Capitolo 1

Analisi

1.1 Descrizione e requisiti

Il software descritto è un videogioco di azione ispirato a “Vampire Survivors”, che prende il nome di “OOP-Survivors” (abbreviato “OOPS”). Il giocatore interpreta un personaggio in un mondo infestato da entità ostili, con l’obiettivo di sopravvivere più a lungo possibile, eliminando le orde di nemici che compariranno in modo progressivo e incessante. Il gioco è strutturato in sessioni a durata limitata, durante le quali il personaggio guadagna esperienza e può evolvere le proprie abilità e armi. Alla fine della sessione, se il giocatore sarà sopravvissuto, verrà considerato vincitore.

#### Requisiti funzionali

* Il sistema deve generare orde di nemici con frequenza e quantità crescente nel tempo.
* Il personaggio giocante deve potersi muovere liberamente nell’area di gioco e attaccare in modo automatico.
* (Da rivedere) Devono essere presenti più personaggi sbloccabili, ciascuno con caratteristiche e armi iniziali differenti.
* Devono essere presenti vari tipi di nemici, ciascuno con comportamento, velocità e resistenza differente.
* Durante il gioco, il personaggio deve poter salire di livello, scegliendo tra una serie di potenziamenti casuali.
* (Da rivedere) Alla fine di ogni sessione, devono essere mostrati i risultati ottenuti (tempo sopravvissuto, nemici sconfitti, oggetti raccolti).
* (Da rivedere) Deve esistere un sistema di potenziamenti permanenti acquistabili con la valuta ottenuta durante le sessioni.
* (Da rivedere) Il gioco deve salvare i progressi relativi a potenziamenti, personaggi sbloccati e obiettivi raggiunti.

#### Requisiti non funzionali

* Il sistema deve essere in grado di gestire centinaia di nemici simultaneamente a schermo, garantendo fluidità e responsività.
* (Da rivedere) Il gioco deve essere compatibile con i seguenti sistemi operativi: Windows, Linux

1.2 Modello del dominio

Giocatore (Player): rappresenta il personaggio controllato dall’utente. Il giocatore ha varie statistiche come salute, attacco e velocità.

Nemici (Enemies): entità ostili che si muovono verso il giocatore, anch’essi hanno statistiche di salute, attacco, velocità.

Armi (Weapons): attacchi automatici a disposizione del giocatore.

( Da Rivedere) Oggetti (Items): elementi raccoglibili dal giocatore come monete ed esperienza.

Gestore dei Nemici (Enemy Manager): componente responsabile dello spawning e della gestione del ciclo vitale dei nemici nel gioco.

Gestore delle Collisioni (Collision Manager): classe incaricata di rilevare e gestire le interazioni tra il giocatore, i nemici e altri oggetti di gioco, come i proiettili, fondamentali per la gestione di eventi come danni( o raccolta di oggetti (adesso viene fatto dall’ experience manager)).

