“L’Armata delle Tenebre”

Fabrizio Marzo

Luglio 2025

Indice

Capitolo 1

Analisi

1.1 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.

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.
- La gestione del livello è affidata a un `LevelManager`, responsabile della creazione delle ondate di Zombie in base a intervalli di tempo prestabiliti.
- Deve essere rispettata una chiara separazione tra la parte di *View* e quella di *Model*, in linea con l'architettura MVC.

1.2 Analisi e Modello del Dominio

Il progetto si pone l'obiettivo di separare il dominio logico del gioco dalla parte grafica, seguendo il principio dell'architettura MVC.

La parte di **Model** (dominio) contiene le principali *entità* del gioco, come i diversi tipi di *Survivor*, *Zombie* e *Livelli*. Ogni entità è caratterizzata da dimensioni specifiche espresse in **centimetri** (**unità di misura del modello**). Essendo un gioco in 2D, si è scelto di non creare una classe dedicata `Vettore2D`, ma di utilizzare la classe `Pair` di Apache. Entrambe le soluzioni sono molto simili sia nell'implementazione sia nell'utilizzo all'interno del progetto, e la classe `Pair` incapsula esattamente l'informazione desiderata, ovvero una coppia di valori. Il progetto distingue chiaramente tra le entità del modello e le entità di gioco. Le entità di gioco sono composte dalla parte grafica e dall'entità di riferimento; per i Survivor è inclusa anche la componente di input.

Questa "traduzione" avviene durante l'avvio del gioco: il livello del modello viene convertito in un livello di gioco e, a cascata, tutti gli Zombie e i Survivor vengono trasformati da entità di modello in entità di gioco.

Schema UML, rappresenta l'Entità di gioco Survivor.

Capitolo 2

Design

2.1 Architettura

2.2 Implementazione del Pattern MVC e uso del Pattern Composite

L'idea fondamentale alla base del mio progetto, in linea con il paradigma MVC (Model-View-Controller), è la seguente: nel `package model` definisco tutte le entità che rappresentano ciò che il mio mondo, in altre parole, qui risiedono le classi che descrivono gli oggetti di gioco a livello concettuale e logico, come il `Survivor`, gli `Zombie`, i `Livelli` e l'`Armamentario`. Nel `package game` invece si trova il vero e proprio "mondo di gioco", ovvero la rappresentazione attiva e interattiva di questi elementi durante l'esecuzione del gioco. Qui gli elementi del `model` vengono trasformati in oggetti giocabili e dinamici. Per realizzare questa trasformazione, ed implementare efficacemente il pattern MVC, ho fatto uso del **pattern Composite**. Questo perché gli elementi di gioco non sono semplici oggetti isolati, ma sono *composti* da più componenti:

- una **componente grafica** comune a tutti gli elementi, necessaria per la loro rappresentazione visiva sullo schermo;
- in alcuni casi, come per il `Survivor`, anche una **componente di input** che permette di gestire l'interazione diretta con il giocatore.

Questa composizione consente di costruire un'entità di gioco che integra in modo modulare sia la logica interna (definita nel `model`) sia la sua rappresentazione e la sua interattività (gestite nel `game`).

In sintesi, per implementare il pattern MVC nel mio progetto, ho dovuto necessariamente adottare il pattern Composite: grazie a questo, ogni elemento di gioco è strutturato come un oggetto composto da più sotto-elementi (componenti grafiche, input, ecc.), permettendo una chiara separazione tra modello, vista e controllo e facilitando la gestione di oggetti complessi nel contesto di gioco.

2.3 Ulteriore Utilizzo del Pattern Composite e Introduzione del Pattern Factory

Un ulteriore aspetto importante dell'uso del pattern Composite nel mio progetto riguarda la flessibilità e la variabilità delle componenti che costituiscono gli elementi di gioco. In particolare, tutte le componenti che vengono utilizzate all'interno delle classi nel `package game` possono variare, sia in fase di creazione degli oggetti di gioco sia durante il loro ciclo di vita.

Ad esempio, per la componente grafica e per la gestione dell'input non è detto che si debbano usare sempre le stesse implementazioni. A seconda delle esigenze o delle configurazioni, è possibile adottare diverse varianti per rappresentare e controllare un'entità come il `GameSurvivor`.

