



Tesina Finale di
Programmazione di Interfacce Grafiche e Dispositivi Mobili
Corso di Laurea in Ingegneria Informatica ed Elettronica – A.A. 2024-2025
DIPARTIMENTO DI INGEGNERIA

docente
Prof. Luca GRILLI

Aetherlum Survivor

applicazione desktop JFC/SWING



studente
364138 **Lorenzo Cleti** lorenzo.cleti@studenti.unipg.it

0. Indice

1 Descrizione del Problema	2
1.1 <i>Concetto alla Base dell'Applicazione</i>	2
1.2 <i>Riferimenti a Giochi Preesistenti</i>	3
2 Specifica dei Requisiti	4
2.1 <i>Funzionali</i>	4
2.2 <i>Non Funzionali</i>	5
2.3 <i>Di Gioco</i>	5
3 Progetto	7
3.1 Architettura del Sistema Software	7
3.2 Model	8
3.3 View	11
3.4 Controller	14
3.5 Util	15
3.6 Problemi Riscontrati	15
4 Appendice	17
5 Bibliografia	20

1. Descrizione del Problema

L’obiettivo di questo lavoro è lo sviluppo di un’applicazione desktop ispirata al genere “*bullet-hell*”.

L’applicazione sarà implementata utilizzando la tecnologia JFC/Swing in modo da favorire un’ampia portabilità su diversi sistemi operativi (piattaforme), riducendo al minimo eventuali modifiche al codice sorgente. Tuttavia, il codice prodotto sarà testato su un *Sistema Operativo Windows 10 (64-bit)*.

Di seguito sarà presentata una breve descrizione di **Aetherlum Survivor** a sua volta seguita dalle fonti di ispirazione del progetto, il quale sarà una versione molto semplificata di quest’ultime.

1.1 *Concetto alla Base dell’Applicazione*

Il giocatore controlla un personaggio che deve sopravvivere a onde incessanti di nemici statici e/o in movimento. Eliminare i nemici garantisce *Punti Esperienza* che permettono di migliorare alcune caratteristiche del personaggio, rendendolo così più potente. In aggiunta, durante il gioco sono generati casualmente ulteriori *Potenziamenti*. Il personaggio si può difendere in 2 modi: spostarsi evitando così il contatto con i nemici e/o ucciderli tramite il lancio automatico di incantesimi. L’obiettivo non è quello di uccidere tutti i nemici, che continuano ad apparire ininterrottamente fino al *Game Over*, bensì quello di sopravvivere più a lungo possibile.

1.2 Riferimenti a Giochi Preesistenti

Nella scelta del progetto sono stati presi come riferimento i giochi [Magic Survival](#) e [Vampire Survivors](#), il cui concetto e idea generale sono stati di ispirazione per la progettazione e creazione di [Aetherlum Survivor](#). Quest'ultimo è infatti una versione semplificata dei videogiochi sopracitati.

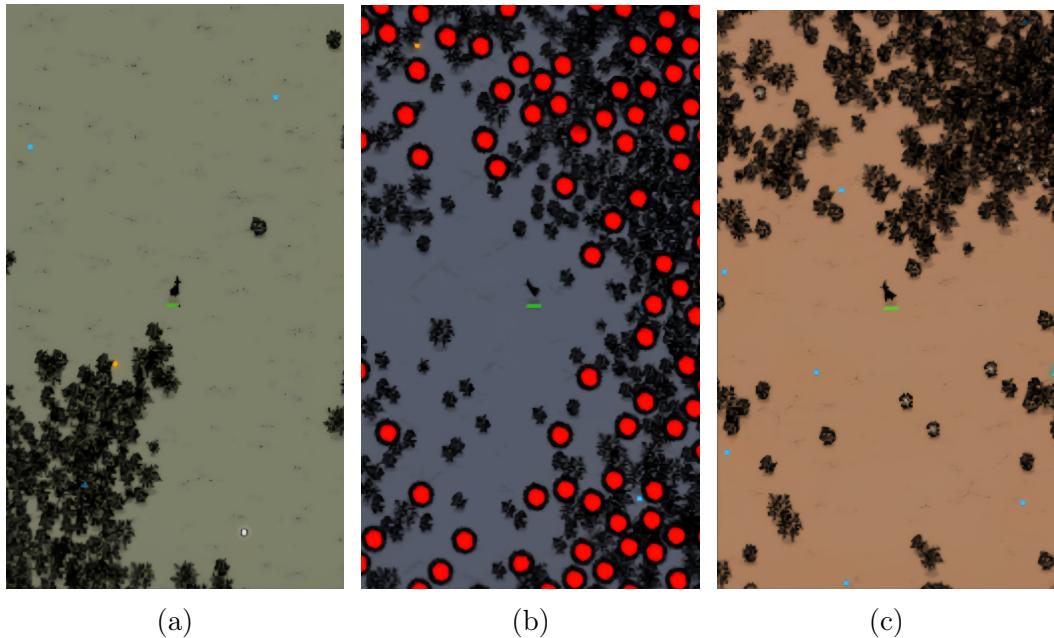


Figura 1.1: Tre schermate del videogioco *Magic Survival*. In ciascuna, al centro dello schermo si colloca il personaggio giocante, circondato da nemici in movimento. In particolare, in (a) si notano in alto delle sfere celesti che se raccolte conferiscono Punti Esperienza mentre in (b) e (c) ulteriori tipologie di nemici. Ciascuna schermata, inoltre, rappresenta uno scenario diverso, da cui derivano tali differenze.

