

Relazione per
“vampire.io”

Bassi Ciro, Foschini Paolo, Testa Alessandro

17 giugno 2025

Indice

1	Analisi	3
1.1	Descrizione e requisiti	3
1.2	Modello del Dominio	5
2	Design	7
2.1	Architettura	7
2.2	Design dettagliato	11
2.2.1	Bassi Ciro	11
2.2.2	Foschini Paolo	17
2.2.3	Testa Alessandro	20
3	Sviluppo	23
3.1	Testing automatizzato	23
3.2	Note di sviluppo	25
3.2.1	Bassi Ciro	25
3.2.2	Utilizzo di GridBagLayout	25
3.2.3	Foschini Paolo	25
3.2.4	Testa Alessandro	26
4	Commenti finali	27
4.1	Autovalutazione e lavori futuri	27
4.1.1	Bassi Ciro	27
4.1.2	Foschini Paolo	28
4.1.3	Testa Alessandro	28
4.2	Difficoltà incontrate e commenti per i docenti	28
4.2.1	Bassi Ciro	28
4.2.2	Foschini Paolo	29
A	Guida utente	30
A.1	Selezione del salvataggio	30
A.2	Menu principale	30

A.3	Selezione del personaggio	31
A.4	Partita	32
A.5	Shop	34
A.6	Scoreboard	35
A.7	Salvataggi	36

Capitolo 1

Analisi

1.1 Descrizione e requisiti

vampire.io è un action game in cui il giocatore controlla un personaggio che attacca automaticamente mentre affronta ondate continue di mostri. L'obiettivo è sopravvivere il più a lungo possibile, fino a un massimo di cinque minuti, al termine dei quali apparirà il “Reaper”, un nemico (quasi) invincibile che segna la fine della partita.

Sconfiggendo i nemici, il giocatore guadagna punti esperienza che gli permettono di salire di livello e scegliere se sbloccare nuove armi o potenziare quelle già ottenute, fino a un massimo di tre armi attive per partita. Durante la partita, i nemici possono anche rilasciare monete e cibo: le prime servono per acquistare potenziamenti permanenti (come aumento della salute, velocità o resistenza ai danni) o sbloccare nuovi personaggi, ciascuno con caratteristiche e armi iniziali uniche; il cibo, invece, consente di recuperare salute.

I progressi vengono salvati in modo permanente, così da avere anche la possibilità di riprenderli in un secondo momento.

Requisiti funzionali

- Il giocatore deve poter muovere il personaggio per la mappa tramite input da tastiera;
- I nemici devono comparire in modo casuale sulla mappa (al di fuori della visuale di gioco) e attaccare il giocatore, aumentando il loro livello con il passare del tempo;
- Il giocatore deve infliggere danno ai nemici tramite attacchi automatici;

- Durante la partita, il giocatore deve ottenere potenziamenti che migliorano le caratteristiche delle proprie armi;
- Il punteggio finale e le statistiche del giocatore devono essere mostrati al termine della partita.

Requisiti non funzionali

- Possibilità di scegliere tra più personaggi giocabili, ognuno con un'arma di partenza;
- Possibilità di memorizzare su file le statistiche delle partite, i potenziamenti e i personaggi ottenuti, visualizzando anche una classifica dei risultati;
- L'interfaccia di gioco dovrà essere ridimensionabile;
- Possibilità di mettere in pausa il gioco in qualsiasi momento della partita.

1.2 Modello del Dominio

Il modello del dominio della partita ruota attorno a 5 entità principali:

- Personaggi;
- Nemici;
- Oggetti collezionabili;
- Armi;
- Attacchi.

Durante l'analisi sono state definite diverse interfacce per rappresentarle correttamente, e per permettere, in eventuali aggiornamenti futuri, di avere una buona estensibilità e flessibilità:

- Positionable: rappresenta tutte le entità posizionabili all'interno della mappa 2D. Fornisce metodi per modificare le coordinate 2D e per calcolare la distanza rispetto ad un altro Positionable;
- Movable: rappresenta tutte le entità in grado di muoversi per la mappa. Fornisce metodi per modificare la direzione e la velocità, per applicare lo spostamento e per calcolare la posizione futura (necessaria per evitare le collisioni tra nemici);
- Collidable: rappresenta tutte le entità in grado di essere colpite. Ogni Collidable ha una propria hitbox (l'area in cui può essere colpito) descritta da un'area circolare determinata da un raggio. Fornisce metodi per verificare le collisioni tra gli oggetti sulla mappa e gestirle;
- Living: rappresenta tutte le entità viventi (personaggi e nemici). Fornisce principalmente metodi per curare o ferire;
- Collectible: rappresenta tutte le entità che possono essere raccolte durante una partita (monete, punti esperienza, ecc.);
- Attack: rappresenta tutti i tipi di attacco (proiettili o attacchi ad area). Fornisce metodi per definire la logica del movimento;
- Weapon: rappresenta le armi utilizzate dal personaggio per attaccare i nemici. Fornisce metodi per aggiornare lo stato, in modo da generare attacchi automaticamente.

