# Relazione per “Programmazione ad oggetti”

Filippo Patrignani
Giovanni Perreon
Sofia Ricupero

22/06/2025

# 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.
- Devono essere presenti vari tipi di nemici, ciascuno con comportamento, velocità e resistenza differente.
- Durante il gioco, il personaggio deve poter salire di livello, ottenendo un potenziamento casuale.

### Requisiti non funzionali

- Il sistema deve essere in grado di gestire centinaia di nemici simultaneamente a schermo, garantendo fluidità e responsività.

- 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.

Oggetti (Items): elementi raccoglibili dal giocatore come (es. orb di esperienza).

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

Gestore delle Collisioni (CollisionManager): 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

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 ed entra a contatto con informazioni di *model* solamente tramite classi *controller* apposite.

Il *controller* principale è **GameThreadImpl**, che 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 per lavorare con loro come **checkCollisions()**, infine dirà alla *view* di aggiornare 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 essa sia thread-safe.

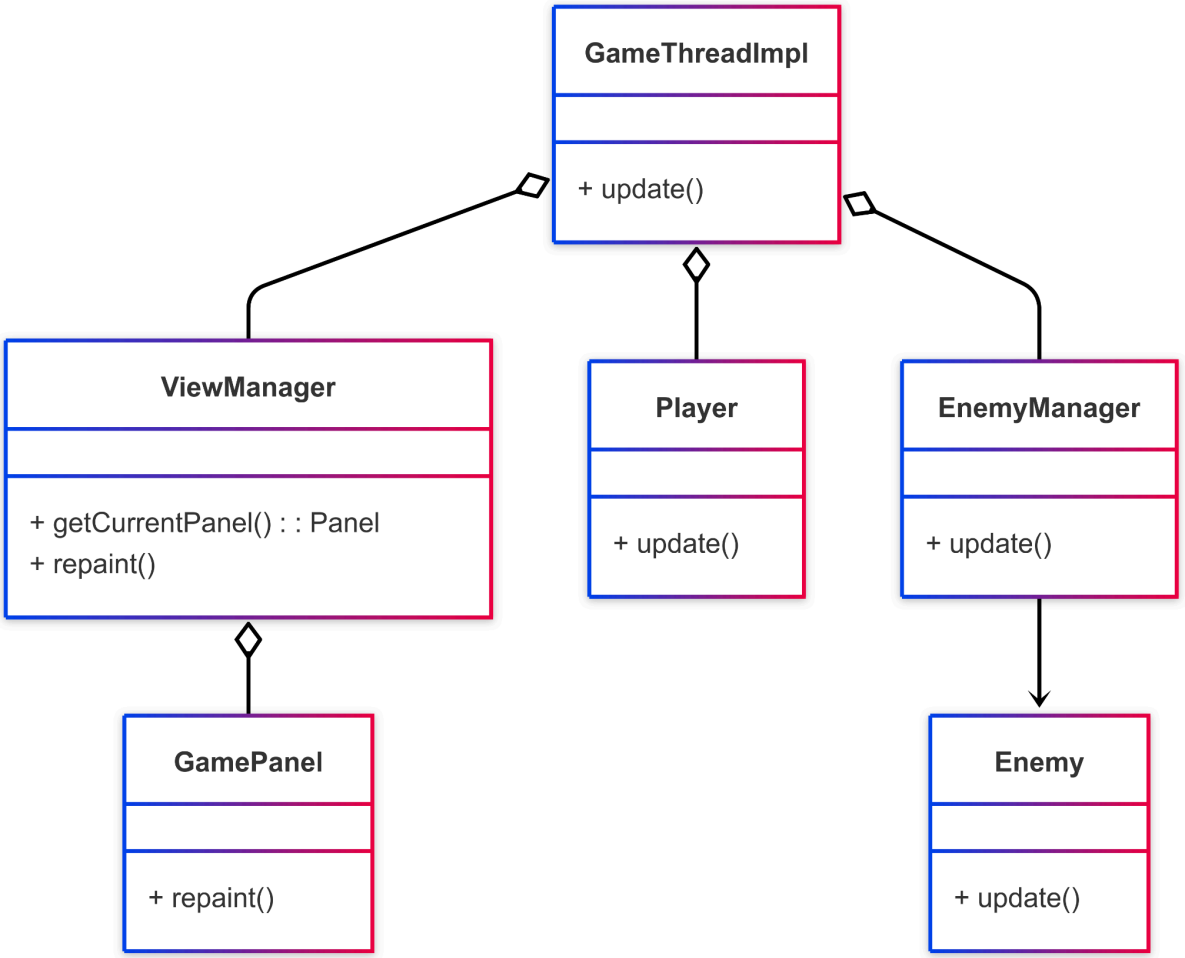


Fig 2.2: Il GameThreadImpl (*controller* principale dell'applicazione) ottiene in input il pannello corrente tramite il ViewManager e, in base ad esso esegue un aggiornamento delle principali classi del *model* e richiama il metodo repaint() sulla *view* per aggiornare la rappresentazione grafica.