2. Specifica dei Requisiti

Suddividiamo per praticità i requisiti nelle seguenti sezioni.

2.1 *Funzionali*

Sono elencati i requisiti che hanno un diretto riscontro da parte del giocatore.

- Presenza di un sottofondo musicale dedicato a ciascuna schermata ed effetti sonori. Il tutto è attivabile/disattivabile dal pannello delle impostazioni dell'applicazione.
- Controllo del personaggio tramite tastiera (sia FRECCE che tasti WASD).
- Timer che indica la durata della partita corrente.
- Presenza di un tasto pausa (P).
- Nel menù di pausa è presente la possibilità di interrompere la partita o accedere alle impostazioni.
- Nel menù di pausa sono mostrati gli attributi (vita, velocità, danno, ecc.) che caratterizzano il personaggio.
- Nel menù di pausa è mostrato come i potenziamenti ottenuti cambiano sui detti attributi. Le caratteristiche modificate sono evidenziate da un colore dorato.
- Sistema di Movimento multiassiale. È possibile spostarsi sull'asse X, sull'asse Y e sulle diagonali.
- Mappa infinita. Non è presente un limite alla mappa su cui il giocatore può spostarsi. Risulta infatti sempre centrato nello schermo e lo scenario sottostante viene proceduralmente ripetuto.

2.2 *Non Funzionali*

Sono elencati i requisiti di gioco ignoti all'utente.

- Architettura *Model-View-Controller*.
- Utilizzo del pattern *Singleton*.
- Applicazione progettata per operare con fluidità su hardware meno prestante.
È stata progettata su una piattaforma con le seguenti specifiche tecniche:

Componente	Specifiche
Sistema Operativo	Windows 10 (64-bit)
Processore	Intel(R) Pentium(R) CPU 2117U @ 1.80 GHz
Memoria RAM	4 GB (DDR3)
Scheda Grafica	Intel(R) HD Graphics (96 MB)

Tabella 2.1: Requisiti Minimi di Sistema per lo Sviluppo

- Controllo di collisioni tra molteplici entità: personaggio, attacchi - nemici, potenziamenti
- Suddivisione tra coordinate grafiche e logiche.

2.3 *Di Gioco*

Sono elencati i requisiti più strettamente legati alle meccaniche di gioco.

- Difficoltà variabile. Ciascuno scenario è caratterizzato da diverse tipologie di nemici presenti e da un numero variabile di nemici massimali contemporaneamente generati. Più precisamente sono presenti 3 scenari, ciascuno caratterizzato da tiles di background diverse.
 - Uno scenario di base, dove è presente solo un tipo di nemico e la quantità di generazioni è molto limitata.
 - Uno scenario di livello intermedio, in cui compaiono nuovi nemici e la generazione è incrementata.
 - Uno scenario di livello esperto, dove compaiono tutti i nemici del gioco in quantità ulteriormente incrementata.

- Più tipologie di nemici.
 - Un nemico base capace di muoversi.
 - Un nemico fisso, che funge da ostacolo inamovibile sulla mappa.
 - Un nemico più resiliente.
 - Un nemico più veloce e con meno vita.
- Meccaniche di Potenziamento del Personaggio. Viene data la possibilità di modificare tramite incrementi fissi (*ad esempio: +10Danno // +45Hp*) o percentuali (*ad esempio incremento 10% del danno corrente // aumento velocità del 5%*) gli attributi del personaggio.
- Durante il gioco, oltre ai nemici, compaiono anche degli oggetti raccoglibili che conferiscono Punti Esperienza, i quali permettono di salire di livello più velocemente.

3. Progetto

Viene ora descritta la struttura dell'applicazione realizzata, presentando prima l'architettura software per poi illustrarne nel dettaglio i blocchi funzionali che la compongono.

3.1 Architettura del Sistema Software

Per la realizzazione di *Aetherlum Survivor* è stata scelta l'architettura *Model-View-Controller* e si è utilizzato il pattern di progettazione *Singleton*, garantendo così che le classi chiave abbiano una sola istanza accessibile globalmente. Tali classi, le cui funzionalità sono ulteriormente approfondite in seguito, sono:

- `Controller.java`
- `KeyHandler.java`
- `Model.java`
- `View.java`

Una panoramica generale sui vari package e sul loro funzionamento è la seguente.

Il package **controller** lancia l'applicazione e presenta metodi di comunicazione tra classi e package. Il suo ruolo è duplice: da un lato inoltra le richieste scambiate tra *model* e *view*, dall'altro gestisce operazioni logiche richiamando in metodi unici sia il *Model* che il *View*, coordinandone così l'azione. Un aspetto importante è che il **Controller** invia informazioni e richieste esclusivamente alle classi *Model* e *View*, le quali a loro volta le gestiscono richiamando le altri classi dei rispettivi package. Al contrario, ciascuna di queste può invece richiedere direttamente informazioni al **Controller** o, nel caso di quelle del *model*, anche inviare richieste. Questa scelta evita la necessità di introdurre metodi nel *Model* e nel *View* di solo inoltro verso il **Controller**, riducendo così la presenza di codice superfluo.