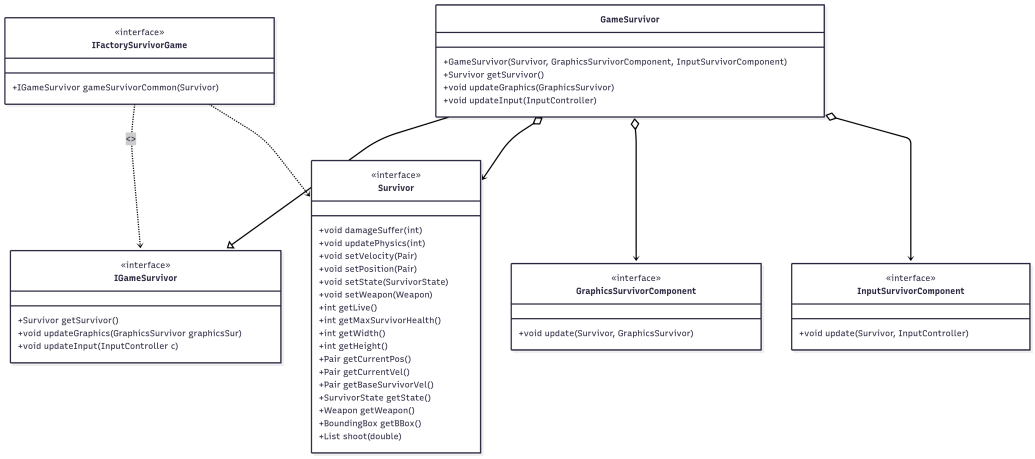
Per gestire questa variabilità ho fatto uso del **pattern Factory**, che permette di creare oggetti con diverse implementazioni senza modificare il codice client che li utilizza. In pratica, la Factory incapsula la logica di creazione degli oggetti, restituendo l'istanza più adatta in base al contesto o ai parametri forniti.

Nel mio progetto, la Factory viene utilizzata in due ambiti principali:

- nella creazione degli oggetti di gioco all'interno del `package game`, ad esempio per istanziare un `GameSurvivor`, `GameZombie`, `GameLevel`, con una determinata componente grafica o con una diversa gestione dell'input, a seconda delle specifiche esigenze;

- anche nella parte di `model`, per la creazione delle entità di base come `Survivor`, `Zombie`, `Level`, garantendo un metodo centralizzato e flessibile per l'istanziazione degli oggetti modello.

Questa combinazione di pattern Composite e Factory consente di ottenere un'architettura estremamente modulare, estendibile e facilmente mantenibile, capace di adattarsi a nuove necessità senza richiedere modifiche invasive al codice esistente.



2.4 Gestione del Survivor e uso avanzato della Factory

In questa sezione approfondisco la gestione del `Survivor` all'interno del `package model`, con particolare attenzione all'uso della **Factory** per la sua creazione.

Il concetto fondamentale alla base di questa implementazione è che, attraverso la `SurvivorFactory`, posso costruire diversi tipi di survivor (e in modo analogo anche gli zombie, livelli o nell'armamentario vari tipi di munizioni, caricatori o Armi) assegnando loro caratteristiche ben definite fin dalla fase di istanziazione. Questo approccio rispetta il principio di separazione delle responsabilità e migliora l'estensibilità del sistema.

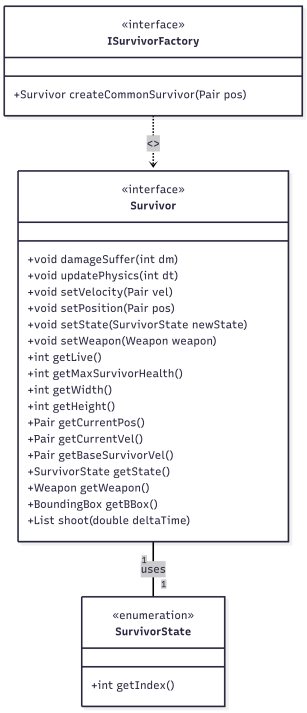
La factory, infatti, non si limita a creare oggetti, ma incapsula tutta la logica necessaria per configurare completamente un'entità nel mondo logico. Tra le caratteristiche configurabili tramite la factory troviamo:

- **La componente fisica**: ogni survivor può essere associato a un differente comportamento fisico, implementato tramite una specifica classe del `package model.physics`. In questo modo è possibile modificare, ad esempio, la logica di movimento dell'entità senza intervenire direttamente sulla logica della classe `Survivor`.