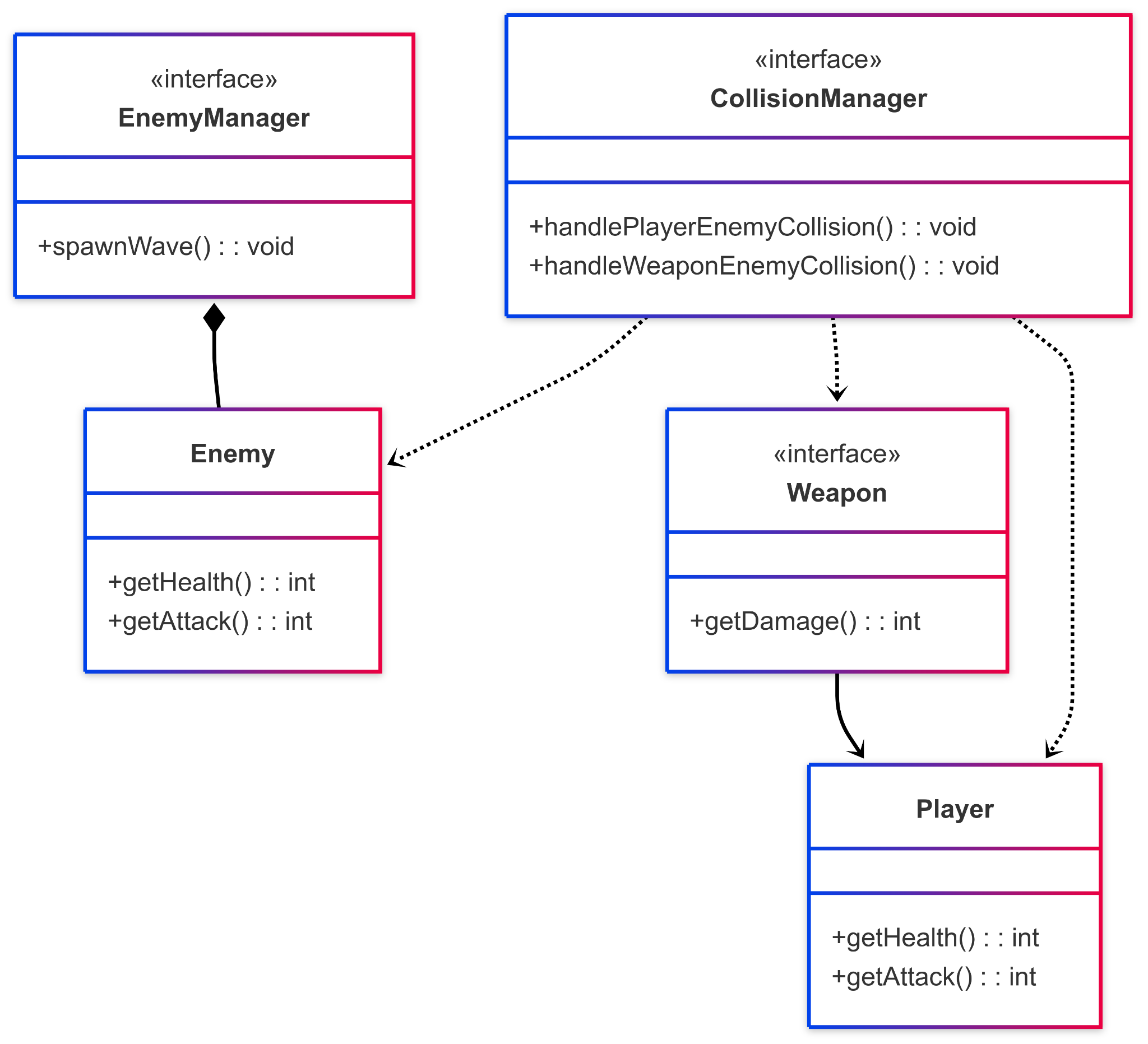


Figura 1.1: Schema UML dell’analisi del problema, con rappresentate le entità principali legate alla realizzazione dell’applicazione ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura (Nota le classi camera e Input/Mouse Handler dovrebbero essere cambiate rispettivamente in Model e Controller)

L’architettura del progetto segue il pattern architetturale Model-View-Controller (MVC). Più nello specifico, il *controller* agisce come componente centrale attivo del sistema: possiede riferimenti alle principali classi del *model* e della *view*, e coordina l’interazione tra le due.

Il *model* è costituito da più classi che rappresentano le entità del dominio di gioco, ciascuna delle quali incapsula dati e logica specifici. Queste classi non dipendono né conoscono la view, rispettando così l’indipendenza del modello dalla presentazione.