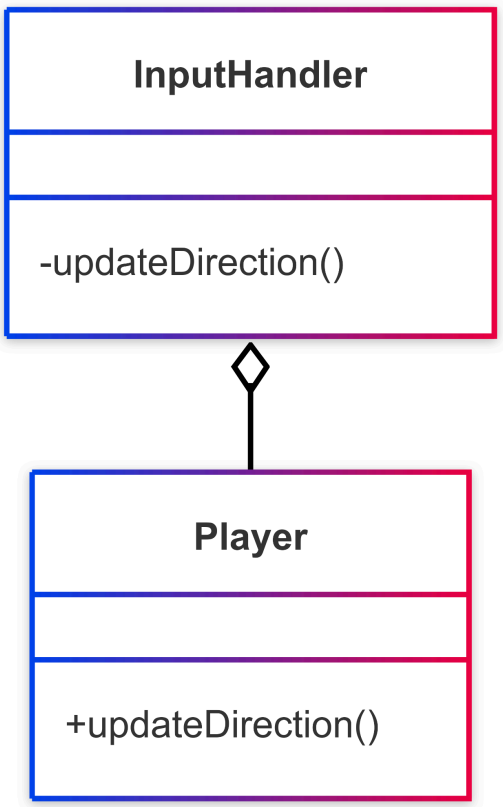


Fig 2.1: InputHandler (classe *Controller*), in base ad un input da tastiera, cambia la direzione in cui il player si muove.

## 2.2 Design Dettagliato

**Sezione 1** - Giovanni Perreon

- Aggiungere comportamenti ai nemici senza creare molte sottoclassi

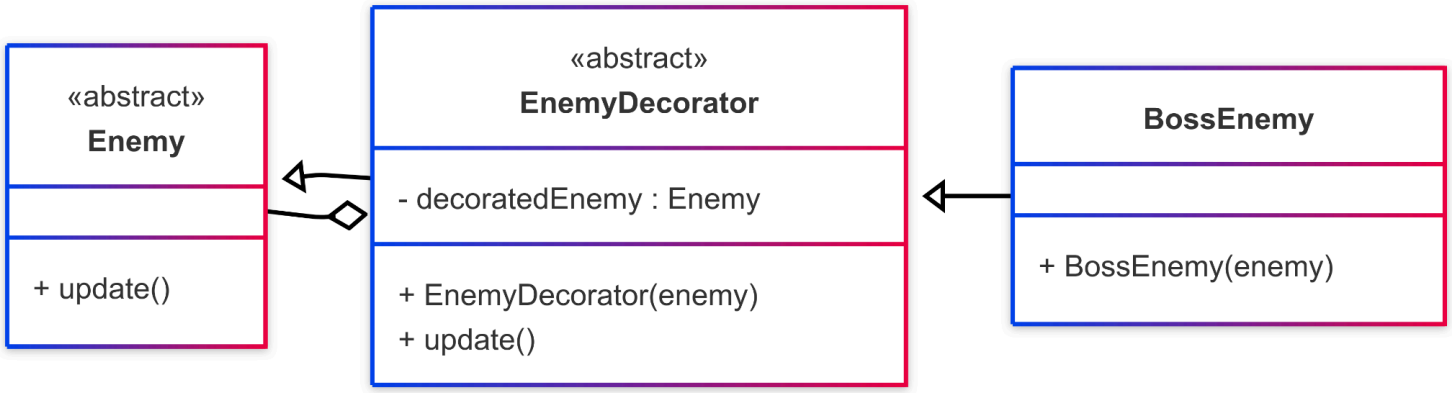


Fig 2.5: Rappresentazione UML del pattern Decorator per aggiungere caratteristiche ai nemici.

**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

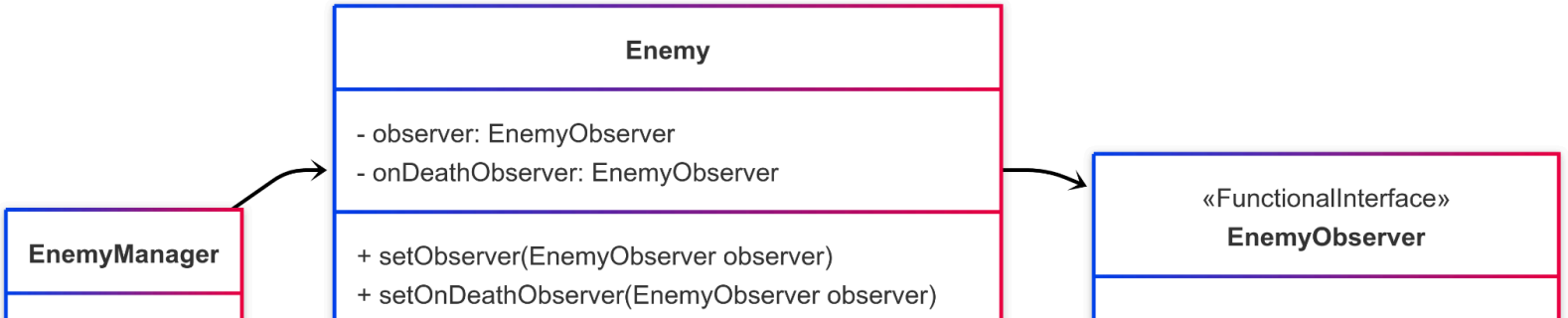

Fig. 2.6: Rappresentazione UML del pattern Observer, in cui Enemy agisce come Subject notificando le implementazioni dell'interfaccia EnemyObserver. Tali implementazioni sono fornite sotto forma di espressioni lambda, implementate da EnemyManager.

**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.

- Definizione di un'interfaccia per la creazione di oggetti, delegando alle implementazioni la decisione su quale classe concreta istanziare e in che modo.