- **Il bounding box**: la factory consente anche di specificare il tipo di `BoundingBox` da assegnare all'entità, implementato attraverso una classe del `package model.bounding_box`. Questo permette di gestire in maniera flessibile il rilevamento delle collisioni, rendendo possibile modellare entità con forme e dimensioni differenti a seconda delle esigenze del gioco.

Questo approccio orientato ai pattern consente di centralizzare e uniformare la creazione delle entità, rendendo il codice più leggibile, manutenibile e pronto ad accogliere nuove funzionalità con un impatto minimo sul resto del sistema.

Nella pratica, l'interfaccia `ISurvivorFactory` definisce il metodo per creare i survivor, mentre la classe `SurvivorFactory` ne fornisce una concreta implementazione, specificando valori predefiniti come salute, velocità, dimensioni, fisica e bounding box. Eventuali nuovi tipi di survivor potranno essere creati facilmente estendendo la factory o aggiungendo nuove strategie di configurazione.



2.5 Gestione del Livello nel Modello

La logica del livello di gioco è stata progettata per essere parte integrante del `model`, mantenendo una netta separazione tra le responsabilità di rappresentazione logica del mondo e quelle relative al gameplay e all'interfaccia utente. A tal fine, è stata definita un'interfaccia `Level`, implementata dalla classe concreta `LevelTutorial`, pensata come esempio base di un possibile livello.

Per supportare la creazione di diversi tipi di livelli, è stata inoltre introdotta una **factory dei livelli**, incaricata di istanziare i vari oggetti di tipo

`Level` in base alle caratteristiche desiderate (es. dimensioni, bounding box e fisica).

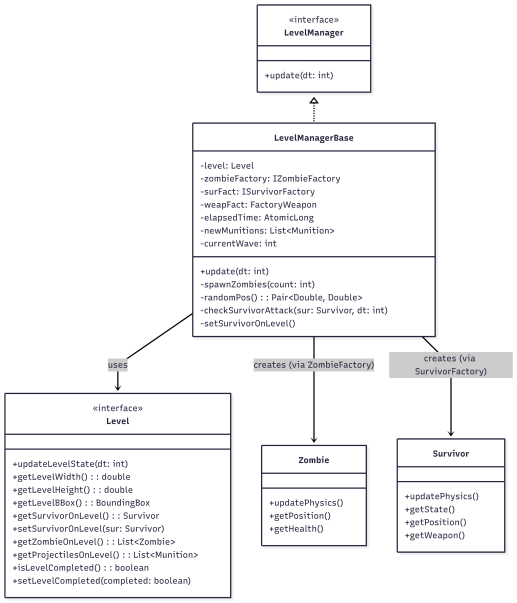
Una particolarità rilevante della progettazione è l'inclusione, all'interno di ciascun `Level`, di un campo `LevelManager`. Questo componente ha un ruolo fondamentale nella gestione dinamica degli eventi di gioco interni al livello, tra cui:

- **Gestione dello spawn del Survivor**: quando il livello viene avviato, il `LevelManager` si occupa di posizionare il survivor in una posizione predefinita all'interno dello scenario.
- **Controllo sullo stato del Survivor**: ad esempio per verificare se è in fase di attacco o meno.
- **Gestione dell'apparizione degli Zombie**: vengono gestite ondate (*waves*) di zombie, che possono comparire a intervalli regolari. Il manager decide quante entità generare e in quali posizioni del livello (interne o esterne).
- **Creazione di zombie specifici per il livello**: ogni livello dispone del proprio `LevelManager`, che incapsula una logica personalizzata per la creazione di zombie. Questo permette di generare tipologie di nemici differenti in base al contesto, favorendo così la varietà di gameplay tra i diversi livelli.

