImpl`, nel controller del progetto. L'obiettivo è quello di delegare i 2 comportamenti fondamentali di ogni entità player a due classi distinte nella sezione controller. Entrambe le interfacce vengono usate da `AbstractCharacterImpl` e i loro metodi vengono richiamati all'interno di `update()` e `drawPlayer(Graphics2D)`. Per questa architettura mi sono basato sul pattern Strategy, ma applicandone una variazione, cioè incapsulare in 2 interfacce diverse animazione e movimento del player. Queste due interfacce rappresentano la strategia. Per gestire tali comportamenti entrambe le classi usano `CharacterCommand`, la classe che gestisce l'input da tastiera. `CharacterAnimationHandler` definisce metodi fondamentali come: `frameChanger()`, per gestire il cambiamento dei frame in base agli input e `imagePlayer(boolean rightDirection)`, che fornisce l'immagine corretta in base al frame corrente. `CharacterMovementHandler` definisce metodi fondamentali come: `setLoca-`

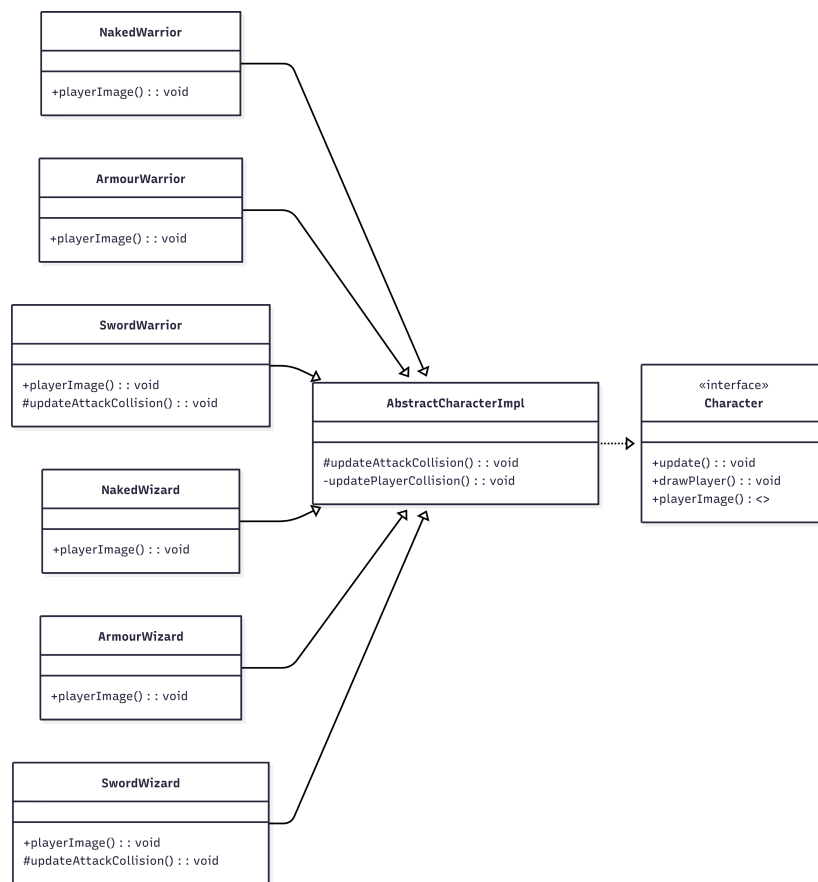


Figura 2.2: UML del pattern Template per il personaggio principale

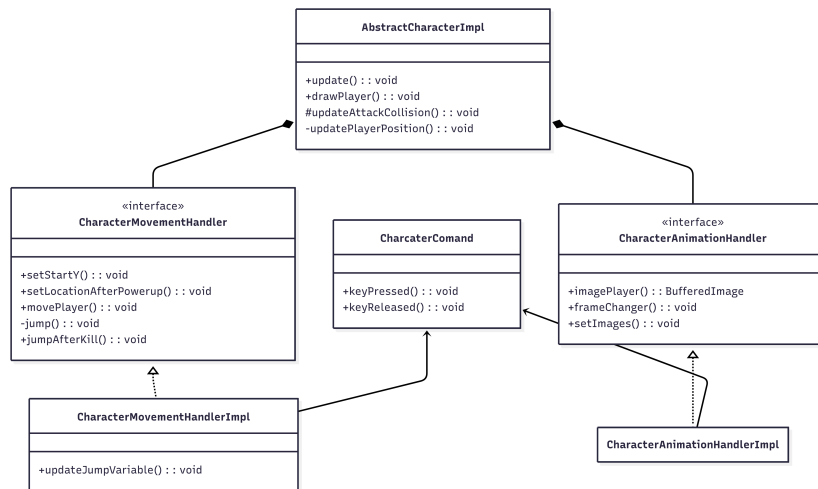


Figura 2.3: UML architettura implementazione animazione e movimento player

tionAfterPowerUp(), per reimpostare la posizione del player dopo aver guadagnato o perso una vita, movePlayer(), per gestire spostamento a destra e sinistra, salto e collisione con le diverse entità.

Problema: gestione delle collisioni con le diverse entità: blocchi della mappa, powerups, nemici e monete. Mantenere la relazione con le classi che gestiscono eventuali conseguenze di una specifica collisione.

Soluzione: in questo caso all'interno dell'architettura MVC, ho realizzato 4 interfacce nel controller del progetto, ognuna delle quali gestisce le collisioni con 1 delle 4 entità: CollisionDetection, per i blocchi delle mappe, PowerUpDetection, per la collisione con l'uovo e i powerups, KillDetection, per i nemici, e CoinDetection, per le monete, rispettivamente implementate da CollisionDetectionImpl, PowerUpDetectionImpl, KillDetectionImpl e CoinDetectionImpl.