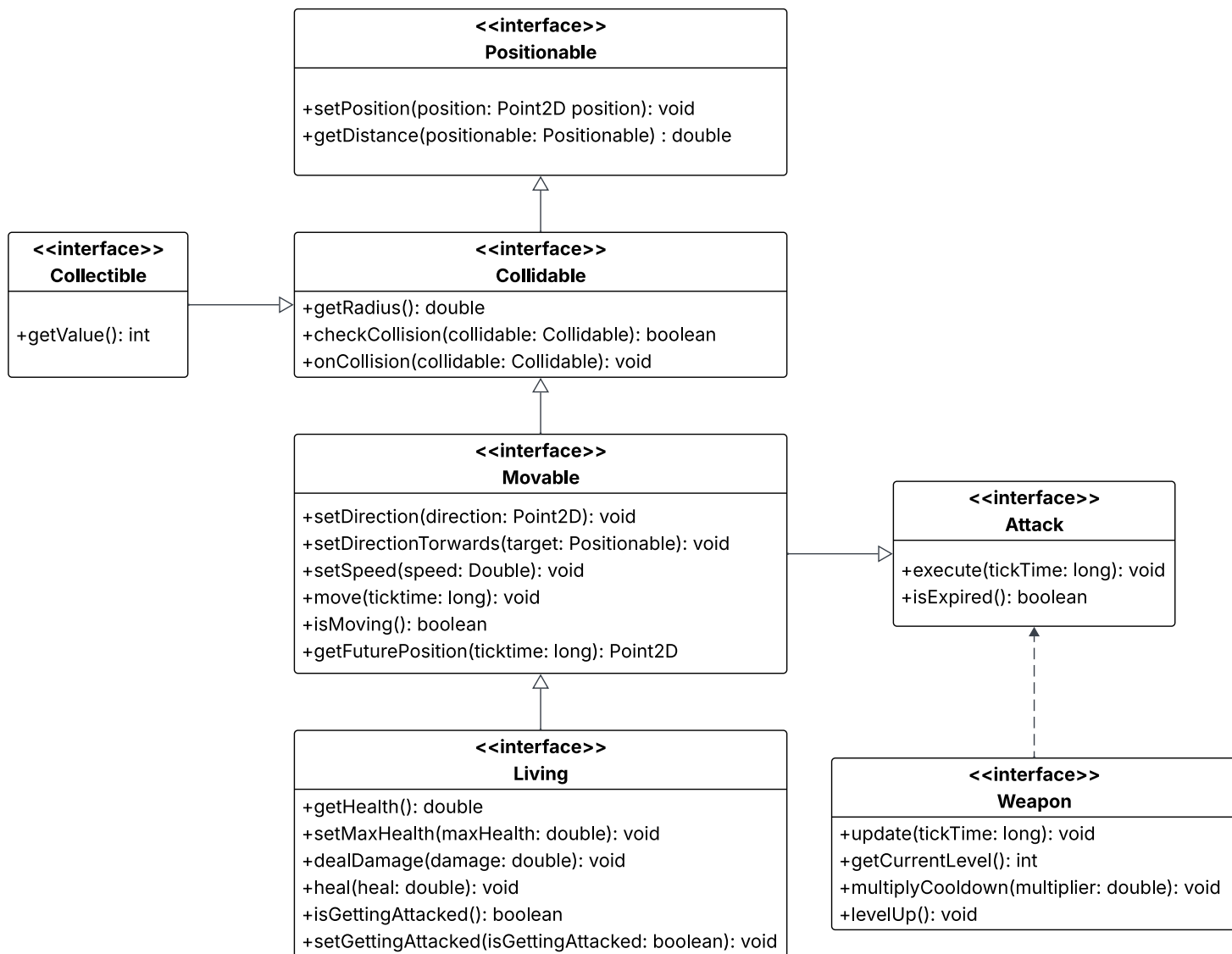


Figura 1.1: Schema UML rappresentante le principali interfacce delle entità in gioco.

Capitolo 2

Design

2.1 Architettura

L'architettura del gioco *vampire.io* segue il pattern architetturale MVC (Model View Controller).

Questo pattern suddivide l'applicazione in tre livelli:

- Il Model gestisce la logica del dominio applicativo, senza dipendere dall'interfaccia utente;
- La View rappresenta il livello di presentazione, responsabile della visualizzazione dei dati, includendo tutti gli elementi grafici e fornendo al Controller i metodi per aggiornarla;
- Il Controller gestisce l'aggiornamento ciclico del Model e della View. Riceve l'input dall'utente e invoca le operazioni necessarie sul Model, aggiornando di conseguenza la View con i nuovi dati.

Questo tipo di architettura rende il codice più modulare e manutenibile, permettendo eventualmente, per esempio, di sostituire agevolmente l'implementazione della View senza modificare il Model e il Controller.

