“L’Armata delle Tenebre”

Fabrizio Marzo

Luglio 2025

Indice

Capitolo 1

Analisi

1.1 Descrizione e Requisiti

L'applicazione emula il gioco 2D per smartphone *Survivor.io*, con alcune modifiche alla gestione del personaggio (il *Survivor*) e al comportamento dei nemici (gli *Zombie*). L'obiettivo del gioco è far sopravvivere il Survivor a tutte le ondate di Zombie previste per il livello.

- **Inizio partita:** Il Survivor inizia la partita in una posizione predefinita, già equipaggiato con un'arma (una *pistola*). Gli Zombie entrano nel livello da posizioni casuali, che possono trovarsi sia all'interno che all'esterno dei confini della mappa, e iniziano immediatamente a inseguire il Survivor.

- **Fase di combattimento:** Il Survivor deve evitare gli Zombie spostandosi liberamente all'interno dei limiti del livello (**non** oltre). Può sparare in tutte le direzioni con l'arma equipaggiata. I colpi sono infiniti, grazie a un meccanismo di *ricarica automatica*.

- **Fine partita:** La partita termina quando il Survivor riesce a sopravvivere a tutte le ondate zombie stabilite per il livello, eliminando tutti gli Zombie presenti, oppure alla morte del Survivor.

Requisiti Funzionali

- Il gioco deve gestire correttamente il movimento degli Zombie verso il target (Survivor), evitando la sovrapposizione tra di essi durante il movimento e rallentandone la velocità quando si trovano vicini ad altri Zombie.

- Il gioco deve gestire correttamente la meccanica di sparo del Survivor verso gli Zombie, includendo una gestione precisa delle collisioni tra munizioni e Zombie, oltre al movimento del Survivor all'interno dei limiti della mappa di gioco.
- Una gestione corretta delle ondate di Zombie in base a intervalli di tempo.

1.2 Modello del Dominio

Il gioco deve essere in grado di presentare al giocatore un Livello dove affronterà dei zombi, il giocatore deve evitare di entrare in contatto con i zombi per non perdere vita, il personaggio ha a disposizione un armentatario con vari munizioni, caricatori e armi, pronti per essere usati contro i zombi, il personaggio e i zombi hanno una specifica fisica che li contraddistinue, ogni livello ha una sua specifica gestione dei Zombie al suo iterno, si dal loro posizionamento iniziale e anche il tipo di Zombie che il personaggio dovrà sconfiggere.

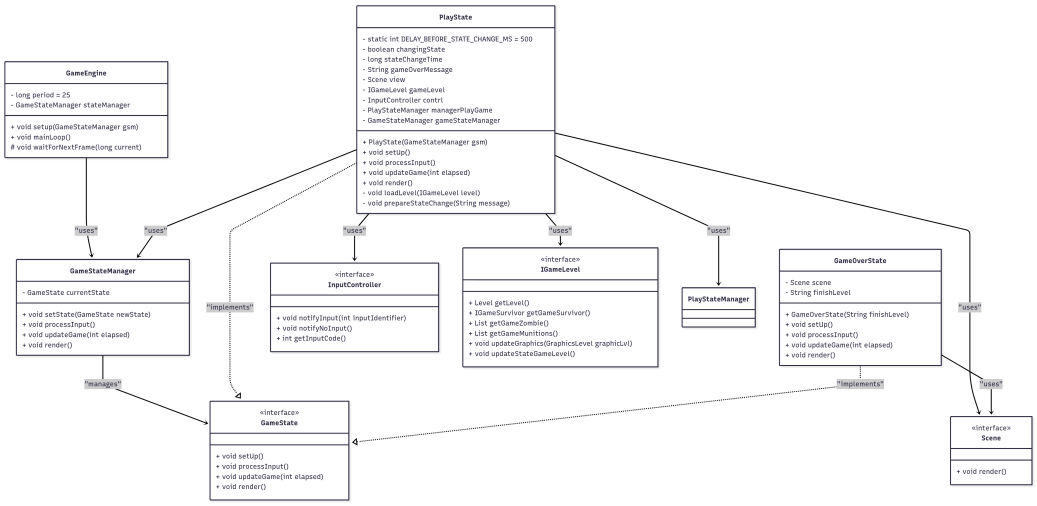


Figura 1.1: Diagramma UML dell'analisi del dominio rappresentante alcune interfaccie e i vari concetti di gioco

Capitolo 2

Design

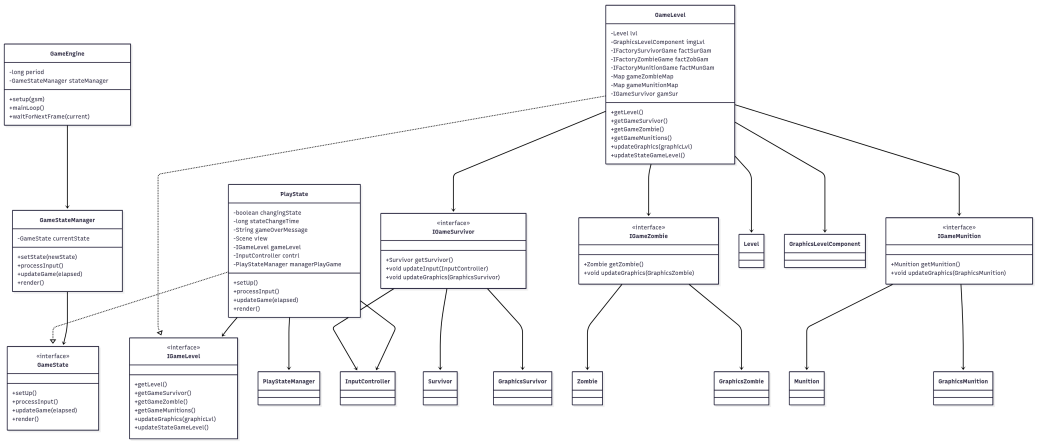
2.1 Architettura

L'architettura dell' *Armata delle Tenebre* segue una versione semplificata del pattern MVC (Model-View-Controller), e utilizza librerie di Java Swing per gestire l'interazione con i pulsanti della tastiera e, quando un pulsante viene premuto il controller invia l'input al modello che lo processa e aggiorna i valori interni, la pressione di un pulsante determina un'azione corripondente al giocatore. Inoltre o deciso di utilizzare per il modello del gioco una versione anch'essa semplificata del pattern Entity-Component-System (ECS), questo pattern mi è sembrato il più adatto a modellare il gioco perchè supporta la facile creazioni di nuovi attori (ad esempio nuovi survivor, zombie, armi, ecc), cosi facendo garantisco una buona suddivisione del codice e delle responsabilità. Di seguito viene spiegato le varie parti del architettura del progetto:

- **Model**: Si occupa del modello, cioè della gestione di tutti i valori interni e delle azioni su zombi, survivor e livelli.
- **Input-Controller**: Viene usato per mettere in comunicazione il modello con l'interfaccia grafica, in particolare il controller gestisce l'input da tastiera, interpretando i comandi dell'utente e aggiornando di conseguenza lo stato del modello.
- **View**: Contiene le interfaccie grafiche utilizzate per le varie scene del gioco e delle componenti grafiche dei vari elementi del model, mi sono servito Java Swing.
- **Game-Model**: E' un raccoglitore di Component, ogni Entità è descritta dai suoi componenti che permettono alla stessa di distinguerla

dalle altre entità in gioco, ad esempio, diverse entità presenti nel gioco potrebbero contenere una GraphicsComponent o InputComponent diversi.

- **Game-Engine e Game-State**: Contengono le classi che controllano effettivamente lo svolgersi del gioco. Engine si occupa di gestire il *game loop*, mentre State con il suo *manager* gestisce i cambiamenti di state del gioco.



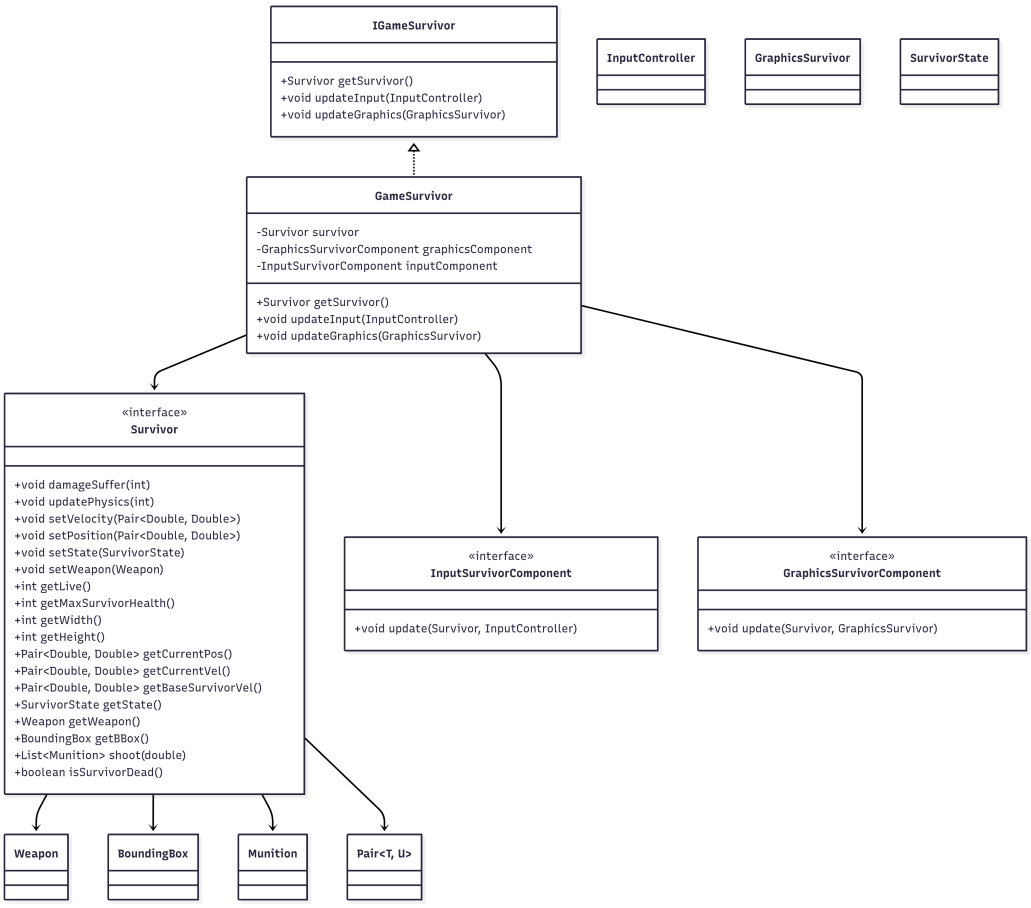
2.2 Design Dettagliato

MVC e Composite

Problema : Come rappresentare in modo modulare, estendibile e mantenibile gli elementi di gioco, assicurando una netta separazione tra logica di gioco, rappresentazione visiva e interazione utente, in conformità con il pattern architetturale MVC?

Soluzione : Per risolvere questo problema architetturale, ho scelto di adottare una composizione basata sul *pattern Composite*, integrandolo all'interno del modello *MVC*. Ogni elemento di gioco viene costruito come un'entità composta da diverse componenti, ognuna con una responsabilità ben definita. Ad esempio, tutti gli elementi condividono una **componente grafica**, indispensabile per la loro rappresentazione sullo schermo, mentre solo alcuni – come il personaggio controllato dal giocatore (es. il `Survivor`) – includono anche una **componente di input** per gestire l'interazione diretta dell'utente.