Dal punto di vista strutturale, la classe `Level` contiene:

- Il `Survivor` attualmente presente nel livello.
- Una lista di `Zombie`, attivi e visibili sulla mappa.
- Una lista di `Munition`, ovvero i proiettili generati nel livello durante gli scontri.
- Informazioni dimensionali del livello (larghezza e altezza in centimetri).
- Un flag che indica se il livello è completo o meno.

Il cuore della logica di aggiornamento è rappresentato dal metodo `updateLevelState`, chiamato ciclicamente nel gioco. Questo metodo è responsabile dell'evoluzione dello stato del livello: aggiornamento delle entità presenti, generazione di nuovi nemici, ecc.



2.6 Fisica del Livello e Gestione del Comportamento degli NPC

La **componente fisica** rappresenta una parte fondamentale dell'architettura del livello, poiché si occupa della gestione della fisica di tutte le entità presenti al suo interno. In questo contesto, il termine "fisica" non si riferisce esclusivamente alla simulazione realistica delle leggi fisiche, ma alla logica che regola il movimento, le collisioni e le interazioni spaziali tra le entità di gioco.

In particolare, la fisica del livello gestisce:

- **Il movimento del Survivor**: aggiornando la sua posizione in base all'input ricevuto e assicurandosi che resti sempre all'interno dei limiti del livello.
- **Il comportamento delle munizioni**: ogni proiettile sparato dal Survivor è soggetto a un sistema fisico che ne aggiorna posizione e traiettoria in base alla direzione di fuoco.
- **Il sistema di collisioni**: la componente fisica contiene anche la logica per il rilevamento e la gestione delle collisioni tra gli oggetti del gioco, come proiettili e zombie. Questo avviene tramite l'uso dei `BoundingBox` assegnati a ogni entità, i quali vengono aggiornati ad ogni frame per verificare eventuali intersezioni.

- **Il rispetto dei vincoli spaziali**: il Survivor non può uscire dai confini prestabiliti del livello, e tale vincolo è imposto proprio dalla logica fisica associata al livello stesso.

Un aspetto particolarmente rilevante della fisica del livello è la presenza di un campo dedicato alla gestione del comportamento dei personaggi non giocanti (NPC), ovvero gli Zombie. Questo campo è un'istanza dell'interfaccia `AINPCBehavior`, la quale definisce il comportamento di movimento degli NPC all'interno del livello.

L'oggetto `AINPCBehavior` viene generato dinamicamente tramite una `FactoryAINPCBehavior`, secondo il principio di separazione delle responsabilità e dell'inversione del controllo. Questo approccio permette di assegnare in modo modulare e flessibile una logica di movimento diversa a seconda del tipo di Zombie o delle esigenze del livello.

In fase di aggiornamento del livello, ogni Zombie invoca il comportamento definito da `AINPCBehavior`, il quale calcola la direzione ottimale verso il target, cioè il Survivor, e ne aggiorna di conseguenza la posizione. Questo sistema consente di ottenere un movimento realistico e credibile degli Zombie, che sembrano "inseguire" attivamente il giocatore, aumentando così la profondità tattica e l'immersione nel gameplay.

In sintesi, la componente fisica del livello agisce come un *motore interno* che garantisce la coerenza e l'interattività del mondo di gioco, orchestrando il comportamento e le interazioni tra tutte le entità attive in modo modulare e facilmente estendibile.

2.7 Approfondimento sul Comportamento degli Zombie: Strategia Composita in `AIZombieBehavior`

A completamento della descrizione relativa al comportamento degli NPC, è opportuno analizzare più nel dettaglio il sistema che governa l'intelligenza artificiale degli `Zombie`, ovvero l'implementazione concreta dell'interfaccia `AINPCBehavior`.