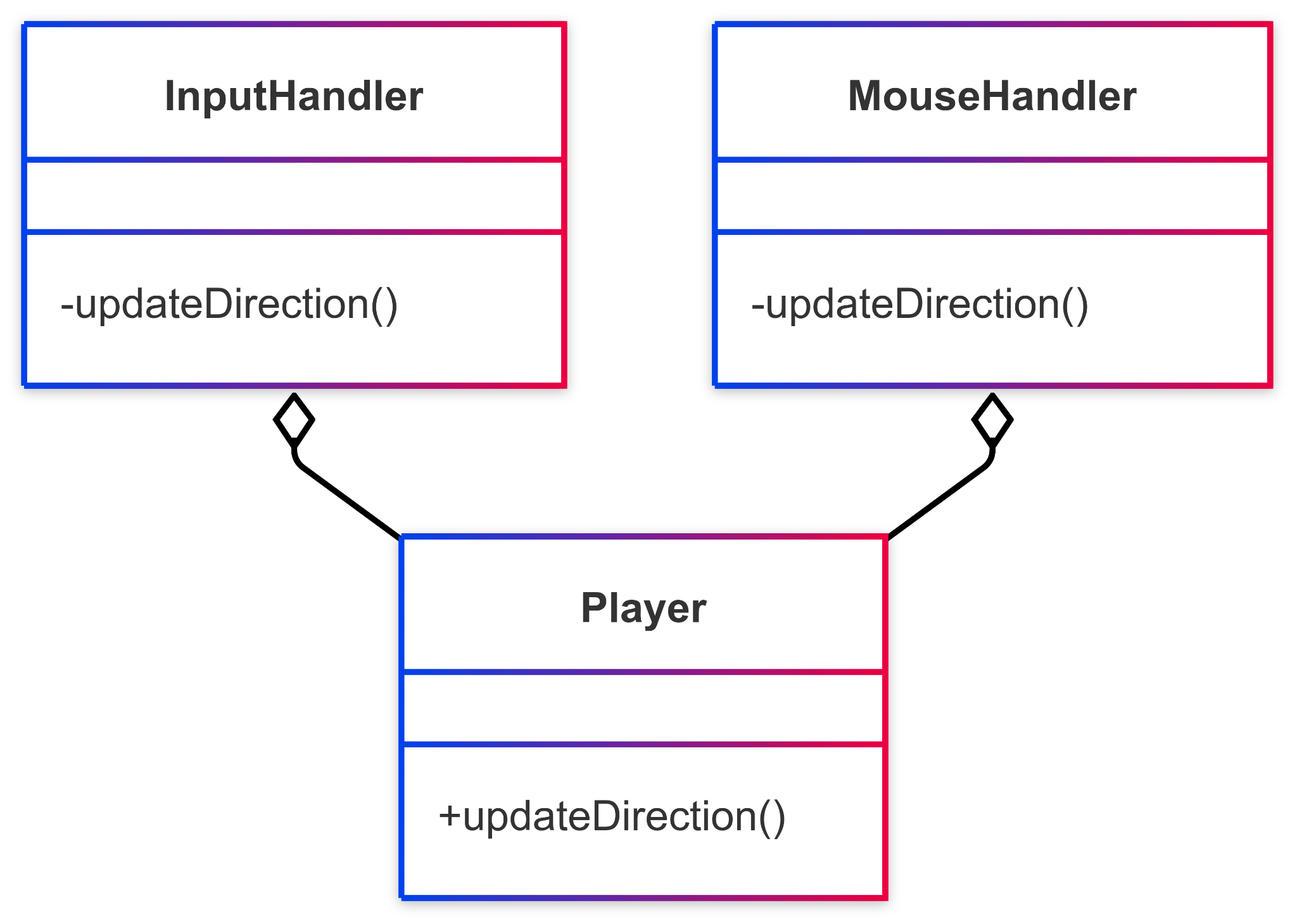
La *view*, a sua volta, è composta da diversi pannelli e componenti grafici, responsabili dell’interfaccia utente.