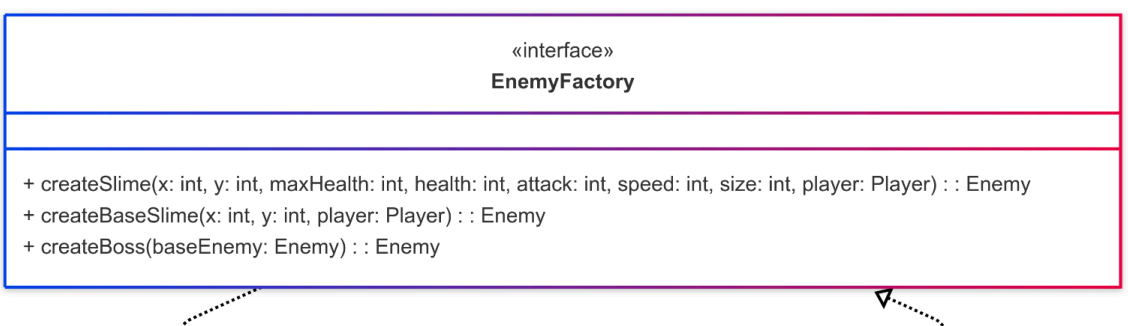

Fig. 2.7: Rappresentazione UML del pattern Factory Method, dove **EnemyFactory** è l'interfaccia che definisce metodi di creazione di **Enemy**, implementato da **EnemyFactoryImpl**, che restituisce un'istanza concreta come **Slime**.

**Problema:** È necessario creare oggetti appartenenti a una gerarchia di classi (in questo caso diversi tipi di nemici), ma non si vuole dipendere direttamente dalle classi concrete.
Questo rende il codice rigido e difficile da estendere: ogni volta che si vuole aggiungere una nuova variante di nemico, bisogna modificare il

codice esistente, aumentando il rischio di errori e riducendo la flessibilità.

**Soluzione:** Implementazione del pattern Factory Method tramite l'interfaccia **EnemyFactory** che dichiara i metodi per creare diversi tipi di nemici, mentre **EnemyFactoryImpl** fornisce le implementazioni specifiche restituendo istanze delle sottoclassi concrete come **Slime**. In questo modo, la logica di creazione è separata dal resto del programma: chi usa la factory può ottenere oggetti **Enemy** senza conoscere i dettagli delle classi concrete o dei costruttori. Per aggiungere un nuovo tipo di nemico sarà sufficiente creare una nuova classe e aggiungere due metodi nella factory, senza dover modificare tutto il codice che istanzia nemici.
Ogni classe nemico, infine, espone una Static Factory che permette di ottenere una configurazione predefinita del nemico.

## **Sezione 2** - Filippo Patrignani

- Differenziare gli accessori e le armi

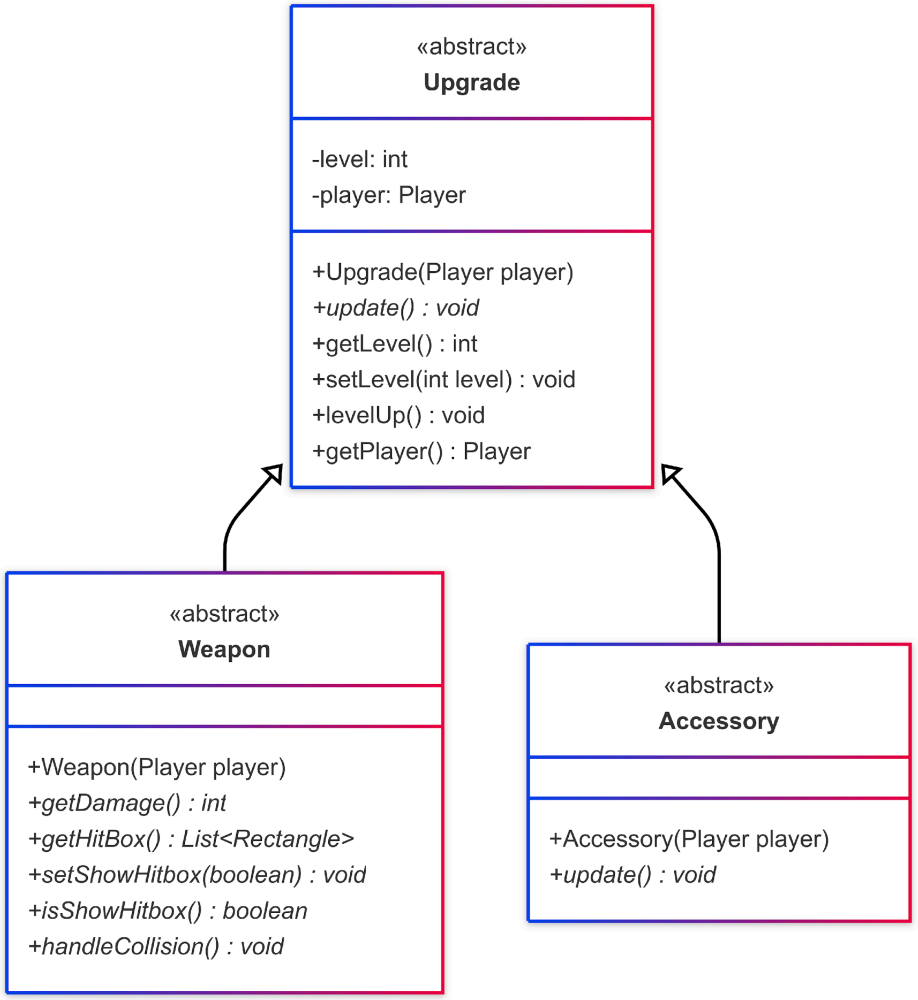


Fig. 2.8: Rappresentazione UML della gerarchia utilizzata per gli upgrades ottenuti dal player all'aumento di livello.