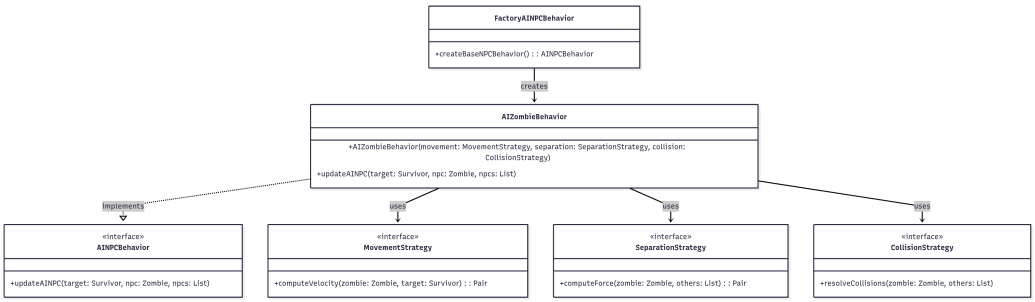
Nel mio progetto, la creazione del comportamento di base per gli NPC avviene tramite un metodo della factory chiamato `createBaseNPCBehavior()`, il quale restituisce un'istanza della classe `AIZombieBehavior`. Quest'ultima rappresenta una strategia composita, costruita unendo tre comportamenti distinti 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.


FactoryAINPCBehavior
+createBaseNPCBehavior() : : AINPCBehavior
creates
AIZombieBehavior
+AIZombieBehavior(movement: MovementStrategy, separation: SeparationStrategy, collision: CollisionStrategy)
+updateAINPC(target: Survivor, npc: Zombie, npcs: List)
implements
uses
uses
uses
«interface»
AINPCBehavior
+updateAINPC(target: Survivor, npc: Zombie, npcs: List)
«interface»
MovementStrategy
+computeVelocity(zombie: Zombie, target: Survivor) : : Pair
«interface»
SeparationStrategy
+computeForce(zombie: Zombie, others: List) : : Pair
«interface»
CollisionStrategy
+resolveCollisions(zombie: Zombie, others: List)

Capitolo 3

View e Gestione Grafica

3.1 Architettura della View nel Pattern MVC

Passiamo ora alla **View**, una delle componenti più complesse del progetto a causa della necessità di rispettare il pattern MVC e garantire una netta separazione tra la logica di gioco (*Model*) e la rappresentazione grafica. 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.

3.1.1 Unità di misura logica e scalatura grafica

Un aspetto fondamentale del progetto è l'utilizzo di un'unità di misura logica uniforme per tutto il modello: il centimetro (cm). Tutti gli elementi del gioco, come l'altezza e la larghezza del *Survivor*, dello *Zombie*, dei proiettili, così come le dimensioni del livello stesso, sono espressi in centimetri. Questo consente una modellazione coerente e realistica dello spazio di gioco.

Tuttavia, per poter visualizzare correttamente questo mondo logico nella *View*, è necessario adattare queste misure al contesto grafico. In particolare, la classe `SwingGraphicsSurvivor`, responsabile del disegno del *Survivor* utilizzando le librerie *Java Swing*, possiede un campo chiamato `Scaler`. Quest'ultimo viene inizializzato durante la creazione della `SwingGraphicsSurvivor` all'interno della classe `SwingSceneTutorial`.

Il compito dello `Scaler` è quello di effettuare una mappatura coerente tra le dimensioni logiche espresse in centimetri e le dimensioni del pannello grafico. In questo modo, ogni elemento viene disegnato mantenendo proporzioni e posizioni corrette rispetto allo spazio di gioco. Per esempio, se il *Survivor* si trova al margine destro del pannello grafico, ciò indica che si trova realmente alla fine del livello, secondo la sua dimensione logica.

Tale approccio garantisce una separazione netta tra il modello logico e la visualizzazione, nel rispetto del pattern MVC adottato.

3.1.2 Gestione dell'Input

Una parte cruciale del progetto è rappresentata dal sistema di gestione degli input, fondamentale per il controllo interattivo dei personaggi all'interno del gioco. Come precedentemente descritto, ogni oggetto di tipo `GameSurvivor` è associato a una componente di input, responsabile dell'aggiornamento dello stato sulla base delle interazioni utente.

Il metodo `update` di questa componente riceve in ingresso un oggetto di tipo `InputController`, la cui responsabilità è quella di notificare gli

input ricevuti e fornire i codici di input rilevati in quel ciclo di aggiornamento. Nel progetto attuale è stata implementata una classe specifica, `KeyboardInputController`, che si occupa della gestione degli input da tastiera. Tuttavia, la struttura del sistema è pensata per essere facilmente estendibile: sarà possibile, ad esempio, integrare un controller per un gamepad, mantenendo invariato il meccanismo di comunicazione con la componente di input del `GameSurvivor`.