Il *controller* principale GameThread prende in input un pannello della *view*, in base ad esso cambia parti di *model* diverse ed aggiorna il pannello. Preso per esempio il GamePanel, il GameThread aggiornerà classi come Player ed EnemyManager, e chiamerà alcuni dei suoi metodi come checkCollisions(), infine notificherà la *view* che aggiornerà la rappresentazione grafica.

Con questa struttura, la sostituzione della *view* (ad esempio passando da Swing a JavaFX) o del model con una nuova implementazione non richiederebbe modifiche né al controller né all’altra componente, a condizione che vengano rispettate le stesse interfacce.

In alcune classi del *model* sono presenti degli observer che notificano i cambiamenti al *controller*.

Qualsiasi aggiornamento della *view* viene effettuato esclusivamente all'interno dell’Event Dispatch Thread (EDT) di Swing, garantendo così che l’applicazione sia thread-safe.



(Da Rivedere) Fig. 2.1: InputHandler e MouseHandler (classi *Controller*), in base ad un input rispettivamente da tastiera e da mouse, cambiano la direzione in cui il player si muove e/o guarda.

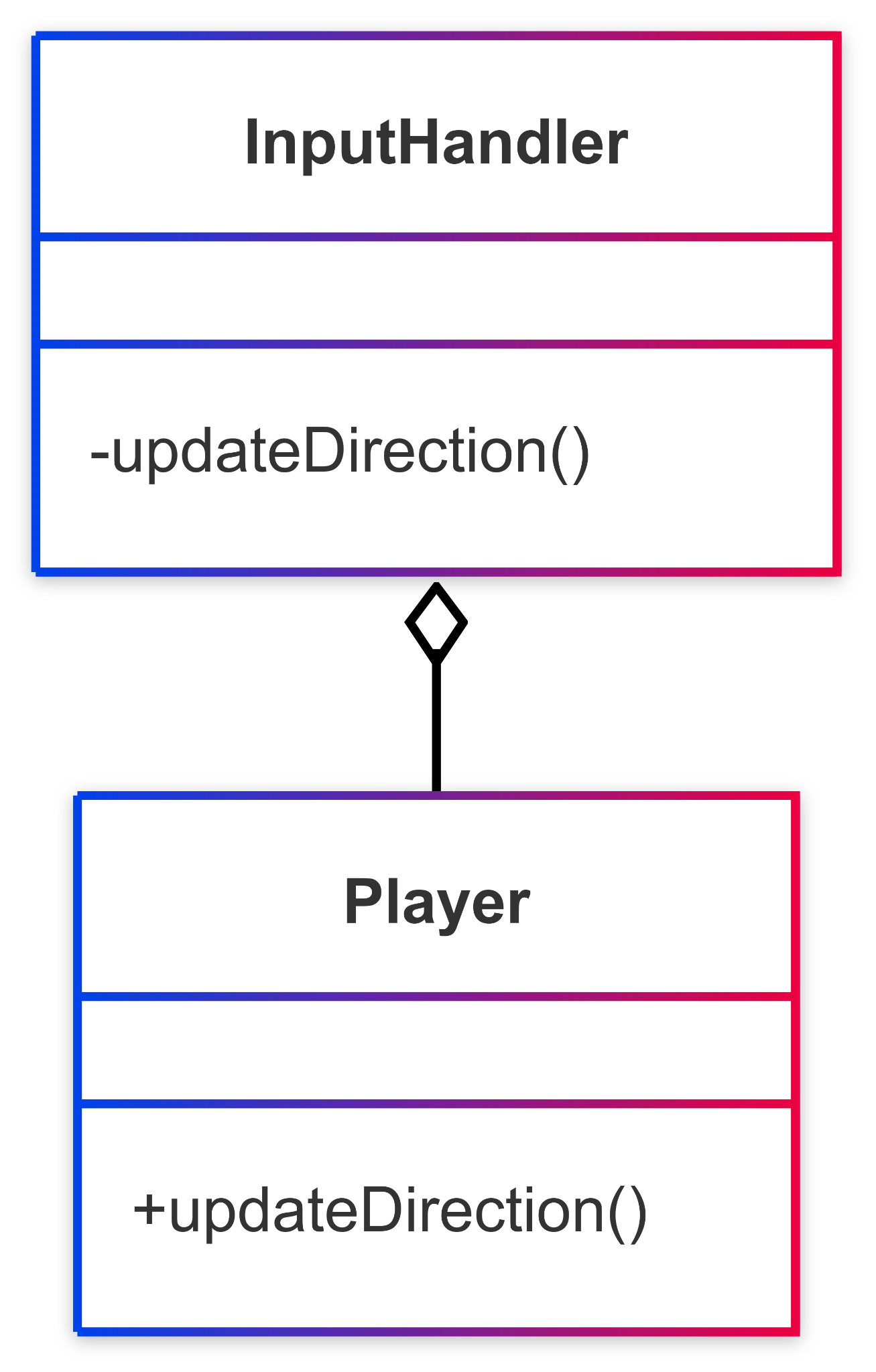


Fig 2.1: InputHandler (classe *Controller*), in base ad un input da tastiera, cambia la direzione in cui il player si muove.