**Problema:** Inizialmente avevo incluso gli accessori nelle armi e trattati come esse, però poi ho dovuto creare una superclasse a causa delle differenze nell'implementazione della loro logica (gli accessori non vanno disegnati, non hanno hitbox e non hanno collisioni).

**Soluzione:** Far sì che siano due estensioni di una superclasse, in modo che siano entrambi considerati un upgrade da estrarre ad ogni aumento di livello, ma facendoli trattare in modo diverso dalle altre classi (CollisionManager, WeaponManager...).

- Aggiungere una meccanica post collisione al proiettile della Magic Staff, senza creare un metodo apposito nel CollisionManager, e senza passare all'arma o al proiettile nessuna referenza dei Manager che gestiranno la collisione

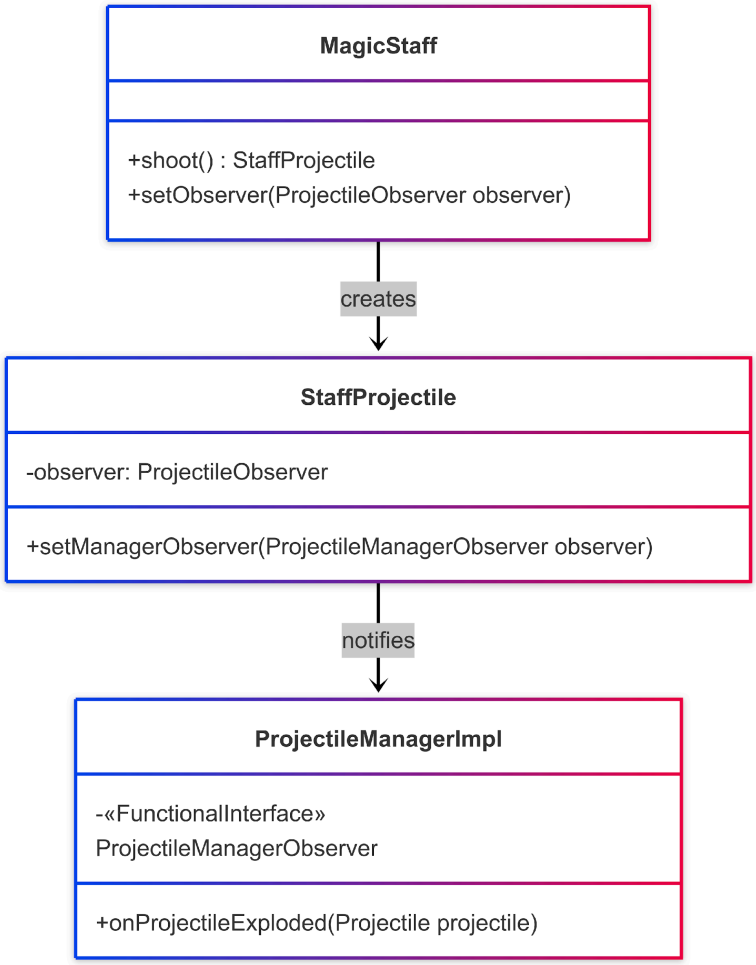


Fig. 2.9: Rappresentazione UML del pattern Observer, in cui StaffProjectile agisce come Subject, su cui verrà settato un observer per l'esplosione alla creazione MagicStaff, e uno per la collisione da ProjectileManagerImpl, così che il proiettile possa notificare la sua esplosione nel suo handleCollision().

**Problema:** Far sì che il proiettile esegua la sua meccanica specifica, senza creare metodi extra nel collision manager o nella classe

dell'arma stessa, senza aver a disposizione una referenza del CollisionManager o del ProjectileManager da cui chiamare metodi.

**Soluzione:** Uso del pattern Observer, in cui StaffProjectile funge da Subject, mentre le implementazioni concrete di questi observer sono espressioni lambda implementate dalla classe dell'arma alla creazione di ogni proiettile. In questo modo non devono essere creati metodi aggiuntivi per gestire la collisione di ogni proiettile con effetto nel collision manager o nel projectile manager, consentendo di usare la stessa strategia per altre armi con effetto.