Il Controller è stato suddiviso in:

- GameControllerImpl: gestisce gli errori dell'applicazione e coordina l'inizializzazione di tutti i componenti del sistema;
- GameLoopManager: responsabile del ciclo principale del gioco. Richiama periodicamente l'aggiornamento di Model e View;
- InputHandler: cattura eventi da tastiera e mantiene una coda dei tasti premuti;
- InputProcessor: elabora gli input ricevuti da InputHandler, trasformandoli in comandi di gioco;
- ScreenManager: gestisce i cambi di schermata dell'interfaccia grafica. Fornisce anche un metodo per ottenere la schermata precedente;
- ListenerInitializer: fornisce metodi statici di supporto per l'inizializzazione tutti i listener della View.

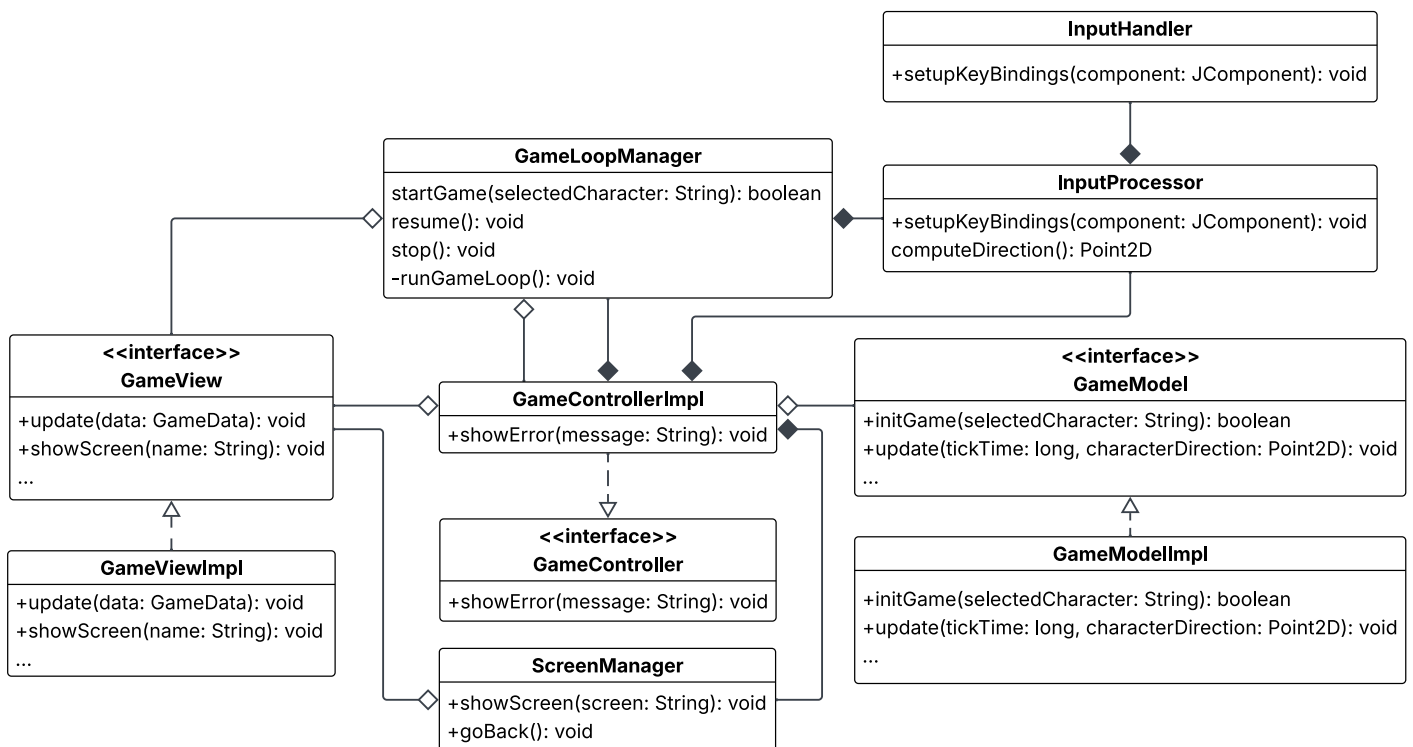


Figura 2.1: Schema UML del diagramma architetturale, in cui viene evidenziata la separazione tra le parti di Model, View e Controller e le interazioni tra essi.