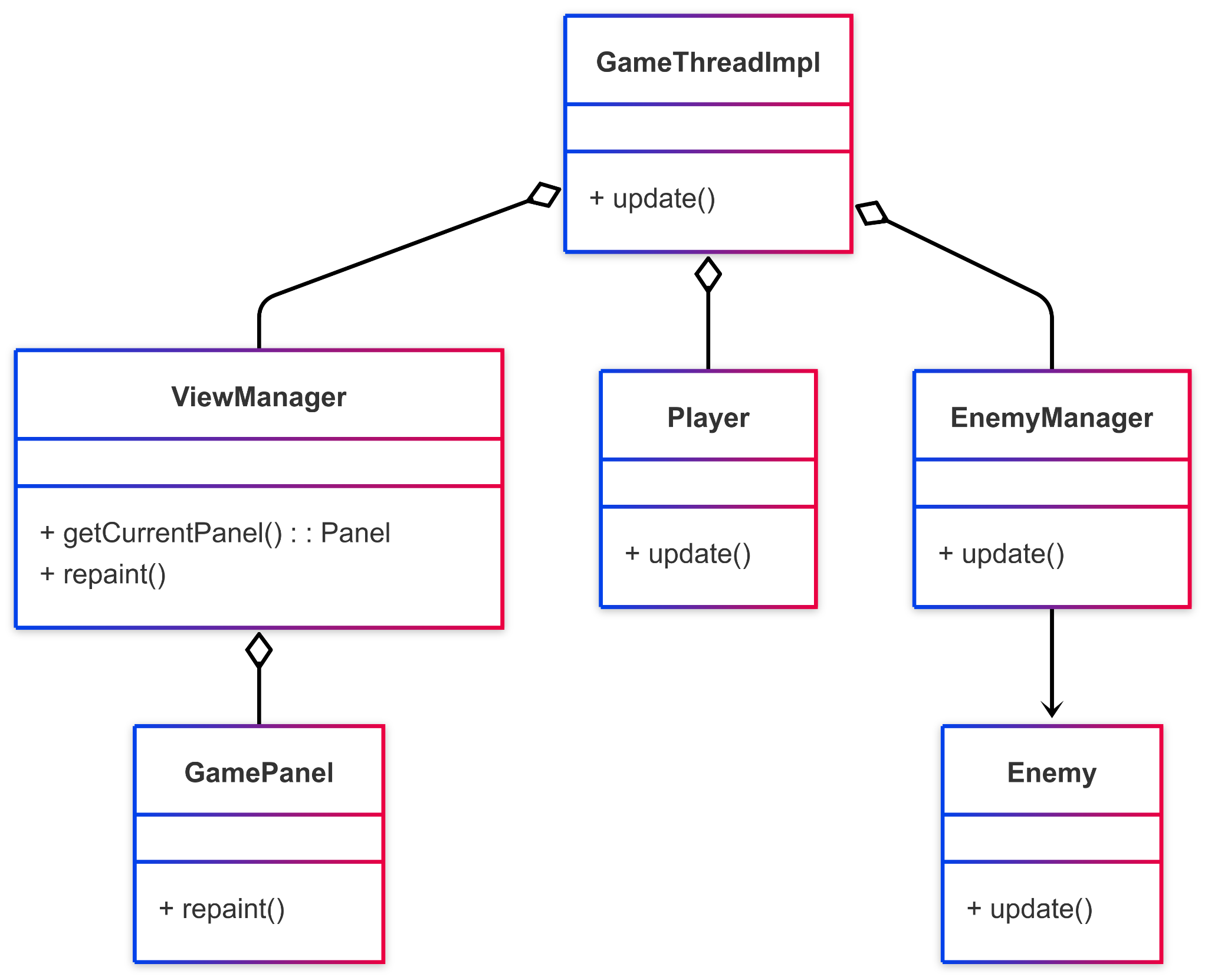


Fig 2.2: Il GameThreadImpl (*controller* principale dell’applicazione) ottiene in input il pannello corrente tramite il ViewManager e, in base ad esso esegue un aggiornamento delle principali classi del *model* e richiama il metodo repaint() sulla *view* per aggiornare la rappresentazione grafica.