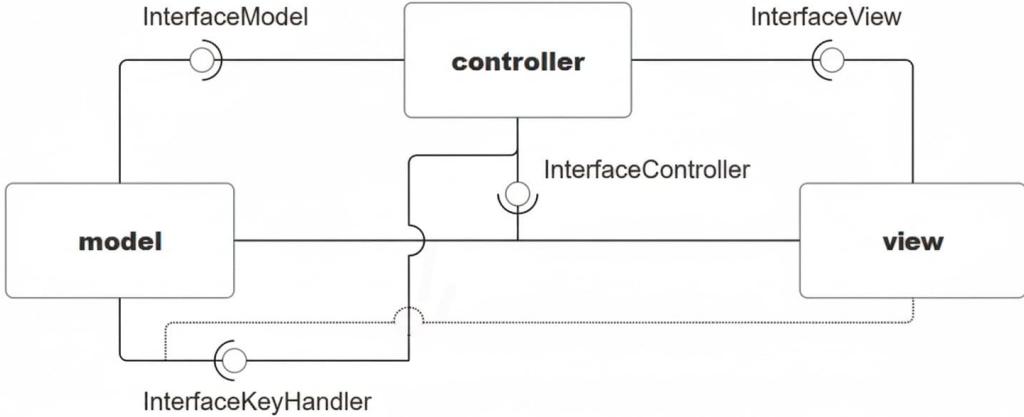


Figura 3.1: Architettura *Model View Controller* (*MVC*) del sistema Aetherlum Survivor. Sono evidenziate le interfacce esposte dai singoli moduli architetturali e dunque anche le direzioni in cui vanno i flussi informativi. Il collegamento tra `InterfaceKeyHandler` e `view` è tratteggiato poiché quest'ultima si limita ad aggiungere KeyHandler (che implementa `KeyListener`) al `JPanel` su cui avviene la partita.

Il package **model** presenta esclusivamente le classi necessarie alla gestione della logica dell'applicazione e un sottopackage *loop*, il quale custodisce la classe di gestione del *gameloop*, implementato tramite `javax.swing.Timer`.

Il package **view** raccoglie le classi dedicate alla sola visualizzazione dell'applicazione. Viene istanziato un unico `JFrame` su cui si alternano i vari `JPanel`. Inoltre gestisce l'input dell'utente fintanto che questo riguarda l'interazione con i menù privi di logica, alternando opportunamente i pannelli corrispondenti. Quando la partita viene avviata, inizia la comunicazione con *controller* per la gestione logica dell'input.

È inoltre presente un ulteriore package **util**, il quale presenta classi di utilità indipendenti che sono state ritenute più adeguate in un ambiente neutrale. Tra di esse si trovano classi per la custodia e gestione ordinata di dati tramite sottoclassi pubbliche apposite. Tale suddivisione ha consentito di mantenere negli altri package solo le classi strettamente legate al comportamento dell'applicazione.

3.2 Model

Segue, suddiviso in 2 parti per migliorarne la leggibilità, il diagramma UML del package *model*. Nell'appendice il diagramma completo (Vedi Fig 4.1).



Figura 3.2: Prima parte del diagramma di classe del package *model*. Mostra le componenti che gestiscono il gameloop.

In Fig. 3.2 è mostrato il **Model** e il **GameLoop**, mentre in Fig. 3.3 è mostrata **Entity** e le classi che la estendono.

N.B. La necessità di spezzare il diagramma ha reso impossibile realizzare le righe di associazione tra **Model** e le classi che estendono **Entity**, ma le due comunicano.

È ora esposta, grazie anche ad alcuni esempi, la *relazione* presente tra le varie classi.

Il **Model** crea un'istanza di **GameLoop** quando la richiesta di inizio partita giunge tramite il **Controller** dal **GamePanel**. Ad ogni **ciclo** il *model* viene aggiornato tramite `update()` e il **Controller** richiede l'aggiornamento del *view*. Il **Model** possiede dei metodi appositi per estrarre solo le informazioni che saranno utili per il rendering grafico di ciascuna entità; queste sono contenute nella classe del package *util EntityLogicalData*. Sempre durante l'inizializzazione è istanziato **Player** e sono preistantiate liste di tutti gli oggetti che potranno essere generati: ciò garantisce il riutilizzo degli stessi ed evita la necessità di istanziarne di nuovi durante il gameloop. È durante il metodo `update()` che ciascuna entità evolve il proprio stato, solo se **attiva**. Alcuni metodi, come `despawn()`, sono comuni poiché