Questa architettura consente di trattare ogni oggetto del gioco come un insieme modulare di sotto-componenti indipendenti, rendendo semplice l'aggiunta, la modifica o la rimozione di funzionalità senza dover riscrivere l'intera logica dell'entità. Inoltre, la separazione tra *model*, *view* e *controller* è garantita dalla natura stessa del Composite: la logica (model) viene definita in un modulo dedicato, mentre le componenti grafiche e di input (view e controller) risiedono nel modulo di gioco vero e proprio (`game`). Questo approccio migliora la manutenibilità del codice, promuove il riutilizzo delle componenti e semplifica la gestione degli oggetti complessi nel contesto dinamico di un videogioco.



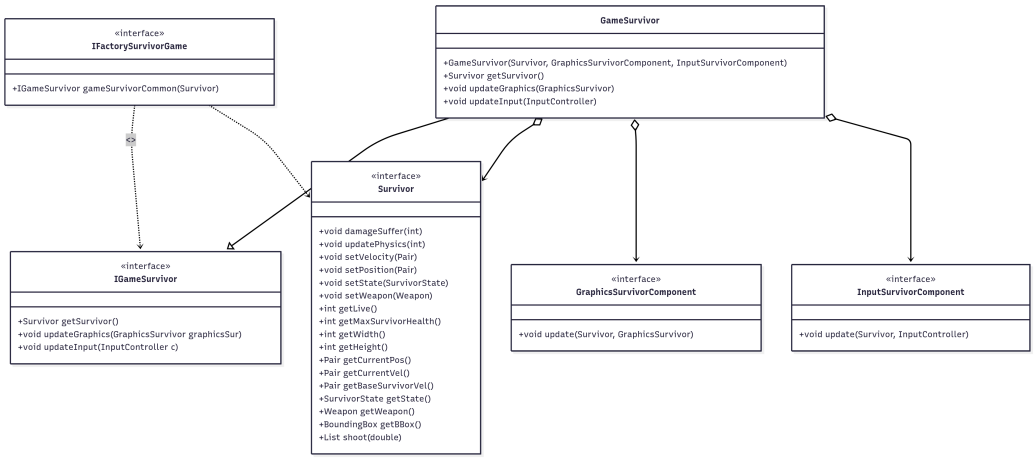
Generazione Entità

Problema : Come gestire la variabilità e la flessibilità delle componenti che costituiscono gli elementi di gioco, evitando di dover modificare continua-

mente il codice ogni volta che cambia l'implementazione di una componente (grafica, input, ecc.)?

Soluzione : Per affrontare questa esigenza progettuale, ho combinato l'uso del *pattern Composite* con l'introduzione del *pattern Factory*. Mentre il Composite consente di strutturare ogni entità di gioco come un insieme modulare di componenti (es. grafica, input, logica), la Factory permette di astrarre e centralizzare la creazione di queste componenti, rendendola flessibile e configurabile. In altre parole, la Factory si occupa di istanziare le componenti corrette in base al contesto o ai parametri specifici, senza che il codice client debba conoscere i dettagli delle implementazioni concrete.

Questa strategia si applica in due contesti principali del progetto. Da un lato, nel `package game`, la Factory viene utilizzata per creare oggetti di gioco come `GameSurvivor`, `GameZombie` e `GameLevel`, assegnando loro le componenti grafiche e di input appropriate a seconda della configurazione desiderata. Dall'altro lato, anche nel `model` la Factory è impiegata per l'istanziazione centralizzata e flessibile delle entità base, come `Survivor`, `Zombie` e `Level`. In questo modo si ottiene un sistema altamente modulare, facilmente estendibile e manutenibile, dove le modifiche alle componenti non impattano sul codice che le utilizza.



Level System

Problema : Come progettare un sistema di livelli che sia flessibile, estendibile e autonomo nella gestione degli eventi di gioco, mantenendo una netta separazione tra la logica interna del gioco e gli aspetti legati al gameplay o all'interfaccia utente?

Soluzione : Per affrontare questo problema, ho progettato la logica dei livelli come parte integrante del **model**, separandola completamente dalla logica di presentazione e di interazione. In questo modo, la gestione degli elementi di gioco e le loro regole rimangono indipendenti dalla visualizzazione o dall'input, facilitando la manutenzione e l'estensione del progetto.

Alla base di questa architettura c'è l'interfaccia `Level`, che definisce le funzionalità comuni a tutti i livelli, ed è implementata da classi concrete come `LevelTutorial`, utilizzata come esempio di livello base. Questa struttura permette di creare facilmente nuovi livelli con comportamenti personalizzati semplicemente implementando nuove classi che rispettano l'interfaccia.

Una componente chiave di ciascun livello è il `LevelManager`, un oggetto interno che si occupa della gestione dinamica degli eventi di gioco. Questo manager agisce come un coordinatore centrale, responsabile di orchestrare tutte le dinamiche specifiche del livello, evitando che la logica si disperda in più parti del codice. Tra le sue responsabilità principali ci sono:

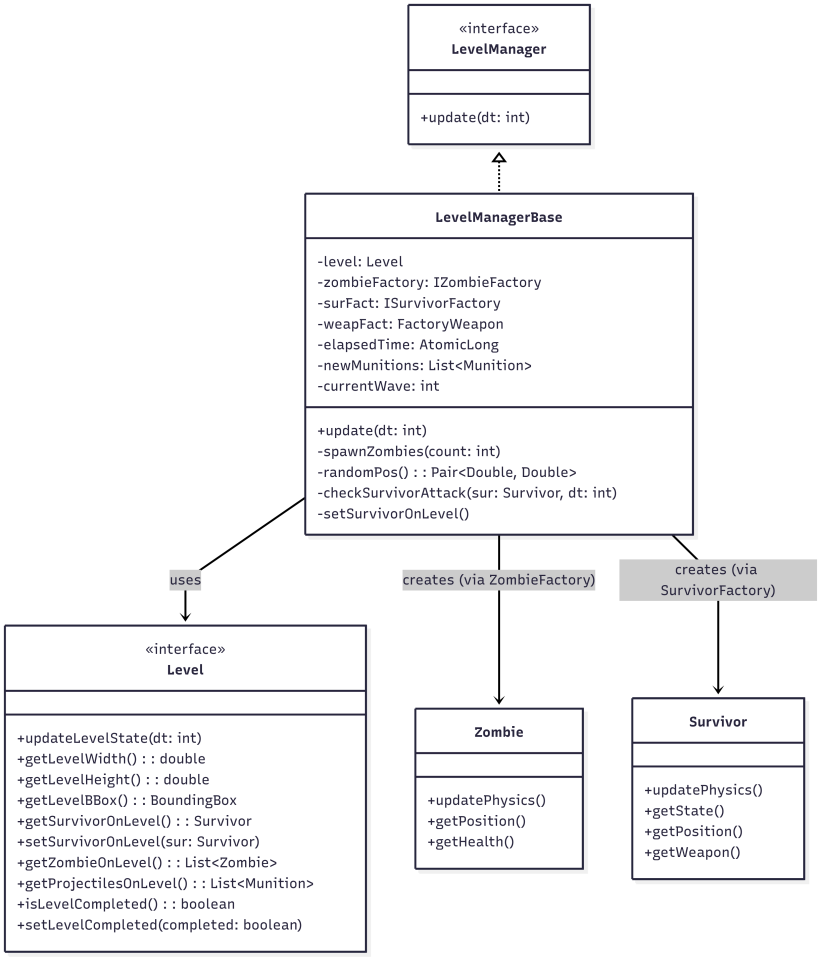
- **Spawn del Survivor**: posiziona il personaggio controllato dal giocatore in una posizione iniziale predefinita, assicurando coerenza e bilanciamento all'interno dello scenario;
- **Monitoraggio dello stato del Survivor**: controlla condizioni come la fase di attacco, permettendo di adattare il comportamento del gioco in tempo reale;
- **Gestione delle ondate di Zombie**: coordina la comparsa di zombie secondo tempistiche e quantità configurabili, aumentando la sfida durante il livello;
- **Creazione di zombie personalizzati**: genera nemici con caratteristiche uniche in base al livello, favorendo varietà e profondità del gameplay.

Dal punto di vista strutturale, ogni oggetto `Level` contiene:

- Il `Survivor` attualmente presente;
- Una lista di `Zombie` attivi nel livello;
- Una lista di `Munition`, ovvero i proiettili generati durante gli scontri;
- Le dimensioni del livello (larghezza e altezza in centimetri);
- Un flag booleano che indica se il livello è stato completato.

Infine, il comportamento dinamico del livello è governato dal metodo ciclico `updateLevelState`, che viene chiamato regolarmente durante il gioco. Questo metodo aggiorna lo stato delle entità, gestisce la comparsa di nuovi nemici, rileva eventi e sincronizza le azioni all'interno del livello, garantendo una simulazione fluida e coerente.

Grazie a questa architettura, ogni livello è **modulare**, **autonomo** e **facilmente estendibile**, permettendo di implementare logiche specifiche e personalizzate senza impattare sugli altri componenti del gioco. Ciò consente di mantenere il codice pulito, organizzato e pronto per futuri ampliamenti o modifiche.



Rappresentazione grafica delle Entità

Problema : Come strutturare la parte grafica dell'applicazione in modo da separarla completamente dalla logica del gioco?

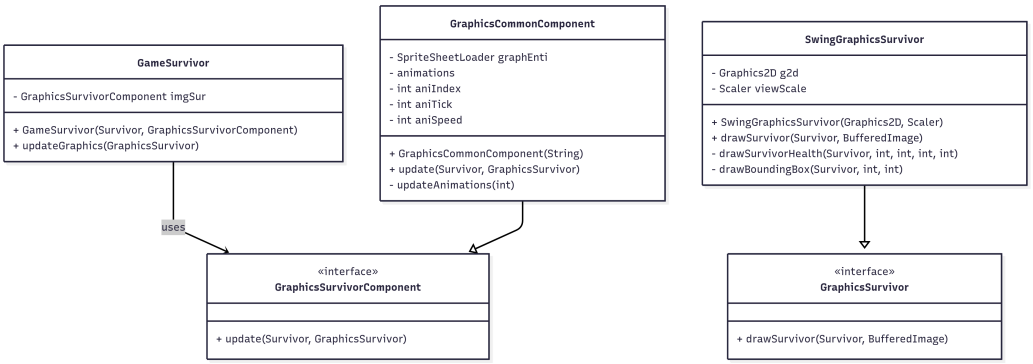
Soluzione : Per capirla al meglio prendiamo in esempio come ho gestito la grafica del Survivor. Nel progetto, la `GameSurvivor` svolge un ruolo centrale nella gestione della View. Per usare un'analogia artistica, può essere intesa come la *tela* su cui si disegnano tutte le componenti visive relative al Survivor. Essa incapsula la logica per la visualizzazione del personaggio nel contesto di gioco, mantenendo così isolata la parte grafica dai dettagli interni del modello.

All'interno di `GameSurvivor` troviamo la `GraphicsSurvivorComponent`, la componente grafica dedicata alla rappresentazione visiva del Survivor. Questa classe si occupa di:

- Gestire la **sprite sheet**, ovvero l'insieme delle immagini che compongono le diverse pose e animazioni del personaggio.
- Controllare l'**aggiornamento delle animazioni**, effettuando l'update dei frame per ottenere una rappresentazione fluida e dinamica del movimento e delle azioni del Survivor.
- Coordinare la resa grafica, che in termini artistici rappresenta la definizione dei *colori*, delle forme e delle transizioni visive necessarie per mostrare lo stato attuale del personaggio.