Per la parte relativa alla View è stata scelta, per semplicità, la libreria Swing, sufficiente per le esigenze del progetto. L'implementazione è stata suddivisa in più componenti specializzate:

- **GameViewImpl**: rappresenta il punto di comunicazione tra il Controller e i vari componenti della View. Riceve i dati e le istruzioni dal Controller e li inoltra ai vari manager (FrameManager, ImageManager, ecc.), delegando a ciascuno le operazioni specifiche;
- **FrameManager**: gestisce l'inizializzazione della finestra (JFrame), e il cambio di schermata (JPanel);
- **ImageManager**: si occupa del caricamento delle immagini utilizzate nell'applicazione. Per ottimizzare l'efficienza e migliorare le prestazioni, adotta un sistema di caching, conservando le immagini già caricate per evitarne il caricamento ripetuto;
- **AudioManager**: gestisce la riproduzione della musica. In futuro sarebbe eventualmente possibile, senza particolari modifiche al codice, l'aggiunta di effetti sonori;
- **ListenerBinder**: offre metodi statici per associare gli ActionListener, forniti alla View da ListenerInitializer (Controller), ai corrispondenti pulsanti grafici (JButton).

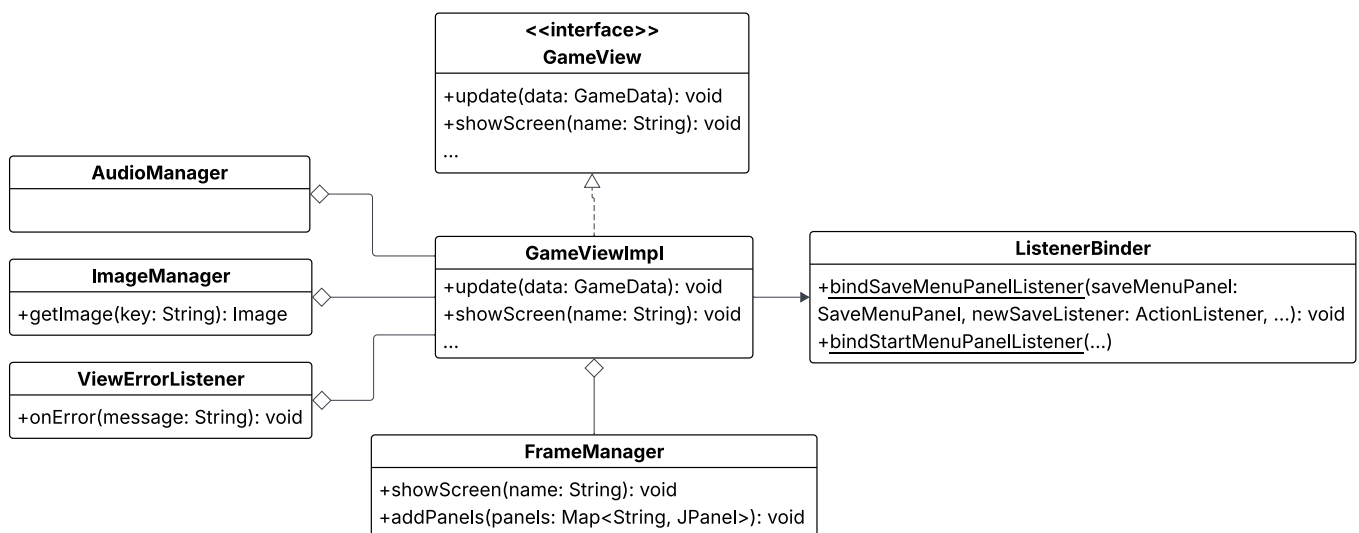


Figura 2.2: Schema UML, rappresentante struttura della View.