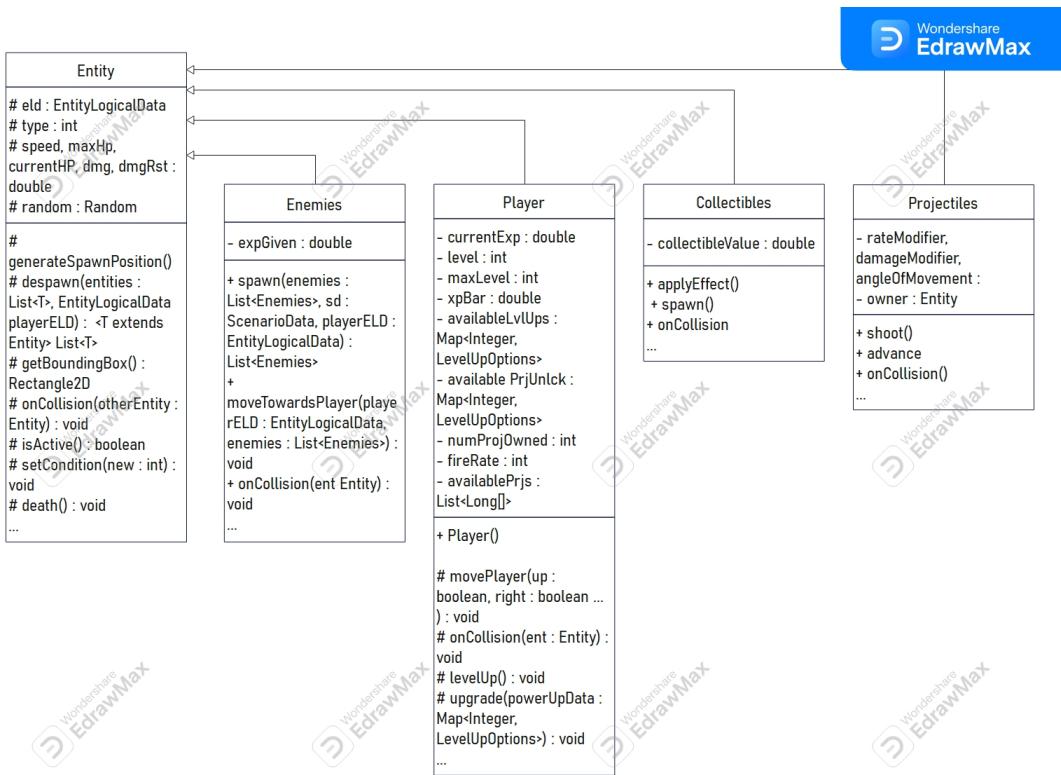


Figura 3.3: Seconda parte del diagramma di classe del package *model*. Mostra le componenti che gestiscono le entità che interagiscono durante il gameloop.

ereditati o derivanti da *override* da **Entity**, mentre altri sono specifici delle singole classi; i più importanti sono riportati in Fig. 3.3. Sono di particolare interesse, al fine di comprendere la logica alla base della gran parte di essi, i seguenti metodi:

- **enemies.spawn(List<Enemies> enemies, ScenarioData sd, EntityLogicalData playerELD)**
List<Enemies> sarà restituita aggiornata, **ScenarioData** contiene informazioni sullo scenario selezionato per determinare numero massimo di nemici e tipi e **EntityLogicalData** è relativo a **Player**: ciascuna coordinata di spawn e despawn è infatti basata sulla sua posizione, in quanto il mondo di gioco è “infinito”; tale illusione è generata facendo basare ogni evento sulla sua posizione logica. Il metodo verifica se può far comparire nemici e in caso affermativo ne sceglie randomicamente il numero, la posizione di spawn e il tipo tra i disponibili, assegnando gli adeguati valori ad un elemento inattivo della Lista.
- **player.levelUp()**

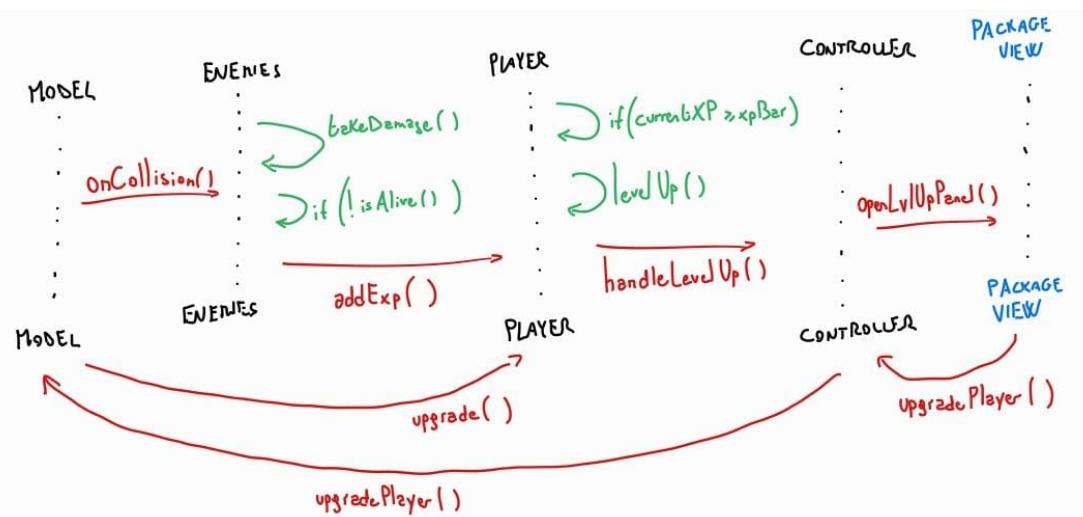


Figura 3.4: Diagramma di sequenza UML di tipo “sketch” disegnato a mano. Mostra la serie di eventi che porta a `player.levelUp()`; non sono indicati i parametri dei metodi né ciò che avviene all’interno del *view*.

È un metodo la cui esecuzione dipende a sua volta da `checkCollision()`. Avviene infatti se a seguito della collisione tra un proiettile con `owner == player` e un nemico, quest’ultimo è morto e tramite gli *XP* conferiti il giocatore sale di livello; è qua interessante osservare che i proiettili non sono considerati sempre del giocatore, qualora si desideri in futuro implementare nemici che ne sparano. Quando viene avviato il metodo sono scelti potenziamenti casuali, con la garanzia che a ogni livello multiplo di `LevelUpData.LVL_PRJ_UNLOCK_INTERVAL` sia sbloccato un nuovo tipo di proiettile; il Controller mette poi in pausa il gameloop e richiede al View l’apertura dell’apposito JPanel. Effettuata la scelta, il Controller riavvia il gameloop e viene eseguito il metodo `player.upgrade(Map<Integer, LevelUpOptions> powerUpData)`. Vedi Fig 3.4.