Il metodo fondamentale di questa componente è `update`, che riceve come parametri un oggetto `Survivor` (ovvero il soggetto da disegnare, o la "modella" in termini artistici) e un'istanza di `GraphicsSurvivor`. Quest'ultima classe definisce lo **stile di disegno** e può essere implementata utilizzando diverse librerie grafiche Java, come `Java Swing` o `JavaFX`. In termini artistici, `GraphicsSurvivor` rappresenta la *tecnica pittorica* o il *medium* con cui la tela viene decorata, permettendo quindi di scegliere l'approccio grafico più adatto alle esigenze del progetto o alle preferenze di implementazione.

Questa suddivisione tra componente grafica e stile di disegno permette di mantenere una forte modularità e flessibilità nella gestione della View, facilitando l'estensione o la modifica delle tecniche di rendering senza impattare sulla logica del modello o del gioco.

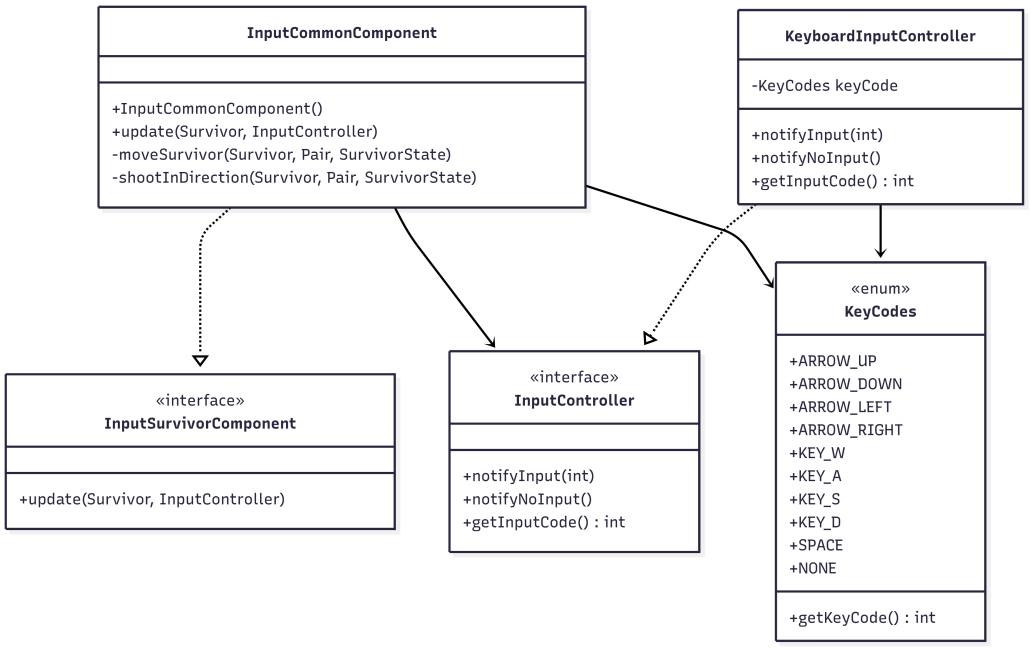


Logica d'Input

Problema : Come posso progettare un sistema di gestione degli input che sia flessibile, scalabile e coerente permettendo l'interazione dell'utente con i personaggi senza compromettere la separazione tra logica e interfaccia grafica?

Soluzione : Nel progetto, la gestione degli input rappresenta una componente fondamentale per l'interazione tra l'utente e i personaggi del gioco. Ogni oggetto di tipo `GameSurvivor` è associato a una componente di input, incaricata di aggiornare lo stato del personaggio sulla base delle azioni dell'utente. Il metodo `update` di tale componente riceve un oggetto `InputController`, che ha il compito di rilevare e notificare gli input ricevuti durante ciascun ciclo di aggiornamento. Attualmente, il sistema include una specifica implementazione denominata `KeyboardInputController`, dedicata alla gestione degli input da tastiera. Tuttavia, l'intera architettura è progettata per essere facilmente estendibile: ciò consente, ad esempio, l'integrazione di nuovi dispositivi di input, come un gamepad, senza modificare la logica di comunicazione con il `GameSurvivor`.

L'istanza di `KeyboardInputController` è inoltre utilizzata dalla classe `SwingSceneTutorial`, la quale si occupa della rilevazione degli eventi da tastiera tramite l'interfaccia Swing e della loro trasmissione al controller. In questo schema, l'`InputController` agisce da intermediario tra la vista (la scena grafica) e il modello (la logica di controllo del personaggio), mantenendo una netta separazione dei ruoli secondo il paradigma Model-View-Controller (MVC). Questa organizzazione architetturale garantisce modularità e riusabilità, facilitando l'integrazione di nuovi dispositivi di input e la manutenzione del codice.



AI NPC

Problema : Come posso progettare un sistema per il comportamento degli NPC capace di garantire comportamenti credibili?

Soluzione : Nel mio progetto, la creazione del comportamento di base per gli NPC avviene tramite un metodo della factory chiamato createBaseNPCBehavior(), il quale restituisce un'istanza della classe `AIZombieBehavior`. Quest'ultima rappresenta una strategia composita, costruita unendo tre comportamenti distinti ma strettamente correlati, ognuno dei quali incapsula una logica fondamentale per garantire un'interazione credibile degli zombie all'interno del livello.