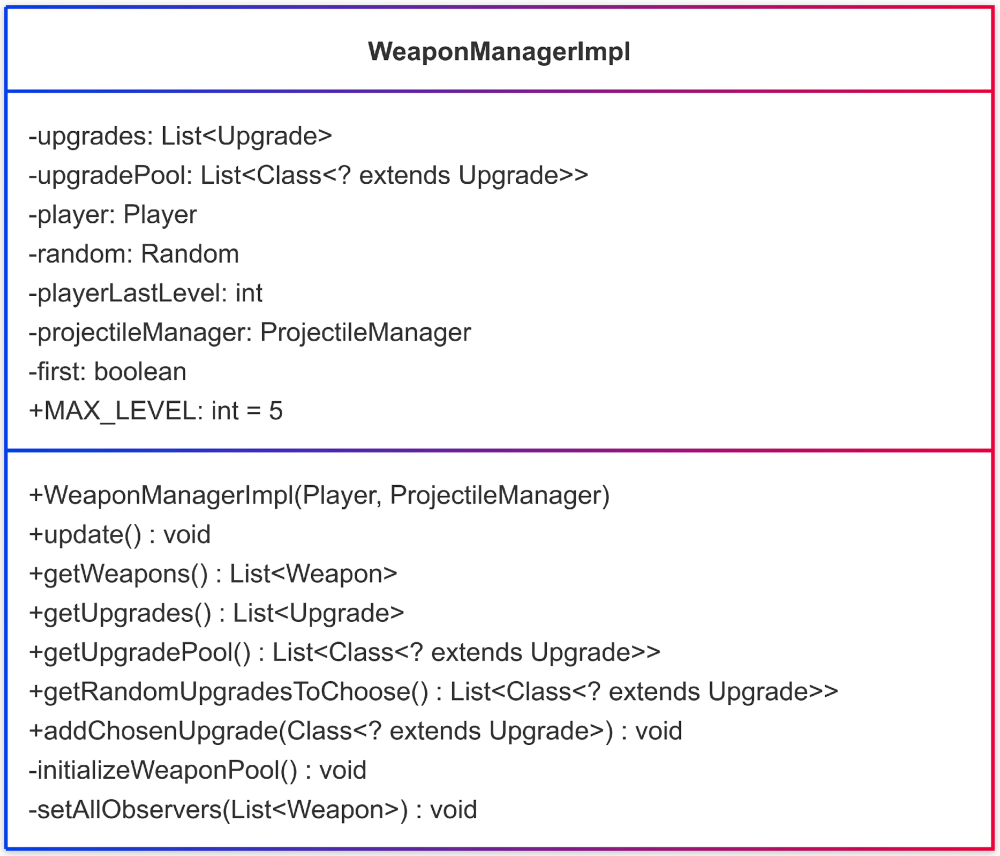


Fig. 2.9: Rappresentazione UML della classe centrale della parte del progetto che riguarda le armi e gli accessori del personaggio.

**Problemi:** Far sì che per ogni Upgrade ci sia una sola istanza di esso, nel momento in cui viene aggiunto agli Upgrades del Player, e far sì

che in una futura implementazione del sistema di scelta multipla tra upgrades random, vengano passate da WeaponManager delle classi e non delle istanze come veniva fatto inizialmente; inoltre dare un modo alle armi che ne avessero bisogno in futuro, di notificare eventi senza passargli un riferimento ad alcun Manager.

**Soluzione:** La pool inziale di armi contiene solo le classi, non delle istanze, e sempre solo classi vengono restituite all'estrazione randomica, poi sarà il compito del metodo addChosenUpgrade() di istanziare l'arma o di aumentarne il livello, inoltre tramite il pattern Observer le armi che ne hanno bisogno diventano Subject di un observer e potranno notificare il manager tramite esso.

## **Sezione 3** – Sofia Ricupero

-Gestione flessibile della pausa tramite notifica di eventi da input

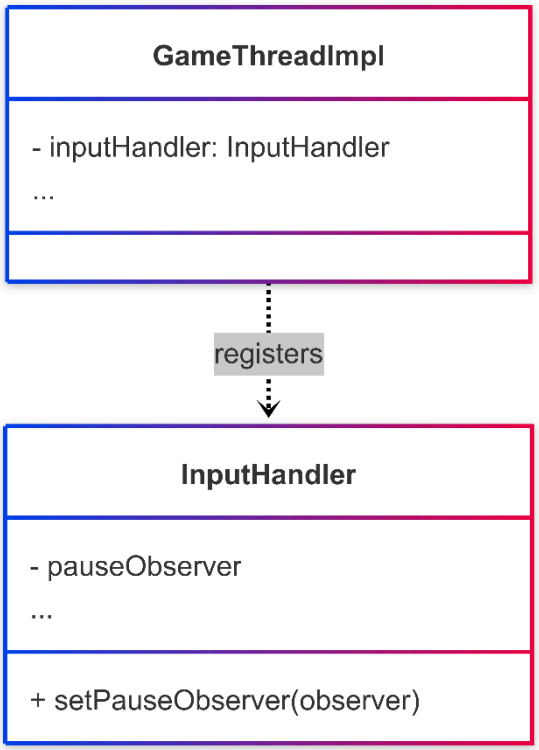


Fig. 3.0: Rappresentazione UML del pattern Observer, in cui InputHandler funge da Subject e notifica la pressione del tasto di pausa. L'implementazione del pattern è realizzata tramite una lambda, che definisce il comportamento associato alla pausa.

**Problema:** Permettere che la pressione di un tasto specifico, in questo caso 'P' per la pausa del gioco, generi un comportamento applicativo, senza che il componente che gestisce l'input debba conoscere la logica della pausa o mantenere riferimenti a classi esterne.

**Soluzione:** Utilizzando il pattern Observer, il modulo di gestione dell'input espone un metodo per registrare un osservatore (nel caso specifico, un oggetto Runnable). Alla pressione del tasto P, l'osservatore viene notificato tramite l'esecuzione della lambda precedentemente fornita dal componente di controllo del gioco.

Questo consente di disaccoppiare completamente la rilevazione dell'input dalla logica applicativa, rendendo il sistema più estendibile e manutenibile.