È importante ribadire che il campo logico di gioco non è dunque limitato da una mappa o dimensioni finite, bensì il mondo che circonda il giocatore viene creato e arbitrato continuamente a partire dalle sue coordinate logiche.

3.3 View

Anche nel caso di *view* il diagramma UML è stato suddiviso in 2 per fini di migliore leggibilità. Nell’appendice il diagramma completo (Vedi Fig 4.2).

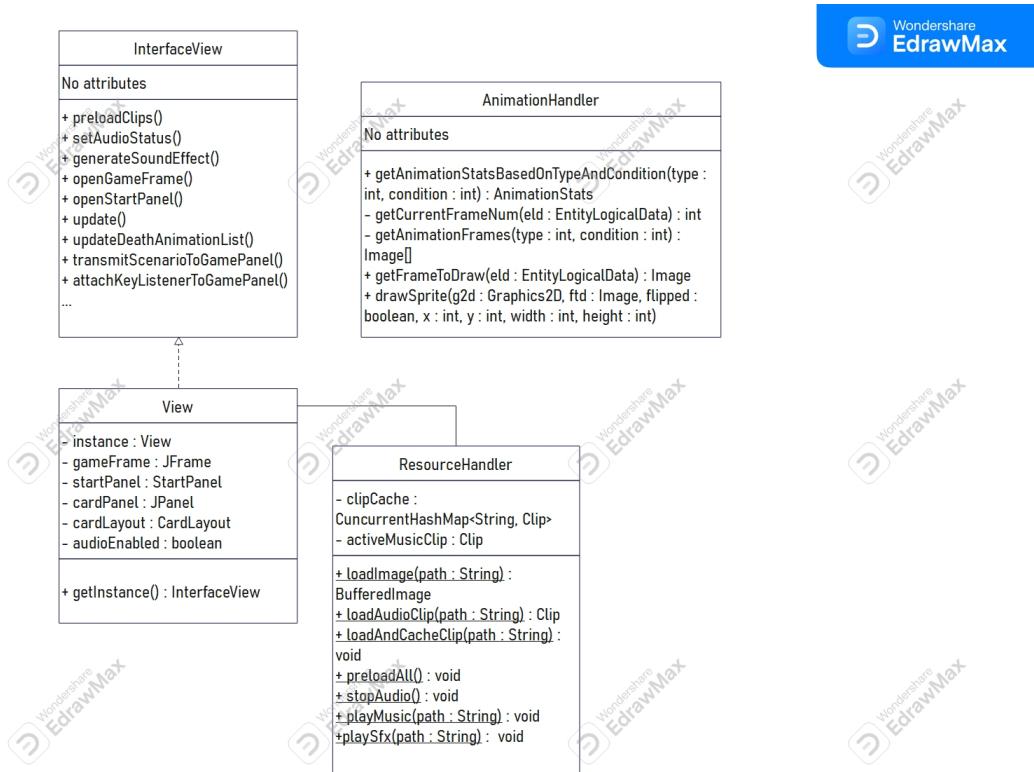


Figura 3.5: Prima parte del diagramma di classe del package *view*. Mostra le componenti che coordinano i pannelli e gestiscono il rendering e l’audio.

In Fig. 3.5 è mostrato il **View** e le classi di gestione audio e grafica, mentre in Fig. 3.6 le classi che estendono **JPanel** e si alternano sul **JFrame**.

N.B. **View** e **ResourceHandler** comunicano con tutte le classi che estendono **JPanel**, mentre **AnimationHandler** solo con **GamePanel**.

Il package *view* si occupa del rendering grafico, dell’output audio a seguito delle azioni del giocatore e della lettura e comunicazione dell’input tramite **Controller**. In particolare ciò avviene tramite **JButton** e **KeyHandler** qualora ci si trovi in gioco nel **GamePanel**. Su un unico **JFrame** si alternano i vari **JPanel** per mezzo di **CardLayout**. Alcuni pannelli sono di sola visualizzazione e interazione tramite **JButton**, mentre altri vengono aggiornati a seconda delle azioni dell’utente. Il più importante sotto questo aspetto è il **GamePanel**, il quale si occupa del rendering grafico di tutte le entità che abitano il gameloop e dello scorrimento infinito del background. Ciascuno di questi metodi fa uso di **playerELD** e di **convertLogicalToGraphical()** per rappresentare ciascun elemento in coordinate grafiche, basate sulla posizione di **player**. È interessante l’attributo **deathAnimation**, il quale presenta una lista di elementi che viene con-