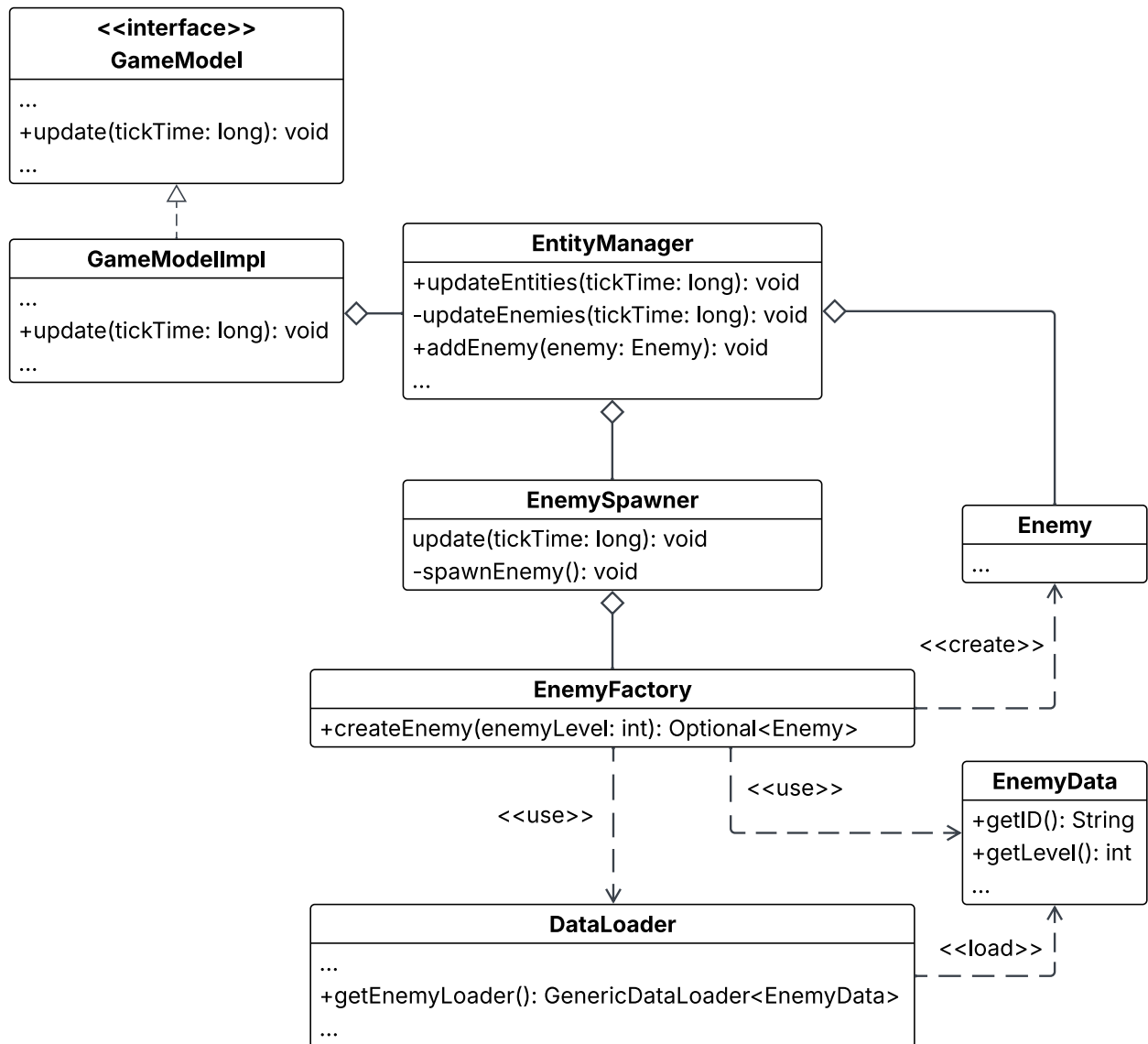
L'interazione tra Controller, Model e View avviene nel seguente modo:

- Il GameControllerImpl inizializza Model e View, configurando tutti i listener per i pulsanti e per la gestione degli errori e delegando la gestione del gioco al GameLoopManager;
- Il GameLoopManager esegue il ciclo di gioco, processando gli input attraverso l'InputProcessor e aggiornando il Model con la direzione del personaggio e il tempo trascorso dall'ultimo aggiornamento;
- Il Model aggiorna lo stato del gioco (posizioni delle entità, salute, generazione di nemici, ecc.);
- Il Controller utilizza DataBuilder per convertire i dati ottenuti dal Model in un formato compatibile con la View, che provvede poi ad aggiornare;
- Lo ScreenManager gestisce la navigazione tra le diverse schermate (menu, gioco, pausa, ecc.) mantenendo una cronologia delle schermate precedenti;
- Durante la partita, l'InputProcessor traduce gli eventi di tastiera in direzioni di movimento e segnala eventuali richieste di pausa al GameLoopManager.