Le 4 interfacce vengono usate da CharacterMovementHandlerImpl e i loro metodi vengono richiamati all'interno di movePlayer(). Le 4 interfacce descritte rappresentano la strategia per il pattern Strategy, che anche in questo caso non si tratta di Strategy puro, ma di una definizione di 4 strategie separate, infatti ognuna di esse incapsula i metodi utili a verificare le collisioni con le diverse entità.

- CollisionDetection usa checkCollision(), che verifica la collisione del player con i blocchi della mappa tenendo conto di 6 punti della sua area di collisione che lo "circonda". Per verificare che tipologia di blocco viene toccata e in quale direzione avviene la collisione richiama touch-

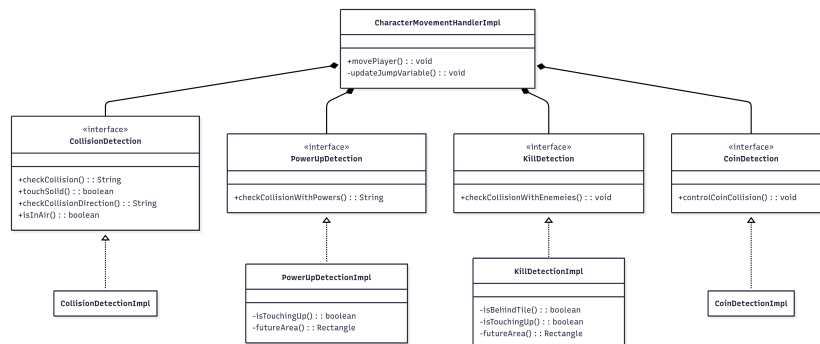


Figura 2.4: UML architettura per gestire le collisioni

Solid() e checkCollisionDirection(). Inoltre CollisionDetection fornisce un metodo gameOver() che restituisce true se il player cade in una fossa, e win() quando tocca un blocco di tipo “Portale”.

- PowerUpDetection usa checkCollisionWithPowers(), che permette al player di bloccarsi se collide con l’uovo, o di raccogliere un powerup quando l’uovo è stato aperto; tutto ciò avviene consultando la lista di powerups creata dal PowerUpController.
- KillDetection usa checkCollisionWithEnemies(), che permette di identificare 2 tipi di collisione: se il player schiaccia il nemico, lo uccide, se viene toccato da destra o sinistra perde una vita. La posizione dei nemici viene controllata in loop mediante la lista creata da EnemyHandlerImpl.
- CoinDetection usa controlCoinCollision(), che scorre la lista di monete creata da CoinController e quando l’area del player collide con l’area di una moneta viene aggiornato il conteggio delle monete sia in CoinController che in ScoreController.

Problema: creazione dell’entità powerup con meccanica complessa (raccolta powerup solo dopo apertura uovo) e gestione delle collisioni col player.