In particolare, `AIZombieBehavior` è configurata con i seguenti tre componenti principali:

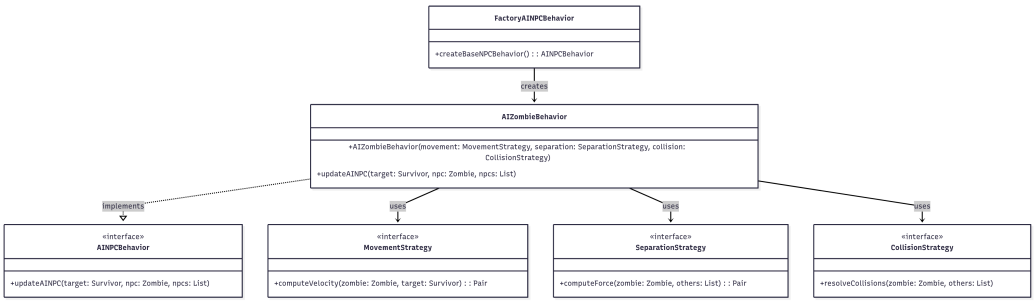
- **Componente di movimento verso il target**: si occupa della logica di inseguimento del `Survivor`. Gli zombie calcolano la direzione ottimale in base alla posizione del giocatore e aggiornano la propria traiettoria di conseguenza, garantendo un comportamento dinamico e reattivo.
- **Componente di separazione (separation)**: implementa un meccanismo di "distanziamento" tra gli zombie. Quando due o più entità si

trovano a una distanza minima predefinita, questa strategia regola la loro velocità o modifica leggermente la direzione di movimento, impedendo la sovrapposizione e mantenendo un comportamento collettivo realistico. Questo approccio si ispira a tecniche utilizzate nei sistemi di boid e flocking.

- **Componente di gestione delle collisioni tra gruppi di zombie**: interviene quando più zombie entrano in contatto diretto. In questo caso, viene applicata una logica di risoluzione delle collisioni che impedisce comportamenti incoerenti (come il completo blocco o la sovrapposizione visiva), mantenendo una corretta fisicità e interazione tra i personaggi non giocanti.

Tutte queste strategie sono iniettate all'interno dell'istanza di `AIZombieBehavior` tramite la factory. In tal modo, il comportamento dell'NPC non è rigido o codificato staticamente, ma altamente modulare e riutilizzabile. Questo design favorisce la flessibilità, poiché è possibile creare rapidamente nuove combinazioni comportamentali semplicemente sostituendo una o più delle strategie componenti, senza alterare la struttura generale del codice.

Questo approccio, che unisce il pattern Factory e il principio di composizione, si è rivelato estremamente efficace nella gestione dell'intelligenza artificiale nemica. Non solo ha permesso di ottenere comportamenti diversificati e realistici da parte degli zombie, ma ha anche semplificato notevolmente l'estensione futura del progetto, rendendo possibile l'introduzione di nuove tipologie di NPC dotati di logiche comportamentali alternative, come movimenti erratici, attacchi coordinati o reazioni ambientali.



Game System

Problema : Come posso strutturare l'esecuzione del gioco in modo fluido e modulare, garantendo un controllo efficiente delle diverse fasi (input, logica e rendering) e permettendo una gestione flessibile dei vari stati di gioco, come il gameplay attivo o la schermata di fine partita?

Soluzione : Nel progetto, la logica principale di esecuzione è affidata alla classe `GameEngine`, che implementa il *game loop*, un ciclo continuo responsabile dell'esecuzione sincronizzata delle principali fasi del gioco. In particolare, il metodo `mainLoop()` esegue ciclicamente quattro operazioni fondamentali: il processamento dell'input, l'aggiornamento della logica, il rendering e la sincronizzazione del frame rate.

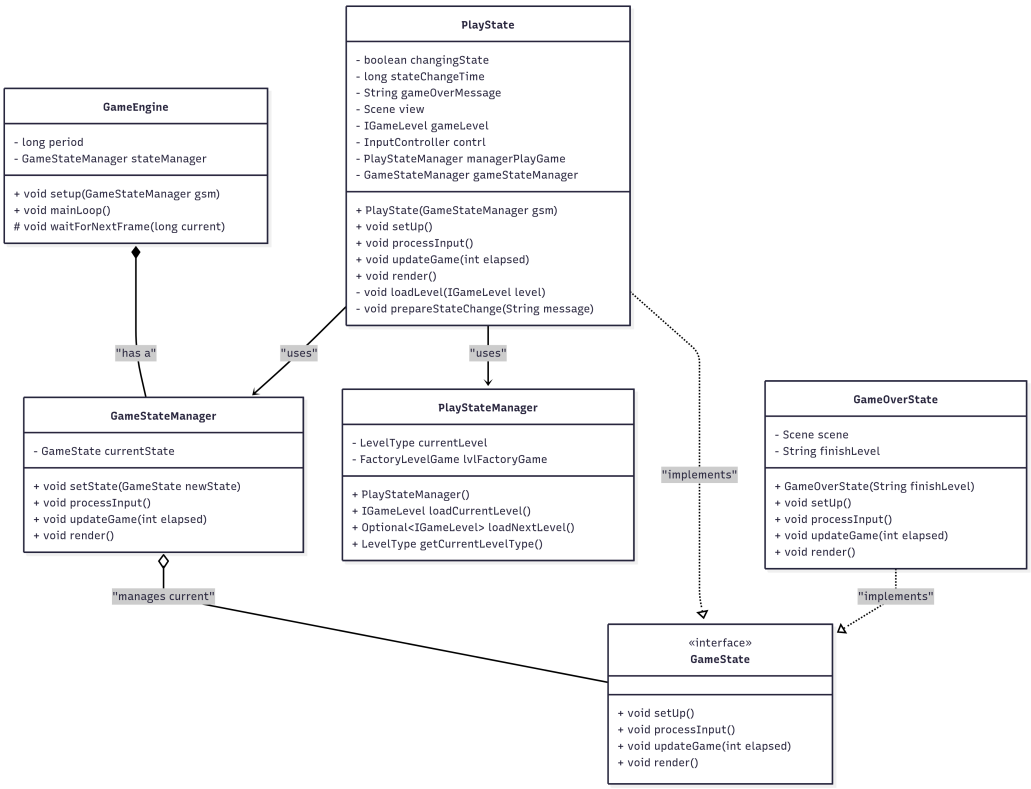
- **Processamento dell'input**: delegato all'attuale stato di gioco tramite il `GameStateManager`.
- **Aggiornamento della logica di gioco**: gestito attraverso il metodo `updateGame()`, che utilizza il tempo trascorso per mantenere una simulazione temporale coerente.
- **Rendering**: visualizzazione grafica dello stato corrente del gioco.
- **Sincronizzazione del frame rate**: mediante il metodo `waitForNextFrame()`, che regola la velocità del ciclo per mantenere un intervallo fisso (circa 25 ms, corrispondenti a 40 FPS).