2.2 Design Dettagliato

Sezione 1

**Sezione 2** - Giovanni Perreon

- Aggiungere comportamenti ai nemici senza creare

molte sottoclassi

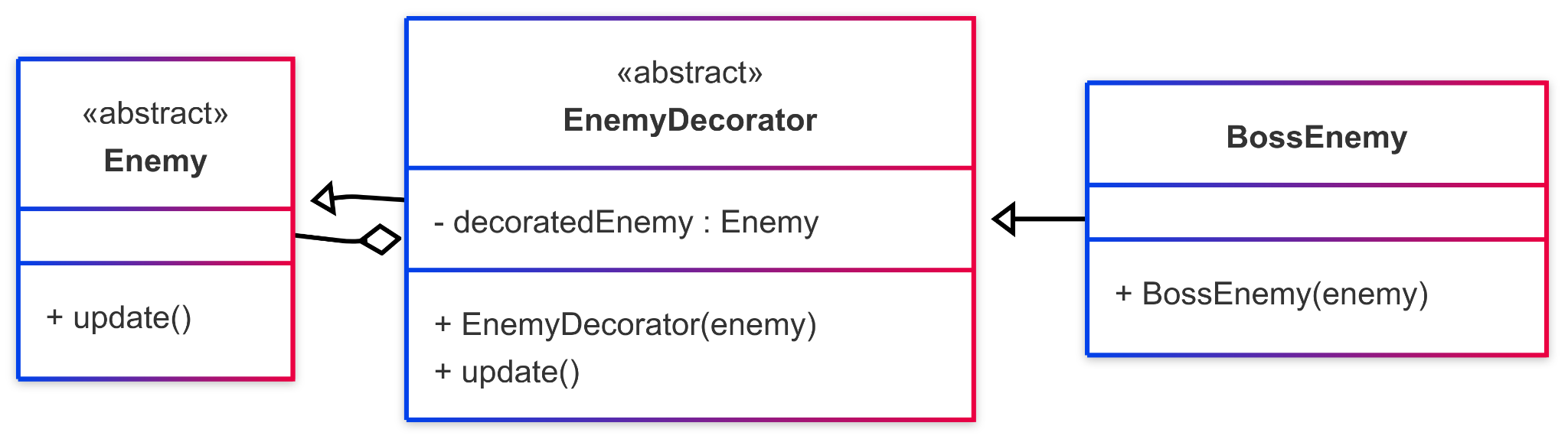


Fig 2.5: Rappresentazione UML del pattern Decorator per aggiungere caratteristiche ai nemici.