- Unificazione della logica per i pannelli di opzioni e pausa

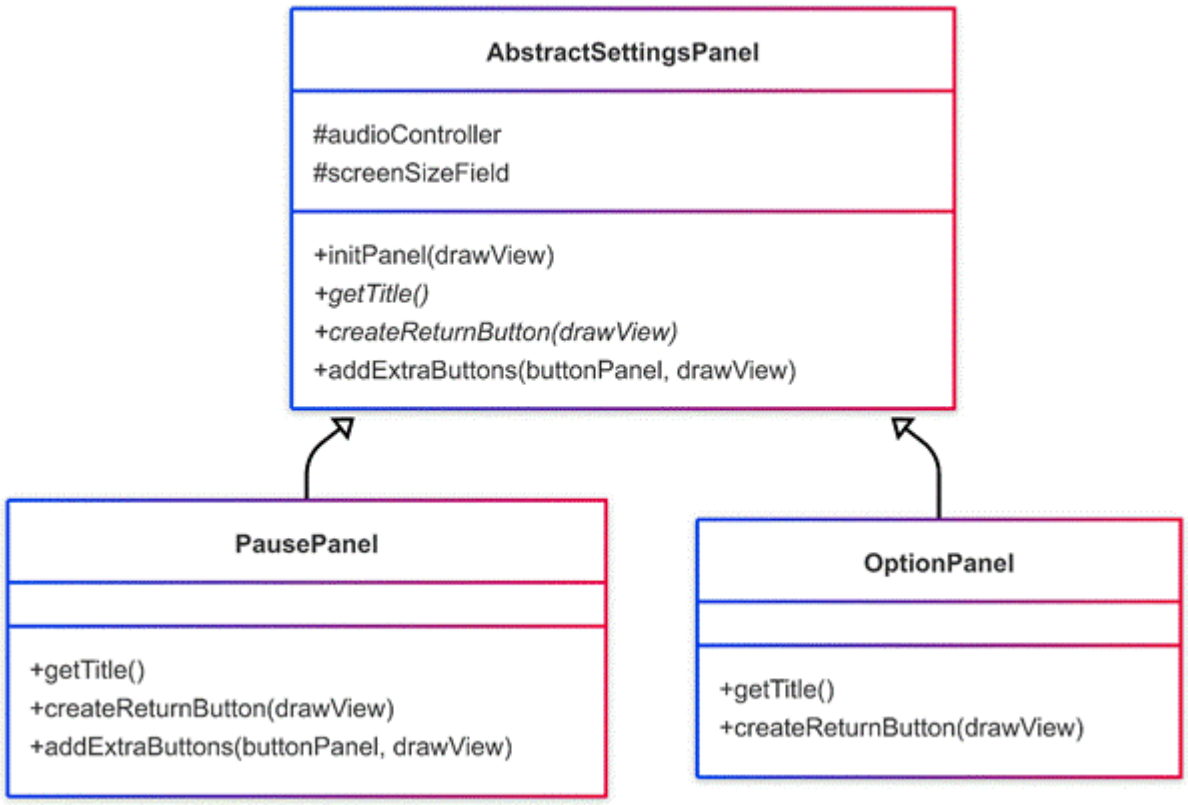


Fig. 3.1: Rappresentazione UML della gerarchia dei pannelli di pausa e impostazioni.

**Problema:** Evitare la duplicazione di codice nella realizzazione di due pannelli grafici che condividono alcune parti, ma hanno componenti specifici distinti, ovvero il pannello di pausa e il pannello delle impostazioni. L'implementazione separata avrebbe comportato una significativa duplicazione di codice e una minore manutenibilità.

**Soluzione:** Adottando il pattern Template Method, si utilizza la classe astratta AbstractSettingPanel per implementare la logica e la struttura comune. All'interno di questa classe vengono definiti metodi astratti che le sottoclassi (PausePanel e OptionPanel) implementano per

personalizzare gli elementi che differenziano i due pannelli, come il titolo e i pulsanti specifici. In questo modo si riutilizza il codice condiviso e si mantiene la flessibilità per le personalizzazioni necessarie.

# Capitolo 3

# Sviluppo

## 3.1 Testing automatizzato

Per il testing automatizzato sono state sviluppate delle classi di test utilizzando il framework **JUnit**.
Le classi che sono state sottoposte a Testing sono:
**Player, EnemyManager, EnemyFactory, CollisionManager, ProjectileManager, ExperienceManager e WeaponManager.**

### **Sezione 1 -** Giovanni Perreon

- TestEnemyManager

Utilizzo di test JUnit per verificare le seguenti funzionalità:

- Aggiungere un nemico alla wave.
- Spawning di un nemico ignorando i limiti massimi dei nemici
- Quando si cercano di inseriscono troppi nemici nella wave, questi vengono ignorati

Inoltre, è stata utilizzata una classe **TestableEnemyManager**, con lo scopo di mantenere alcuni metodi protected.

- TestEnemyFactory

- Controlla che la Factory crei i nemici corretti.

## **Sezione 2 -** Filippo Patrignani

- TestCollisionManager

I test presenti utilizzano JUnit per verificare le funzionalità principali del manager delle collisioni.
I test controllano:

- Se la collisione di due rettangoli viene rilevata correttamente.
- Se la collisione arma/nemico registra il danno correttamente.
- Se la collisione proiettile del player/nemico registra il danno correttamente.
- Se la collisione proiettile del nemico/player registra il danno correttamente.

- TestWeaponManager

I test presenti utilizzano JUnit per verificare le funzionalità principali del manager delle armi.
I test controllano:

- La corretta inizializzazione della pool di upgrades.
- Il corretto funzionamento dell'aggiunta di upgrade ogni level up.
- La corretta gestione della rimozione di ogni arma al livello masssimo.
- La corretta assegnazione degli observers delle armi.