Questa struttura assicura un flusso regolare e prevedibile, separando in modo chiaro le responsabilità tra input, logica e output grafico.

Per gestire le diverse fasi operative del gioco, è stato adottato il *pattern State*. Questo pattern consente di incapsulare il comportamento specifico di ciascuno stato in classi che implementano l'interfaccia `GameState`, la quale definisce metodi comuni come `setUp()`, `processInput()`, `updateGame()` e `render()`. La classe `GameStateManager` funge da gestore centrale, mantenendo il riferimento allo stato corrente e delegando a esso le chiamate appropriate.

La transizione tra stati avviene tramite il metodo `setState()`, che garantisce l'inizializzazione del nuovo stato al momento del cambio. Tra le implementazioni principali vi è la classe `PlayState`, che gestisce la modalità attiva di gioco: carica i livelli tramite il `PlayStateManager`, elabora gli input del giocatore, aggiorna la logica (incluso il comportamento dei nemici) e gestisce le condizioni di fine partita. Al termine del gioco, la transizione avviene verso lo stato `GameOverState`, che visualizza una schermata conclusiva senza ulteriori aggiornamenti o interazioni.

Questo approccio, basato su una struttura modulare e il principio di separazione delle responsabilità, facilita l'estensione del progetto con nuovi stati (come menu, pause, livelli speciali), migliorando la manutenibilità e la leggibilità del codice, e garantendo un'esperienza utente fluida e coerente.


PlayState
- boolean changingState
- long stateChangeTime
- String gameOverMessage
- Scene view
- IGameLevel gameLevel
- InputController contrl
- PlayStateManager managerPlayGame
- GameStateManager gameStateManager
+ PlayState(GameStateManager gsm)
+ void setUp()
+ void processInput()
+ void updateGame(int elapsed)
+ void render()
- void loadLevel(IGameLevel level)
- void prepareStateChange(String message)
GameEngine
- long period
- GameStateManager stateManager
+ void setup(GameStateManager gsm)
+ void mainLoop()
# void waitForNextFrame(long current)
"has a"
"uses"
"uses"
"implements"
GameStateManager
- GameState currentState
+ void setState(GameState newState)
+ void processInput()
+ void updateGame(int elapsed)
+ void render()
PlayStateManager
- LevelType currentLevel
- FactoryLevelGame lvlFactoryGame
+ PlayStateManager()
+ IGameLevel loadCurrentLevel()
+ Optional<IGameLevel> loadNextLevel()
+ LevelType getCurrentLevelType()
GameOverState
- Scene scene
- String finishLevel
+ GameOverState(String finishLevel)
+ void setUp()
+ void processInput()
+ void updateGame(int elapsed)
+ void render()
"manages current"
"implements"
«interface»
GameState
+ void setUp()
+ void processInput()
+ void updateGame(int elapsed)
+ void render()

Capitolo 3

Sviluppo

3.1 Testing

Per garantire l'affidabilità e la correttezza del sistema, è stata adottata una strategia di testing basata su test unitari automatici, realizzati principalmente con JUnit 5. L'approccio utilizzato mira a verificare il comportamento corretto di ogni singola componente, assicurandone il corretto funzionamento in condizioni ordinarie e al contempo gestendo correttamente casi limite e situazioni di errore.

Test delle componenti di armamento

Le classi relative all'armamento, quali `Charger`, `Munition` e `Pistol`, sono state testate approfonditamente per validarne l'inizializzazione, il corretto comportamento dinamico e le interazioni tra oggetti.

In particolare, per la classe `Charger` si è verificato che la fabbrica crei istanze non nulle e del tipo corretto, che la capacità iniziale sia rispettata e che l'estrazione delle munizioni comporti la riduzione del carico disponibile.

La classe `Munition`, nello specifico la munizione `Parabellum`, è stata testata per la corretta inizializzazione delle proprietà (danno, dimensioni, posizione), la corretta generazione della bounding box e la gestione dello stato di tiro, con particolare attenzione al movimento post-sparo e alla gestione delle condizioni non valide.

Per quanto riguarda la classe `Pistol`, i test hanno verificato il corretto conteggio delle munizioni, il rispetto del cooldown tra i colpi e il corretto comportamento anche in presenza di più proiettili per singolo sparo.

Test delle entità e della fisica

Sono stati realizzati test anche per le entità del gioco, quali `Survivor` e `Zombie`. Questi test verificano la corretta gestione dei danni subiti, l'aggiornamento di posizione e velocità, la modifica dello stato interno, nonché l'integrazione con armi (mockate per isolare il test dell'entità).

I test sul `Zombie` includono la verifica della posizione iniziale, la capacità di attacco, la riduzione dei punti vita in seguito a danni e la corretta transizione di stato in caso di morte.

Test del livello di gioco

La classe `Level` è stata sottoposta a test per confermare la corretta inizializzazione delle dimensioni e della bounding box, la presenza di almeno un sopravvissuto e di una lista non vuota di zombie all'avvio, la corretta gestione delle liste di proiettili e la valutazione dello stato di completamento del livello in relazione alla progressione delle ondate nemiche.

Test delle componenti grafiche e di collisione

Infine, la classe `BoundingBox` è stata testata per validare il corretto calcolo degli angoli, la rilevazione delle collisioni (sia in presenza di sovrapposizioni che nel caso di semplici contatti sui bordi) e l'aggiornamento dinamico delle dimensioni in relazione allo spostamento dell'entità a cui è associata.