Soluzione: in questo caso ho sfruttato il pattern architetturale MVC creando l’interfaccia PowerUp che realizza l’entità powerup e la sua implementazione PowerUpImpl nel model del progetto. Nel controller del progetto ho creato la classe PowerUpController con lo scopo di creare la lista dei powerups presenti nelle varie mappe. Infine nella view ho inserito la classe PowerUpManager, che acquisisce la lista di tutti i powerups e si occupa di stamparli a video nel pannello di gioco.

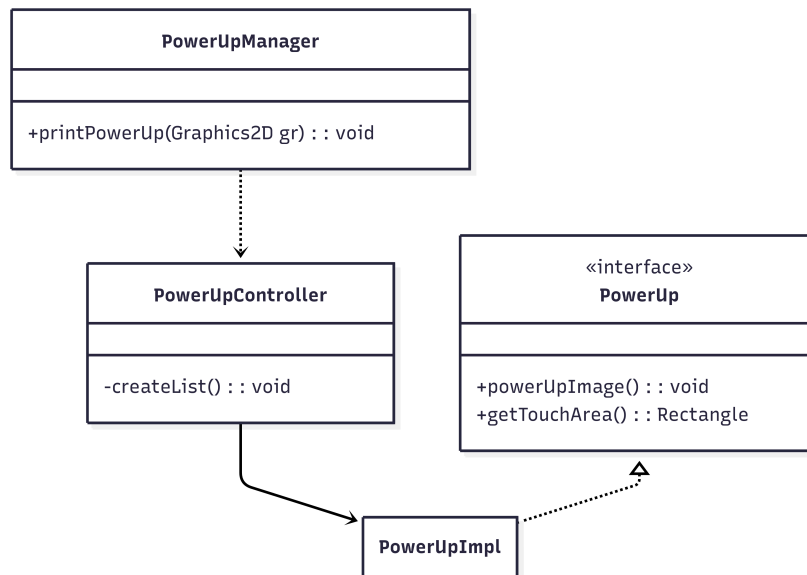


Figura 2.5: UML architettura MVC dei powerup

Problema: gestire il passaggio da una skin all'altra in modo reattivo, in base alle diverse collisioni

Soluzione: creazione della classe PowersHandler nella sezione controller dell'MVC. La classe che gestisce l'inizializzazione del personaggio con cui si inizia il livello e la creazione di nuove istanze Character mantenendole in una lista. Quando il player ottiene 1 vita (raccolta powerup) o la perde (collisione ostacolo o nemico, se non schiacciato) vengono utilizzati i metodi setPowers() e losePowers() dalle classi "Detection"; questi metodi si collegano al Game-LoopController permettendo di cambiare la skin del player attualmente in gioco e di settare la posizione della nuova istanza all'interno della mappa. Inoltre permette di gestire il tempo in cui il player è immortale non appena perde una vita, tramite la variabile lastHit. Infine fornisce il metodo gameOver() che restituisce true se l'entità Character è di tipo "Naked" e perde una vita.