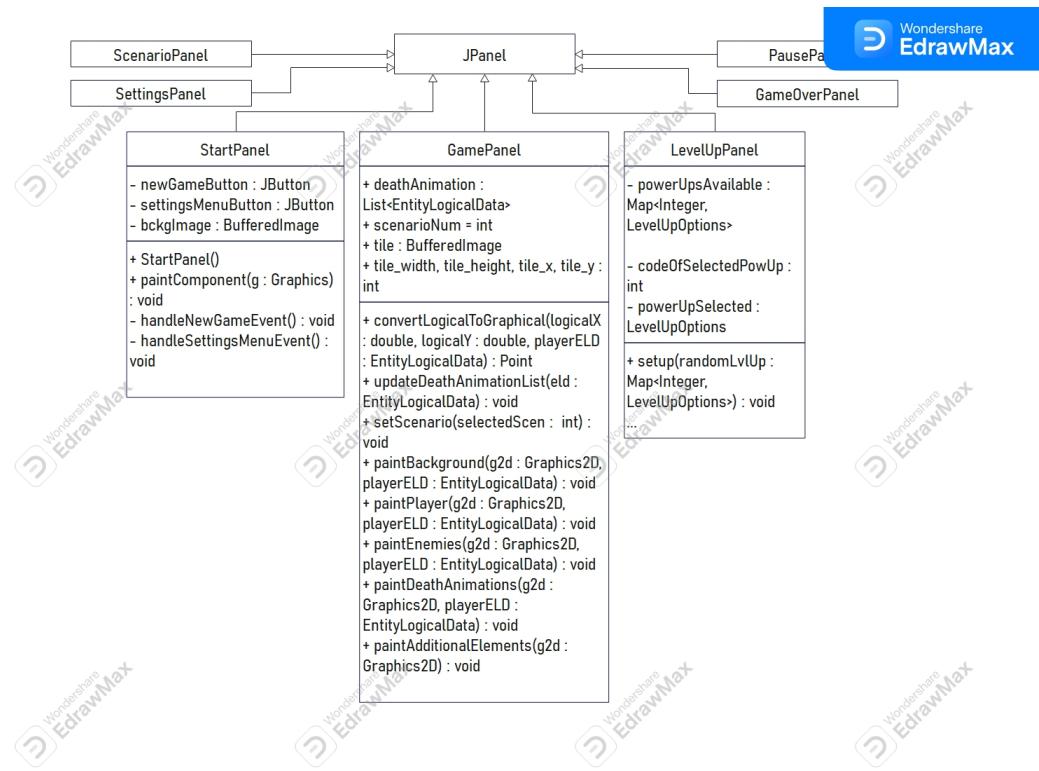


Figura 3.6: Seconda parte del diagramma di classe del package *view*. Mostra i vari JPanel presenti.

tinuamente aggiornata e rimossa e serve a riprodurre le animazioni di morte delle varie entità. Di estrema rilevanza, sono:

- **ResourceHandler.java**

Incaricata del caricamento e della gestione di immagini e audio. Sotto l'aspetto di gestione audio, questa avviene esclusivamente tramite libreria `javax.sound` e la ripetizione di tracce `.wav`. La fluida esecuzione dell'applicazione è resa possibile dal metodo `preloadAll()`, che all'avvio dell'applicazione precarica tutte le clip in `clipCache`: questo espediente permette di poter continuare a lavorare esclusivamente sull'`EventDispatchThread` senza rischio che questo si blocchi nel momento in cui va riprodotto un audio, poiché non sarà necessario caricarlo sul momento. Appositi metodi si occuperanno poi della loro riproduzione in loop o singola.

- **AnimationHandler.java**

Si occupa della gestione delle animazioni, tramite una serie di metodi che gli permettono di estrarre lo **sprite** e il **frame** basandosi su `currentClockCycle`,

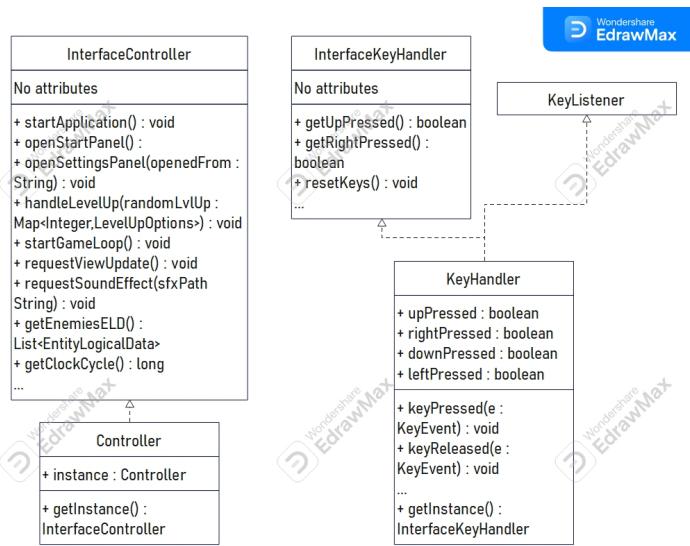


Figura 3.7: Diagramma UML *controller*

`startingClockOfCondition`, `type` e `condition` dell’entità. I vari metodi combinati culminano in `getFrameToDelete()`, che può poi essere disegnato con il giusto orientamento tramite `drawSprite()`.

3.4 Controller

In Fig. 3.7 è illustrato il diagramma UML del package *controller*.

Seguono le descrizioni e i ruoli delle classi che ne sono parte. **Controller** dispone di metodi che permettono di richiamare direttamente solo **Model** e **View**, che poi gestiscono autonomamente tali richieste. Al contrario, ciascuna classe può istanziare il **Controller** e chiamarne i metodi desiderati. **KeyHandler** è la classe dedicata alla lettura di input da tastiera, dunque implementa **KeyListener** e viene collocato su **GamePanel** dal metodo `View.attachKeyListenerToGamePanel()`. È di fondamentale importanza il metodo `resetKeys()`, il quale è stato introdotto per resettare le variabili che salvano gli input di pressione e rilascio nei metodi di **Controller** `handleLevelUp()` e `handlePauseGame`. Tali chiamate evitano che, alla riapertura del **GamePanel**, **player** continui a muoversi in autonomia nella direzione precedente l’apertura dei menù.