2.2 Design dettagliato

2.2.1 Bassi Ciro

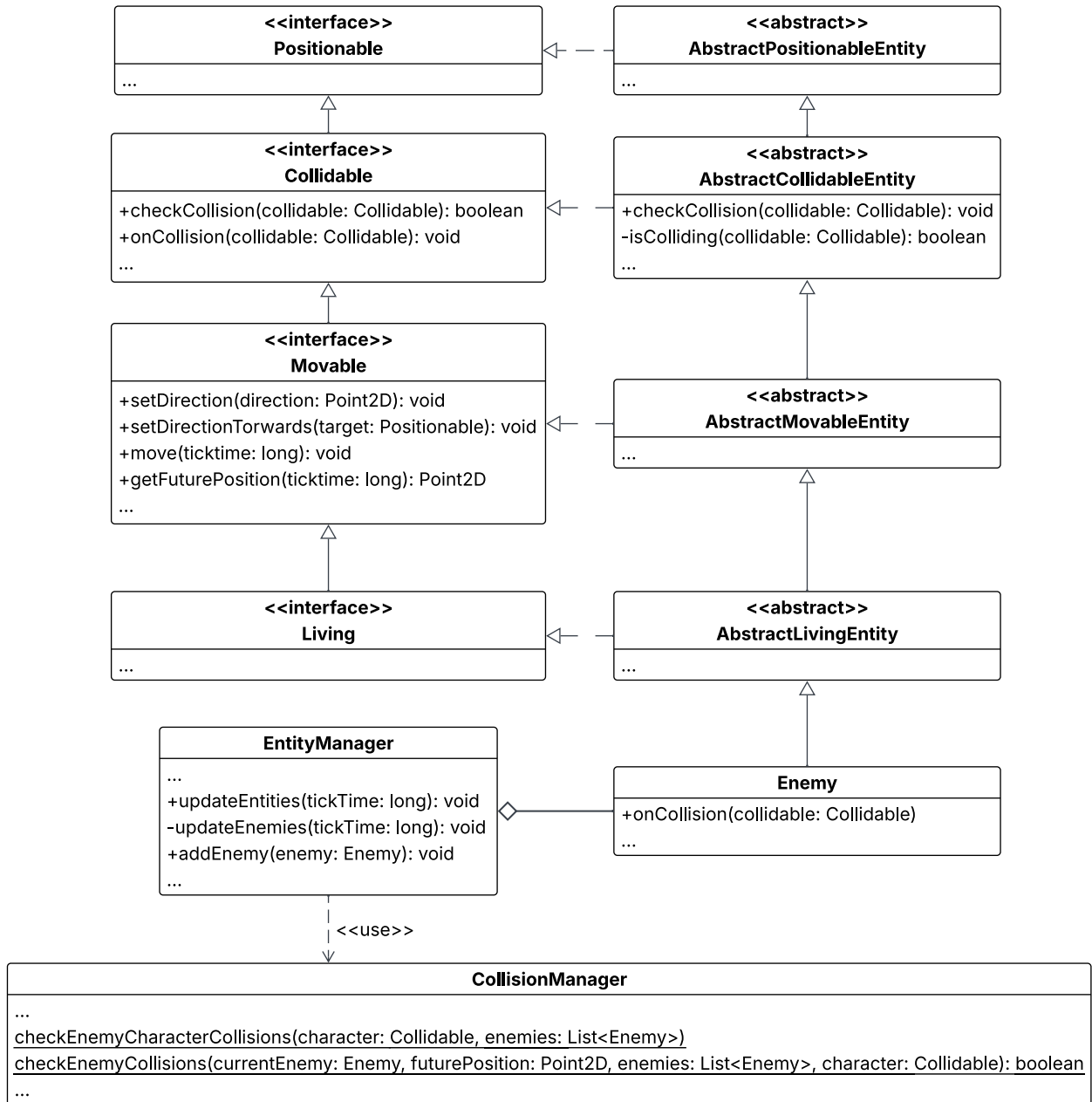
Generazione dei nemici



Problema La generazione dei nemici deve essere gestita in modo tale che, con il progredire della partita, aumentino sia il numero di nemici presenti che il loro livello di difficoltà.

Soluzione All'interno di `EntityManager`, classe responsabile dell'aggiornamento di tutte le entità presenti nella partita, è stato integrato un componente `EnemySpawner`, incaricato di determinare dinamicamente quando e quanti nemici generare in base al tempo di gioco trascorso. Questo consente di aumentare progressivamente la difficoltà della partita in maniera fluida. Per rispettare il Single Responsibility Principle (SRP) e migliorare la modularità e l'estensibilità del codice, la creazione effettiva delle istanze dei nemici è stata delegata a una `EnemyFactory`. Quest'ultima restituisce un'istanza di nemico del livello richiesto, posizionandolo casualmente al di fuori del campo visivo, evitando sovrapposizioni con altri nemici già presenti sulla mappa. Il caricamento dei dati necessari alla creazione dei nemici è affidato al componente `DataLoader` (realizzato da Alessandro Testa), che legge le informazioni da file JSON e fornisce una lista di oggetti `EnemyData`. Ciascun oggetto `EnemyData` contiene le specifiche dettagliate di un tipo di nemico, tra cui ID, livello, vita, velocità, danno e dimensione. Questo approccio (applicato anche dagli altri membri del gruppo per il caricamento dei dati di altre entità come `Character`, `Weapon`, ecc.) rende semplice l'aggiunta di nuovi nemici o la modifica di quelli esistenti senza dover intervenire direttamente sul codice applicativo.