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,

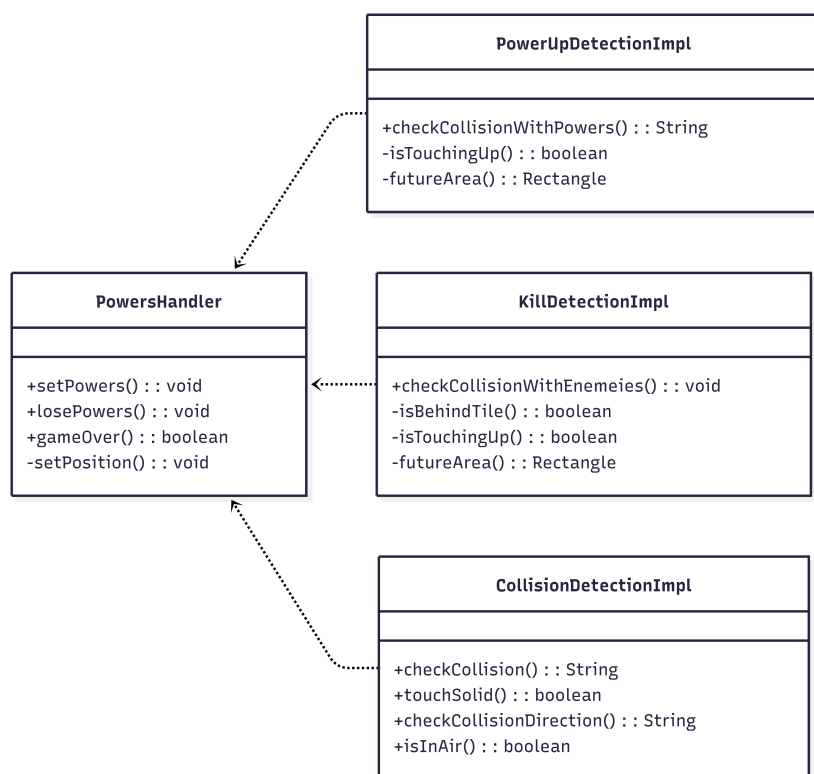


Figura 2.6: UML architettura per gestire cambio skin in base alle collisioni

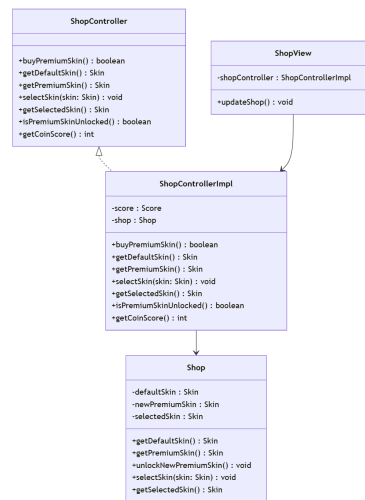


Figura 2.7: UML dell'MVC per la gestione dello shop

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.

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` (implementata 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 aggiungere 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

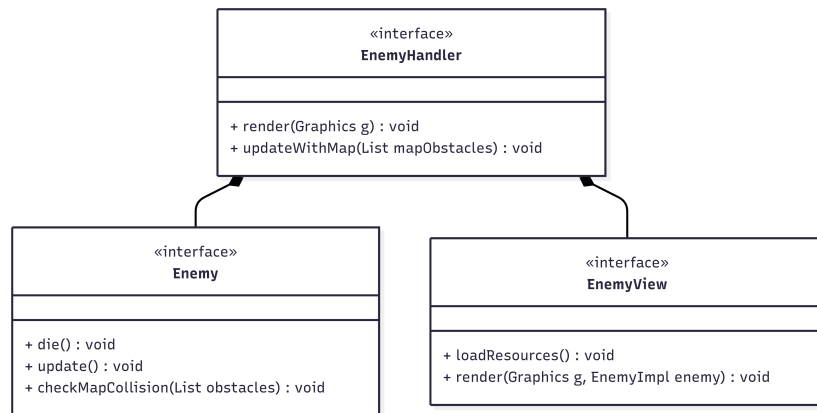


Figura 2.8: UML dell'MVC per la gestione dei nemici

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. 2.8

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`, implementata come singleton tramite il pattern Initialization-on-demand holder idiom, una tecnica che garantisce accesso centralizzato, inizializzazione lazy e sicurezza thread-safe senza costi di sincronizzazione esplicita.¹

All'interno della classe, il caricamento dello stato avviene una sola volta al primo accesso all'istanza, tramite una classe interna statica (Holder) che richiama un metodo privato di inizializzazione. Questa logica garantisce che il file di salvataggio venga caricato una sola volta e che non vengano mai create più istanze del file, evitando duplicazioni o corruzioni dello stato.

Quando il giocatore completa un livello o chiude l'applicazione, i controller — in particolare `ScoreController` e `CoinController` — invocano i metodi pubblici del `GameSaveManager` per aggiornare lo stato di gioco, che comprende:

- Un intero che indica i livelli completati.
- Un intero che indica le monete 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. 2.9

¹Questa implementazione è stata adottata per risolvere un warning PMD, come indicato in [1], seguendo le linee guida descritte in [2].