3.5 Util

Il package *util* presenta classi di ausilio per la conservazione e l'organizzazione di dati e costanti.

- `ResourcePaths`, suddivisa in due sottoclassi `Audio` e `Images`, salva i percorsi relativi necessari ad accedere alle risorse.
- `EntityLogicalData` salva e gestisce gli attributi utili anche al rendering grafico e audio di `Entity` e le classi che la estendono, con appositi metodi `getter` e `setter`. **Posizione logica**, **direzione**, **condizione** e **tipo** sono solo alcune di esse.
- Le classi di custodia dati sono composte da costanti pubbliche statiche e, quando necessario, una sottoclasse di ausilio che le racchiude e una `Map` le cui chiavi sono il `type` dell'entità.

3.6 Problemi Riscontrati

Si riportano di seguito i principali problemi riscontrati nello svolgimento del progetto *Aetherlum Survivor*.

- Nella fase iniziale, comprendere a fondo la separazione tra package dell'architettura e come strutturare le relazioni tra i vari elementi. In un primo momento, avevo inserito due `Controller`, uno per il *view* e uno per il *model*; in un secondo momento ho deciso di unificarli per ridurre la quantità di codice duplicato: molti metodi si ritrovavano infatti a inoltrare solamente richieste da un `Controller` all'altro.
- Gestire la presenza di molteplici entità senza doverne ogni volta istanziare di nuove. Questa problematica e la soluzione di creare liste di oggetti attivi/disattivi mi ha portato a dover modificare vari metodi per supportare ora liste e non più oggetti singoli.
- Gestione di effetti sonori ripetuti nel breve termine. Spesso i metodi, nonostante la presenza dello stop, non interrompevano e riavviavano sufficientemente in tempo le `Clip`. Sono riuscito ad ovviare al problema tramite la combinazione di metodi `flush()` e `drain()`.
- Organizzazione e rendering della `JTable` nel menù di pausa. L'assenza di metodi diretti per la personalizzazione delle celle ha richiesto un tempo significativo per la finalizzazione di questa schermata.

- Essendomi occupato dell’ambito di rendering grafico solo a seguito dell’implementazione del *model*, è stato necessario modificare le classi per agevolarmi nella loro rappresentazione grafica. Tra le modifiche: aggiungere *type* anche nell’**EntityLogicalData** e implementare e aggiungere tutta la logica di gestione delle *condition* e della *direction*, aggiungendo alcune righe di codice a vari metodi.
- Gestire le animazioni di morte. Restando su questo argomento, in un primo momento avevo compiuto un’ampia modifica a ciascuna entità al fine che queste non si potessero muovere o interagire quando *condition == EntityData.DYING*. Prima di ultimarla e notando alcuni malfunzionamenti che avrebbero richiesto ulteriori modifiche a vari metodi, ho scelto di separare la logica dalla visualizzazione delle creature morenti. Quando una creatura muore viene impostata come *inattiva*, con status DYING e aggiunta dal **Controller** alla **List** di creatura morenti nel **GamePanel**. Quando l’animazione termina, la voce viene rimossa dalla lista. In questo modo l’oggetto dell’entità può sin da subito essere riutilizzato senza che il *model* debba attendere che il *view* termini l’animazione
- Trovare *sprite* con un aspetto congruo le une alle altre e file audio che fossero adeguatamente ripetibili in loop.
- In generale, cercare di mantenere l’applicazione il più leggera possibile per evitare di intasare l’**Event Dispatch Thread** su cui, tramite **Timer**, scorre anche **GameLoop**. E’ proprio per evitare ciò che sono ricorso all’escamotage del preload delle **Clip** audio. Se ciò fosse accaduto sarebbe stato necessario istanziare nuovi **Thread** e modificare ampiamente il codice; in un primo momento avevo infatti iniziato la programmazione con un **Thread** separato per il **GameLoop**, ma erano sorte una serie di problematiche riguardo la concorrenza tra operazioni e ho dunque deciso di provare a far girare tutto su **Thread** singolo.

4. Appendice

- I diagrammi UML sono stati realizzati con [edrawmax](#)[8]. La versione gratuita permette di scaricare i documenti realizzati solo con filigrana, che dunque è presente in questo documento.
- Ai fini di garantire la lettura dei diagrammi UML più ampi, sono stati sopra spezzati e qua riportati nella loro interezza. Per il *model* vedi Fig. 4.1, mentre per il *view* vedi Fig. 4.2