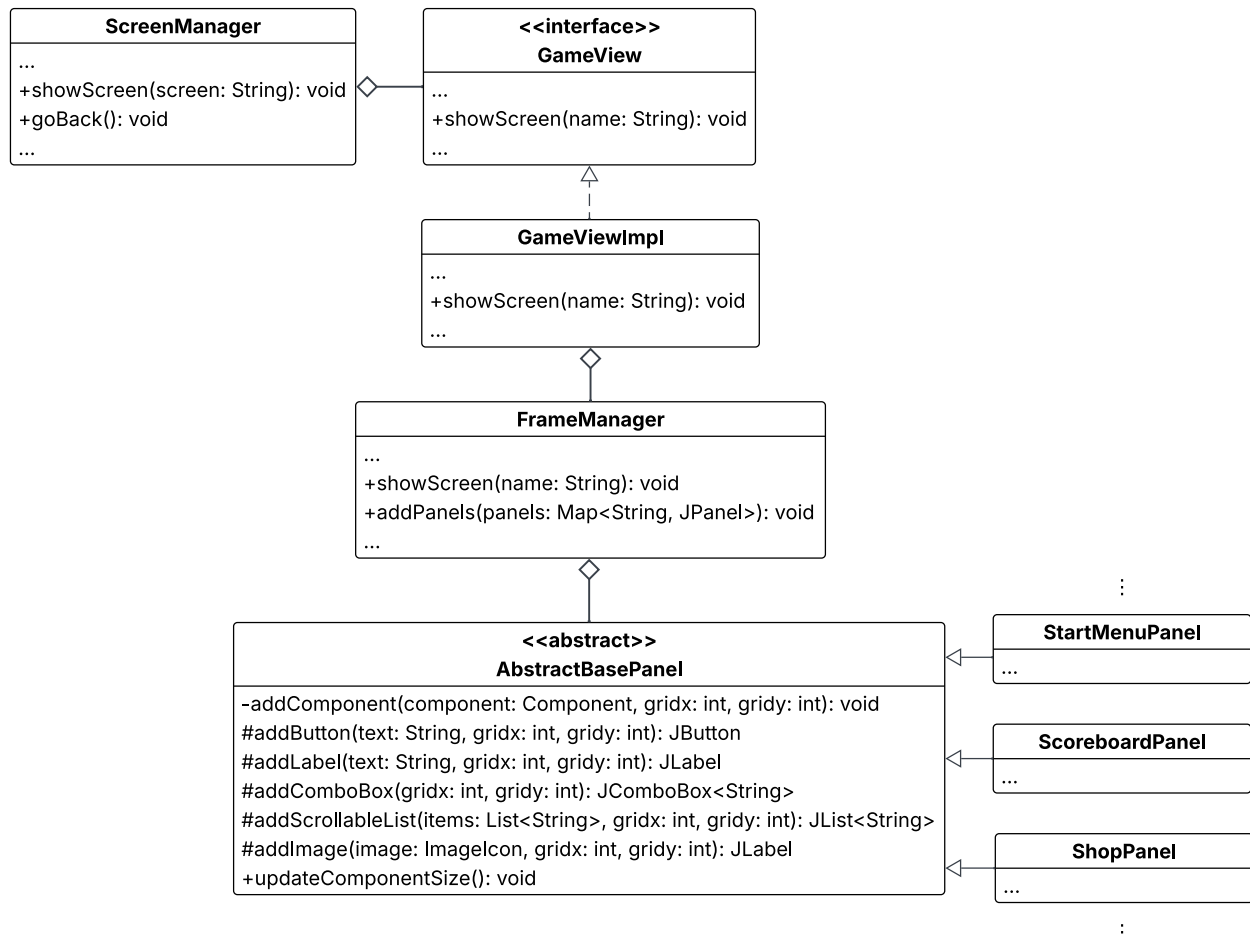
Logica dei nemici



Problema Una volta generato, ogni nemico deve inseguire il personaggio controllato dal giocatore, infliggendogli danno in caso di collisione. Tuttavia, è necessario garantire che i nemici non si sovrappongano né al personaggio né tra loro.

Soluzione La classe `Enemy` eredita funzionalità da diverse classi astratte, tra cui `AbstractPositionableEntity`, `AbstractCollidableEntity`, `AbstractMovableEntity` e `AbstractLivingEntity`, che forniscono rispettivamente il supporto alla gestione della posizione, delle collisioni, del movimento e della vita. In particolare, `AbstractCollidableEntity` implementa la logica per la rilevazione delle collisioni e definisce il metodo astratto `onCollision()`, applicando il *Template Method Pattern*. Questo consente alle sottoclassi, come `Enemy`, di ridefinire soltanto il comportamento specifico in caso di collisione, mantenendo invariata la struttura complessiva. Ad esempio, `Enemy` implementa `onCollision()` per infliggere danno al personaggio nel momento in cui viene rilevato un contatto. Il movimento dei nemici è gestito ciclicamente da `EntityManager` che, per ciascun nemico, imposta la direzione verso il personaggio principale tramite il metodo `setDirectionTowards()` (implementato in `AbstractMovableEntity`). Successivamente, sulla base delle collisioni previste con altri nemici o con il personaggio stesso, decide se procedere al movimento. La rilevazione delle collisioni è centralizzata nella classe `CollisionManager`, che include metodi specifici come `checkEnemyCharacterCollisions()`, incaricato di verificare se un nemico ha colpito il personaggio, e `checkEnemyCollisions()`, utilizzato per evitare sovrapposizioni tra nemici. Quest'ultimo metodo sfrutta la posizione futura dei nemici per prevedere eventuali collisioni.