**Problema:** Per creare un nemico con una particolare caratteristica bisogna creare una sottoclasse di quel nemico, e quando mi servono molti nemici con caratteristiche diverse devo creare altrettante classi includendo anche tutte le combinazioni.

**Soluzione:** Il pattern Decorator permette di aggiungere dinamicamente caratteristiche o comportamenti ad un nemico.

Ad esempio, la classe **BossEnemy** estende **EnemyDecorator** e raddoppia grandezza, vita e attacco del nemico decorato.

Questo approccio consente di combinare più decoratori tra loro (anche più volte), ottenendo tutte le varianti desiderate senza moltiplicare le classi.

- Notifica degli eventi senza passaggio di variabili

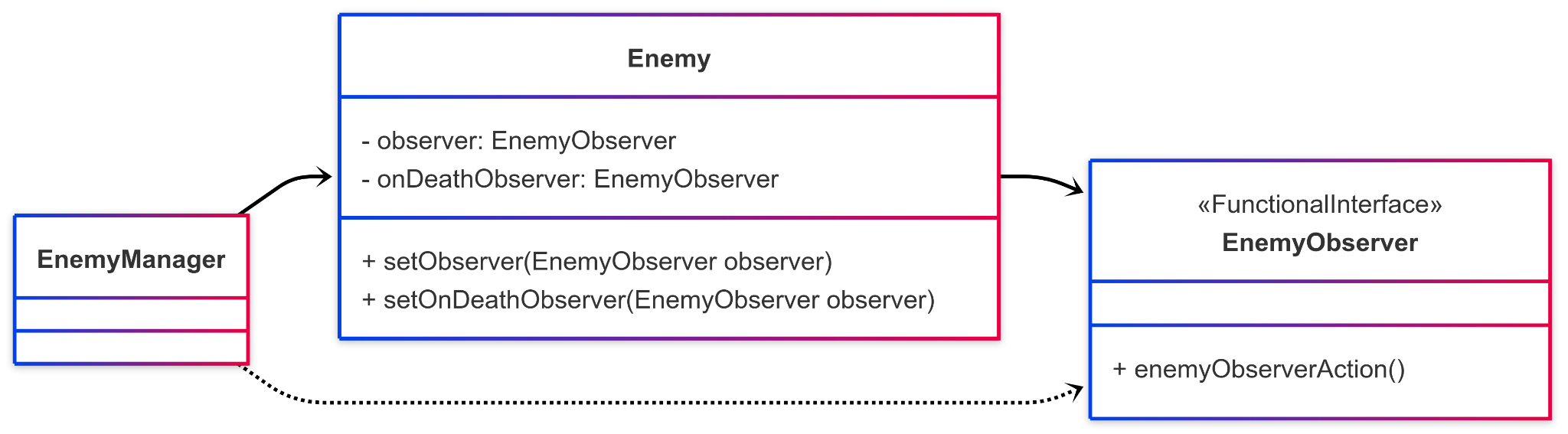


Fig. 2.6: Rappresentazione UML del pattern Observer, in cui Enemy agisce come Subject notificando le implementazioni dell'interfaccia EnemyObserver. Tali implementazioni sono fornite sotto forma di espressioni lambda, implementate da EnemyManager.

**Problema:** Far sì che un nemico possa notificare un evento senza dover gestire o mantenere variabili aggiuntive.

**Soluzione:** Uso del pattern Observer, in cui **Enemy** funge da Subject e usa degli observer che implementano l’interfaccia **EnemyObserver**. Le implementazioni concrete di questi observer sono espressioni lambda implementate dinamicamente da **EnemyManager**. In questo modo, **Enemy** non deve conservare variabili aggiuntive specifiche per ogni osservatore.

Sezione 3

Sezione 4