- TestExperienceManager

I test presenti utilizzano JUnit per verificare le funzionalità principali del manager dell'esperienza.
I test controllano:

- La corretta generazione degli orb di esperienza.

- La corretta raccolta degli orb dell'esperienza e il conseguente aumento di esperienza del player.

- TestProjectileManager

I test presenti utilizzano JUnit per verificare le funzionalità principali del manager dei proiettili.
I test controllano:

- La corretta aggiunta ed eliminazione dei proiettili del player e dei nemici dalla lista e il corretto update dello stato e della posizione dei proiettili.

## **Sezione 3** - Sofia Ricupero

- TestPlayer

I test presenti utilizzano JUnit per verificare le funzionalità principali del modello del giocatore.
I test controllano:

- Il corretto aggiornamento della posizione del giocatore in seguito al suo movimento.
- La gestione della salute del giocatore, assicurando che non scenda mai sotto lo zero.
- Il comportamento del giocatore quando la salute è pari a zero, verificando che venga considerato non più vivo.

# 3.2 Note di Sviluppo

## **Sezione 1 -** Giovanni Perreon

**Utilizzo di *Optional:***

Permalink di un optional per gestire la presenza di observer sui nemici:
https://github.com/COLTELLINO/OOP24-OOPS/blob/a9e6eccdb31cdeb0f99cc1de134f8c3d2d5860c0/src/main/java/it/unibo/oop/model/entities/Enemy.java#L130-L138

**Utilizzo di *Stream* e *lambda expressions:***
Utilizzo frequente di lambda expressions di cui sono riportati solo alcuni esempi.
Permalink di una lambda expression (method reference):
https://github.com/COLTELLINO/OOP24-OOPS/blob/a9e6eccdb31cdeb0f99cc1de134f8c3d2d5860c0/src/main/java/it/unibo/oop/model/managers/EnemyManagerImpl.java#L54
Permalink di un metodo che usa stream e lambda expressions:
https://github.com/COLTELLINO/OOP24-OOPS/blob/8f925d368461dd04de60444217eeae314bfe17f2/src/main/java/it/unibo/oop/model/managers/EnemyManagerImpl.java#L139-L144

**Utilizzo Libreria *javax.sound.sampled.*:***
Permalink dei metodi che meglio rappresentano l'utilizzo della libreria per la cattura e gestione dell'audio:
https://github.com/COLTELLINO/OOP24-OOPS/blob/6fd0fa656b87e97c0f3b13f0acf4f536a1999e8e/src/main/java/it/unibo/oop/model/managers/AudioManagerImpl.java#L122-L143

## **Sezione 2 -** Filippo Patrignani

**Utilizzo di *Stream* e *Lambda expressions*:**

Utilizzate per scorrere l'intera lista di upgrades del player e verificare la presenza di quello garantito all'aumento di livello confrontando le classi.

Permalink:
https://github.com/COLTELLINO/OOP24-OOPS/blob/3275e52721936e275dcd75ac81f157834da53d9c/src/main/java/it/unibo/oop/model/managers/WeaponManagerImpl.java#L196

## **Sezione 3** - Sofia Ricupero

**Utilizzo di *lambda expressions* per la gestione degli input:**

Utilizzate per registrare e notificare l'osservatore della pausa. L'input handler notifica tramite una lambda quando viene premuto il tasto 'P'.

Permalink:
https://github.com/COLTELLINO/OOP24-OOPS/blob/48ec0494c13d12d7438b123563a0953a5f98805d/src/main/java/it/unibo/oop/controller/InputHandler.java#L41C1-L44C10

# Capitolo 4

# Commenti finali

## 4.1 Autovalutazione e lavori futuri

### **Sezione 1 -** Giovanni Perreon

Classi analizzate:
EnemyManagerImpl:
La classe è generalmente ben strutturata e dotata di un'efficace implementazione del pattern **Observer**, questo finché non è stato aggiunto il **metodo** che si occupa della logica di spawn, determinando il momento e il tipo di nemici da generare, nonostante l'uso della **Factory, Stream** possa aiutare, è necessario cambiare drasticamente questo metodo che, per quanto funzionante, non è adatto allo standard che vorrei per il codice.

AudioManagerImpl:
Questa classe è semplice e funzionale, però è molto limitata per la gestione dell'audio, per esempio, si possono utilizzare solo file audio **.wav** (che occupano troppo spazio) e non è possibile cambiare il volume della musica ma mantenere quello degli effetti sonori.
Ci sono soluzioni come usare più clip audio ma non credo sia ideale, sarebbe invece più interessante cercare altre librerie.
Una parte migliorabile della classe è il modo in cui i file audio sono gestiti e selezionati dai metodi: vengono caricati automaticamente nel **costruttore** e messi un una lista, e per selezionarli nei metodi va inserita la posizione nella lista, per cui ho creato un Enum **AudioIndex** che lega i nomi all'indice.

Per il mio ruolo nel gruppo, penso di aver fatto molto a livello di interazione tra classi, direzione generale del progetto, e refactoring di classi che consideravo mal implementate (Esempio: la classe **Projectile** **aveva** degli offSet specifici per i proiettili lanciati dal player, **ora** dipende da chi lo lancia e da quanto è grande il proiettile). Una pratica negativa che ho praticato inizialmente è l'implementazione di **Pattern** (es. **Observer**) con lo scopo di utilizzare il **Pattern** in sé e non come soluzione di un problema ignorandone l'intento.