Approccio metodologico

L'utilizzo di framework di mocking come Mockito ha permesso di isolare i test delle entità dal comportamento delle armi, consentendo di valutare il funzionamento delle componenti singolarmente senza dipendere da implementazioni esterne. I test sono stati organizzati in modo da coprire scenari tipici, casi limite, e comportamenti attesi in presenza di input errati o stati non validi.

3.2 Note di sviluppo

Utilizzo di Streams e Lambda Expressions

Utilizzo di Streams e Lambda Expressions per migliorare l'efficienza e la chiarezza del codice. Alcuni esempi:

- Permalink: https://github.com/Fabrizio-Marzo/L-armata-delle-Tenebre/blob/6ea3c50a29f4dbb5efa86e0a9aca1f45569ffa03/src/main/java/model/level/manager/LevelManagerBase.java#L92C7-L95C1
- Permalink: https://github.com/Fabrizio-Marzo/L-armata-delle-Tenebre/blob/6ea3c50a29f4dbb5efa86e0a9aca1f45569ffa03/src/main/java/model/physics/physics_level/PhysicsLevelTutComponent.java#L48C9-L52C20
- Permalink :

Utilizzo di Optional

Utilizzo di Optional per evitare la notazione Null.

- Permalink: https://github.com/Fabrizio-Marzo/L-armata-delle-Tenebre/blob/6ea3c50a29f4dbb5efa86e0a9aca1f45569ffa03/src/main/java/game/game_state/PlayStateManager.java#L50C1-L53C78

Uso Libreria Apache Commons Lang

L'Uso di **Pair** di questa Libreria è usata per sostituire il concetto di *Vettore*, visto che Pair e Vettore sono molto simili nella loro implementazione, la classe Pair viene usata int tutti il progetto.

- Permalink: https://github.com/Fabrizio-Marzo/L-armata-delle-Tenebre/blob/6ea3c50a29f4dbb5efa86e0a9aca1f45569ffa03/src/main/java/model/bounding_box/RectBoundingBox.java#L27C5-L33C1

Capitolo 4

Commenti finali

4.1 Autovalutazione

Sono molto soddisfatto del lavoro svolto durante lo sviluppo del progetto, non soltanto per la quantità e qualità del codice prodotto, ma soprattutto per l'approccio adottato nella fase di progettazione. Un aspetto che ritengo particolarmente significativo è stata la ricerca e lo studio necessari per individuare e applicare correttamente i design pattern più adatti alle esigenze del progetto. In particolare, l'adozione del pattern *Composite* ha rappresentato una soluzione efficace per la gestione gerarchica degli oggetti di gioco, mentre l'implementazione del pattern MVC è stata curata con attenzione per garantire una separazione chiara tra modello, vista e controllo.

Sono altresì molto soddisfatto dell'architettura sviluppata per la gestione degli input, pensata non solo per supportare gli input da tastiera, ma anche per poter essere estesa in futuro a nuovi dispositivi, come controller o touch input, senza alterare le componenti esistenti. Questo approccio riflette una visione generale di estendibilità e modularità che ha guidato l'intero progetto. L'attenzione rivolta alla futura manutenzione e all'adattabilità del codice mi ha spinto a ragionare in modo più strutturato e orientato alla progettazione software, portandomi a migliorare sensibilmente le mie competenze.

Nel complesso, considero il progetto ben riuscito sia dal punto di vista tecnico che metodologico, e rappresenta un passo significativo nel mio percorso formativo.

4.2 Difficoltà Riscontrate

Durante lo sviluppo del progetto, una delle principali difficoltà è stata la gestione autonoma di alcune componenti che inizialmente non rientravano nella

mia assegnazione. A causa dell'assenza o dell'impossibilità di collaborazione con alcuni membri del gruppo, mi sono trovato a dover realizzare parti aggiuntive del sistema, con conseguente ridefinizione delle tempistiche e delle priorità del lavoro.

Questa situazione ha comportato un inevitabile aumento del carico di lavoro individuale e ha richiesto una riorganizzazione della pianificazione iniziale. Di conseguenza, in alcune sezioni è stato necessario procedere con soluzioni meno ottimizzate rispetto a quanto inizialmente previsto, e non è stato possibile completare pienamente tutte le funzionalità desiderate.

Nonostante ciò, l'idea progettuale di base è rimasta solida e coerente, e la struttura del progetto ha mantenuto una buona modularità. Tuttavia, l'impegno profuso per supplire ad attività non previste ha inevitabilmente influenzato la qualità e la completezza di alcune parti del codice.

Capitolo 5

Guida Utente

Una volta avviato il gioco e caricato il livello, il personaggio principale (*Survivor*) viene automaticamente posizionato all'interno dell'ambiente di gioco. Da questo momento, l'utente può controllarne i movimenti e le azioni tramite la tastiera, secondo la seguente configurazione dei comandi:

- **Movimento del personaggio**
  - `W` – Sposta il personaggio verso l'alto
  - `A` – Sposta il personaggio verso sinistra
  - `S` – Sposta il personaggio verso il basso
  - `D` – Sposta il personaggio verso destra
- **Azione di attacco (sparo)**
  - `Freccia su` (↑) – Spara verso l'alto
  - `Freccia sinistra` (←) – Spara verso sinistra
  - `Freccia giù` (↓) – Spara verso il basso
  - `Freccia destra` (→) – Spara verso destra

Questa configurazione consente un controllo intuitivo e diretto del personaggio, permettendo al giocatore di muoversi liberamente nello spazio di gioco e di difendersi in tutte le direzioni. I comandi sono stati scelti per offrire reattività e semplicità, facilitando l'interazione anche durante le fasi più concitate del gameplay.

Al termine della partita, sia in caso di vittoria (completamento del livello) che in caso di sconfitta (morte del personaggio), verrà visualizzata una schermata conclusiva. Per terminare definitivamente l'esecuzione del gioco, l'utente deve premere il pulsante `Exit Game`, che chiude correttamente l'applicazione e consente di uscire in modo controllato.