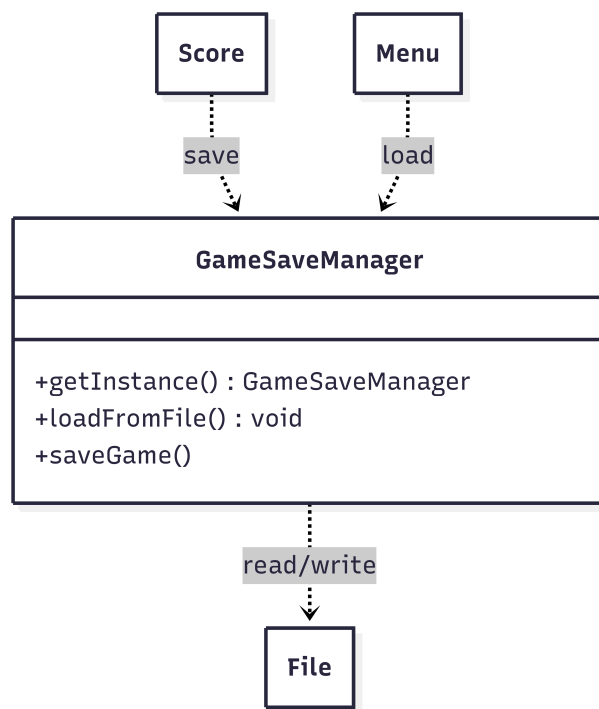


Figura 2.9: UML dell'MVC per la gestione dei nemici

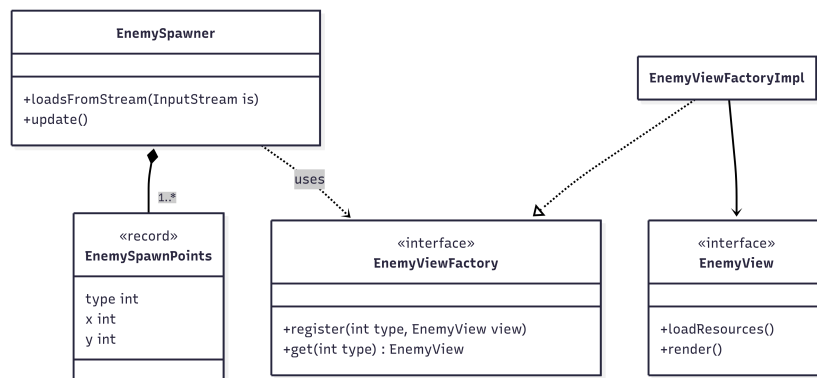


Figura 2.10: UML gestione spawning nemici

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 $1 \rightarrow \text{GuardView}$). 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. 2.10

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 una 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

3.3 Samuele Bianchedi

Una delle scelte di design più importanti del progetto è stata la rigorosa applicazione dei principi di incapsulamento e immutabilità. L'analisi iniziale con SpotBugs ha rivelato criticità rispetto all'esposizione interna. Queste ultime erano potenziali bug difficilmente riconoscibili, in quanto i dati erano vulnerabili a modifiche esterne non controllate. Si è adottata la tecnica delle copie difensive in questi seguenti modi:

- Robustezza: il design incapsulato rende il software meno vulnerabile a bug di side effect, perché lo stato degli oggetti è protetto e controllato.
- Manutenibilità: separando la parte pubblica e privata di una classe contribuisce a facilitare la correzione o l'aggiunta di parti di codice senza “rompere” altre parti del programma.
- Portabilità: L'uso di UTF-8 in modo esplicito garantisce lo stesso comportamento su piattaforme diverse.

Tra le feature avanzate troviamo:

- Uso di Generics: per garantire la type safety nelle collezioni, come `Map<Integer, BufferedImage>` e `List<MapElement>` <https://github.com/GiovanniRava/OOP24-runwarrior/blob/59f5771e783f0430bcbb4859ef986df97121394a/src/main/java/it/unibo/runwarrior/MapElement.java>
- API di I/O: utilizzo di `java.nio.charset.StandardCharset` per specificare l'encoding UTF-8 in lettura <https://github.com/GiovanniRava/OOP24-runwarrior/blob/59f5771e783f0430bcbb4859ef986df97121394a/src/main/java/it/unibo/runwarrior/IO.java>

3.4 Riccardo Cornacchia

3.5 Francesca Gatti

3.6 Giovanni Maria Rava

Commenti finali

4.1.1 Samuele Bianchedi

4.1.2 Riccardo Cornacchia

4.1.3 Francesca Gatti

4.1.4 Giovanni Maria Rava

- 21

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.

Appendice B

Bibliografia

Bibliografia

- [1] PMD Project,
PMD Java Rule: NonThreadSafeSingleton,
https://docs.pmd-code.org/pmd-doc-7.12.0/pmd_rules_java_multithreading.html#nonthreadsafesingleton,
accessed July 2025.

- [2] Wikipedia contributors,
Initialization-on-demand holder idiom — Wikipedia,
https://en.wikipedia.org/wiki/Initialization-on-demand_holder_idiom,
accessed July 2025.