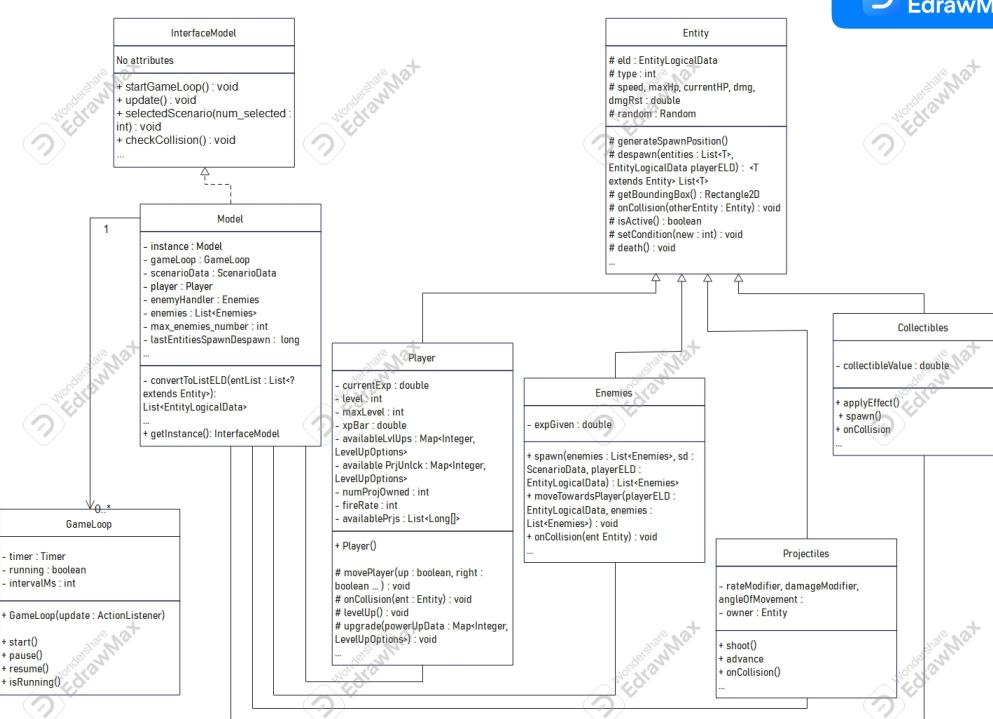


Figura 4.1: Diagramma UML *model* completo.

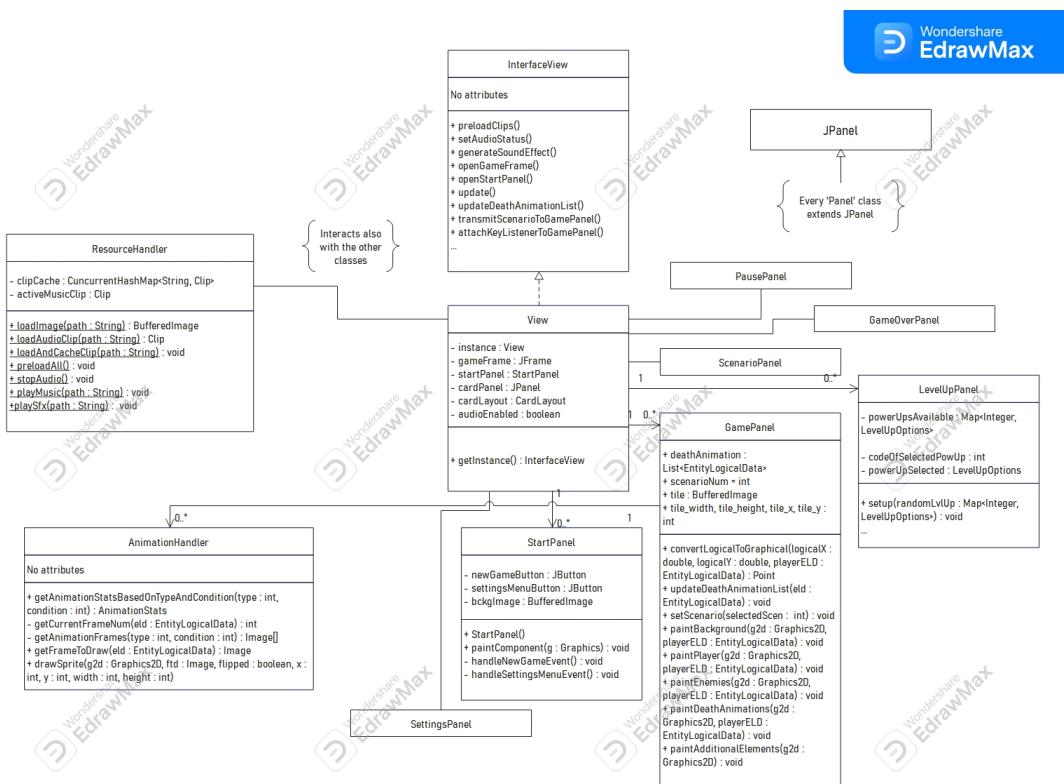


Figura 4.2: Diagramma UML *view* completo.

5. Bibliografia

- [1] Oracle. Java API and Documentation
<https://docs.oracle.com/javase/8/docs/api/>
- [2] Dispense Unistudium del corso
'Progettazione di Interfacce Grafiche e Dispositivi Mobili'
- [3] Fonti .mp3 audio
<https://pixabay.com/>
- [4] Conversione da .mp3 a .wav
<https://convertio.co/it/>
- [5] Fonti sprite e tileset
<https://craftpix.net/>
<https://itch.io/>
- [6] Disegno tiles dei background
<https://www.spritefusion.com/>
- [7] Lettura dimensioni delle sprite
<http://spritecow.com/>
- [8] Realizzazione diagrammi UML
<https://www.edrawmax.com/>
- [9] Selezione colore
<https://imagecolorpicker.com/>
- [10] Immagini di sfondo dei menù
<https://it.pinterest.com/>
<https://www.deviantart.com/>