Il progetto può essere un buon inizio per un progettista, sono presenti alcuni pattern e dipendenze (es. SpotBugs, CheckStyle) che migliorano il codice e instaurano buone abitudini.
Per poterlo sviluppare ulteriormente, secondo me sarebbe necessario un po' di refactoring (quando non lo è), prendendo come esempio principale EnemyManagerImpl.

## **Sezione 2 -** Filippo Patrignani

Nel progetto mi sono occupato della realizzazione di tutti i potenziamenti ottenuti dal player all'aumento di livello, della loro interazione con le statistiche del player nel caso degli accessori e della loro interazione coi nemici nel caso delle armi vere e proprie. Mi sono occupato inoltre della generazione e raccolta dell'esprienza. Penso di aver implementato il sistema degli Upgrades in modo corretto, perché ho utilizzato un sistema facilmente espandibile; infatti aggiungere armi, proiettili ed effetti all'interno del gioco risulta semplice e non richiede la modifica di classi già consolidate all'interno del progetto.

L'implementazione che ha dato più difficoltà è stata quella della MagicStaff, perché i proiettili a differenza delle altre armi avevano un effetto, infatti oltre all'observer aggiunto alla creazione dell'arma per

aggiungere tutti i proiettili generati alla lista del ProjectileManager, ho dovuto usare un observer su ogni proiettile per consentire di farlo esplodere a contatto col nemico.

Sono soddisfatto del mio contributo all'interno del gruppo, penso di aver sempre cercato per quanto possibile di stimolare la comunicazione; per quanto riguarda invece la programmazione in sé sono molto soddisfatto del mio lavoro, perché penso di aver programmato in modo abbastanza strutturato e logico, lasciando un codice che funzionasse, che fosse ordinato e comprensibile per gli altri membri.

Penso di aver imparato molto dall'esperienza di gruppo, sulle difficoltà di coordinazione e di comprensione reciproca in particolare, però anche su delle buone pratiche di programmazione grazie al feedback i altri membri alle mie modifiche al progetto, permettendomi ottenere un risultato finale sicuramente migliore rispetto a quello che avrei ottenuto senza altri input esterni.

## **Sezione 3 –** Sofia Ricupero

Il mio ruolo nel gruppo è stato principalmente legato alla gestione degli input e del controllo del player.
Mi sono occupata in particolare dell'**InputHandler**, implementando la gestione dei movimenti tramite tastiera: inizialmente i quattro movimenti principali (su, giù, sinistra, destra), e successivamente ho esteso il sistema per supportare anche il movimento diagonale, assicurandomi che la velocità rimanesse coerente in tutte le direzioni, evitando vantaggi o rallentamenti involontari.

Ho inoltre lavorato sulla classe **Player**, curandone la logica di movimento e di progressione, cercando di mantenerla modulare e

facilmente estendibile per supportare modifiche future o l'aggiunta di nuove funzionalità.

Una difficoltà inaspettata è emersa nella progettazione della **Camera**, soprattutto nella distinzione tra le coordinate del player e quelle della camera stessa. L'obiettivo era mantenere il player sempre centrato sullo schermo durante il movimento, ma trovare il giusto calcolo per farlo funzionare nel contesto del sistema di coordinate del gioco ha richiesto diversi tentativi e test.

Il progetto rappresenta una buona base, con margini di miglioramento e possibilità di espansione. È stata un'esperienza utile per consolidare le competenze tecniche e mettersi alla prova nel lavoro di gruppo.

## 4.2 Difficoltà incontrate e commenti per i docenti

Le difficoltà principali che abbiamo incontrato nel progetto sono legate per lo più al lavoro di gruppo, è capitato che, a causa di disorganizzazione e poca comunicazione iniziale nel gruppo, non si è fatto quasi nulla per mesi in contrasto ad alcuni giorni o settimane in cui è stata fatta una quantità elevata di lavoro.

Il corso in sè era ben strutturato, a patto di rimanere in pari, non dava l'impressione di essere particolarmente complesso e contribuiva a sviluppare buone abitudini nello scrivere codice.
Per questo non risultano evidenti modifiche necessarie.

# Appendice A

# Guida Utente

All’avvio dell’applicazione l’utente potrà scegliere tra 3 pulsanti: “New Game” che inizia una partita, “Settings” che apre il menù delle impostazioni e “Quit” per uscire dall’applicazione.
Durante una partita l’utente potrà muovere il proprio personaggio tramite la classica combinazione dei tasti “WASD”, e cercherà di evitare il contatto con i nemici mentre cerca di colpirli con varie armi, queste armi verranno sbloccate o potenziate randomicamente ad ogni level up che avviene raccogliendo abbastanza “orb di esperienza” lasciati dai nemici sconfitti.
Il giocatore inoltre può attivare la modalità di debug premendo il tasto “H” per vedere le hitbox di player, armi, nemici e proiettili e può mettere in pausa premendo il pulsante “P” che apre un menù di impostazioni simile a quello della schermata del titolo.
Entrambi i menù settings permettono di mettere l’applicazione in fullscreen, cambiare la dimensione dello schermo (è possibile anche cambiare la dimensione interagendo normalmente con la finestra per non limitare le opzioni dell’utente), aumentare e diminuire il volume.
Quando si cliccherà il pulsante “Quit” (nella schermata principale e dopo un Game Over) la finestra  e l’applicazione si chiuderà.