L'istanza di `KeyboardInputController` viene passata anche alla classe `SwingSceneTutorial`, la quale ha il compito di rilevare gli eventi generati dalla tastiera attraverso il pannello Swing e notificare tali eventi al controller. In questo contesto, l'`InputController` funge da ponte tra la scena grafica e la componente di input del survivor, garantendo così una separazione logica e strutturale tra la vista e il modello, in conformità con il pattern MVC.

Questa architettura rende l'intero sistema flessibile e modulare, favorendo l'aggiunta di nuovi dispositivi di input e il riutilizzo delle componenti esistenti.

3.2 Gestione del Game Loop e Pattern State per la Gestione degli Stati di Gioco

Nel progetto, la logica principale di esecuzione del gioco è affidata alla classe `GameEngine`, che implementa il *game loop*. Questo ciclo continuo è responsabile di orchestrare in modo sincronizzato le fasi di acquisizione dell'input, aggiornamento della logica di gioco e rendering grafico, garantendo una frequenza di aggiornamento costante e una risposta fluida alle interazioni dell'utente.

Il metodo `mainLoop()` esegue iterativamente le seguenti operazioni:

- **Processamento dell'input**: attraverso il `GameStateManager`, viene delegata la gestione degli input all'attuale stato di gioco.
- **Aggiornamento della logica di gioco**: si invoca il metodo `updateGame()`, passando il tempo trascorso dall'ultimo aggiornamento per mantenere una simulazione temporale accurata.
- **Rendering**: si esegue la rappresentazione grafica dello stato di gioco corrente.
- **Sincronizzazione del frame rate**: grazie al metodo `waitForNextFrame()`, il ciclo viene rallentato in modo da rispettare un intervallo fisso (circa 25 ms, corrispondente a circa 40 FPS).

Questa struttura assicura un flusso di esecuzione regolare, separando chiaramente le responsabilità di input, elaborazione e output visivo.

Per gestire le diverse modalità e fasi del gioco, come ad esempio la modalità di gioco attiva o la schermata di fine partita, il progetto adotta il *pattern State*. Questo pattern permette di incapsulare il comportamento specifico di ogni stato in classi dedicate che implementano l'interfaccia `GameState`, la quale definisce i metodi fondamentali per la configurazione iniziale (`setUp()`), la gestione degli input (`processInput()`), l'aggiornamento della logica (`updateGame()`) e la visualizzazione (`render()`).

La classe `GameStateManager` funge da controller centrale che mantiene il riferimento allo stato attivo e delega a esso le chiamate di input, update e render. La transizione tra stati avviene tramite il metodo `setState()`, che garantisce l'inizializzazione del nuovo stato.

Tra le implementazioni principali dello stato di gioco vi è la classe `PlayState`, responsabile della gestione della modalità di gioco attiva. Essa si occupa di caricare i livelli tramite un `PlayStateManager`, processare gli input del giocatore, aggiornare la logica di gioco — inclusi il comportamento dei nemici e le condizioni di vittoria o sconfitta — e infine gestire la transizione allo stato di gioco terminato con un ritardo controllato per consentire eventuali animazioni o messaggi finali.

Lo stato `GameOverState` rappresenta invece la fase finale del gioco, in cui viene visualizzato un messaggio di conclusione, come ad esempio la vittoria di tutti i livelli o la morte del giocatore. Questo stato si limita alla visualizzazione della schermata di fine gioco e non gestisce input o aggiornamenti della logica, fungendo da schermata informativa e di attesa.

Questo approccio modulare e scalabile consente di estendere facilmente il comportamento del gioco aggiungendo nuovi stati, senza modificare la struttura generale del motore, migliorando così la manutenibilità e la chiarezza del codice.

Capitolo 4

Sviluppo

4.1 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.

4.1.1 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.

4.1.2 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.

4.1.3 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.

4.1.4 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.

4.1.5 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.

Capitolo 5

Commenti finali

5.1 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.

5.1.1 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.