Gestione delle schermate di menu



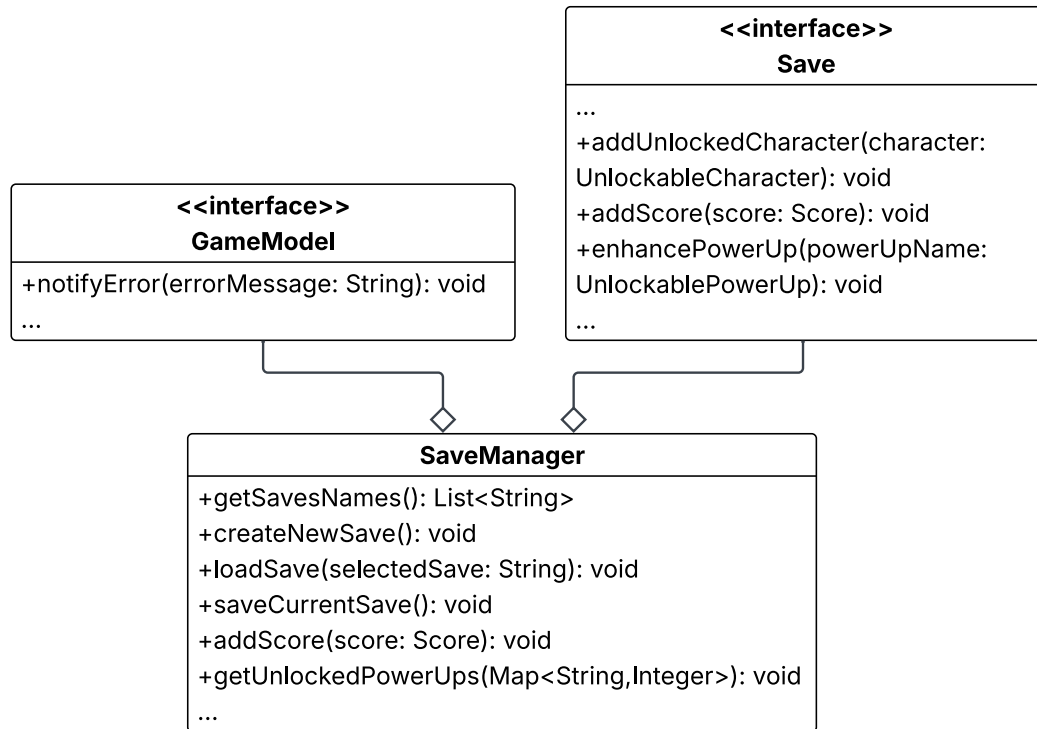
Problema L'applicazione deve gestire il passaggio tra diverse schermate (menu principale, il gioco, il negozio, ecc.) e consentire all'utente di tornare alla schermata precedente (tramite il tasto 'BACK'). Questa funzionalità deve essere implementata evitando la creazione di dipendenze tra Controller e View, e minimizzando la ripetizione di codice sia nella definizione dei pannelli (JPanel) sia nell'aggiunta dei relativi componenti grafici.

Soluzione Per gestire la navigazione tra le diverse schermate, è stata introdotta la classe **ScreenManager** (Controller), che memorizza l'elenco di schermate visualizzate per tenere traccia dello storico delle schermate visitate. Ogni volta che si accede a una nuova schermata tramite il metodo **showScreen()**, questa viene memorizzata; il metodo **goBack()**, chiamato dall'ActionListener associato al pulsante "BACK", consente invece di rimuovere la

schermata corrente e tornare a quella precedente. La classe `FrameManager` (View), oltre a occuparsi dell'inizializzazione della finestra principale (`JFrame`), si occupa anche della visualizzazione e della sostituzione dei pannelli, su richiesta di `ScreenManager`. Questo approccio segue il `Single Responsibility Principle` (SRP), assegnando responsabilità distinte a ciascuna classe, e il `Dependency Inversion Principle` (DIP), migliorando la modularità e la manutenibilità del codice. Per facilitare la creazione dei pannelli e ridurre la duplicazione del codice, è stata sviluppata la classe astratta `AbstractBasePanel`, che estende `JPanel` e fornisce metodi riutilizzabili per l'aggiunta di componenti comuni come pulsanti, etichette, combo box, liste e immagini. Tutti i pannelli specifici dell'applicazione (come `StartMenuPanel`, `ShopPanel`, `ScoreBoardPanel`, ecc.) estendono questa classe, favorendo la riutilizzabilità del codice e rispettando il principio DRY (`Don't Repeat Yourself`), che altrimenti verrebbe ripetutamente violato.

2.2.2 Foschini Paolo

Sistema di salvataggio



Problema Il gioco presenta la possibilità di salvare i propri progressi e di poterli riprendere in seguito.

Soluzione Per il salvataggio dei progressi di gioco è stato adottato un sistema basato su file: ogni salvataggio è rappresentato da un singolo file contenente tutte le informazioni rilevanti. Questi file sono memorizzati all'interno di una cartella dedicata, situata nella cartella **home** dell'utente.

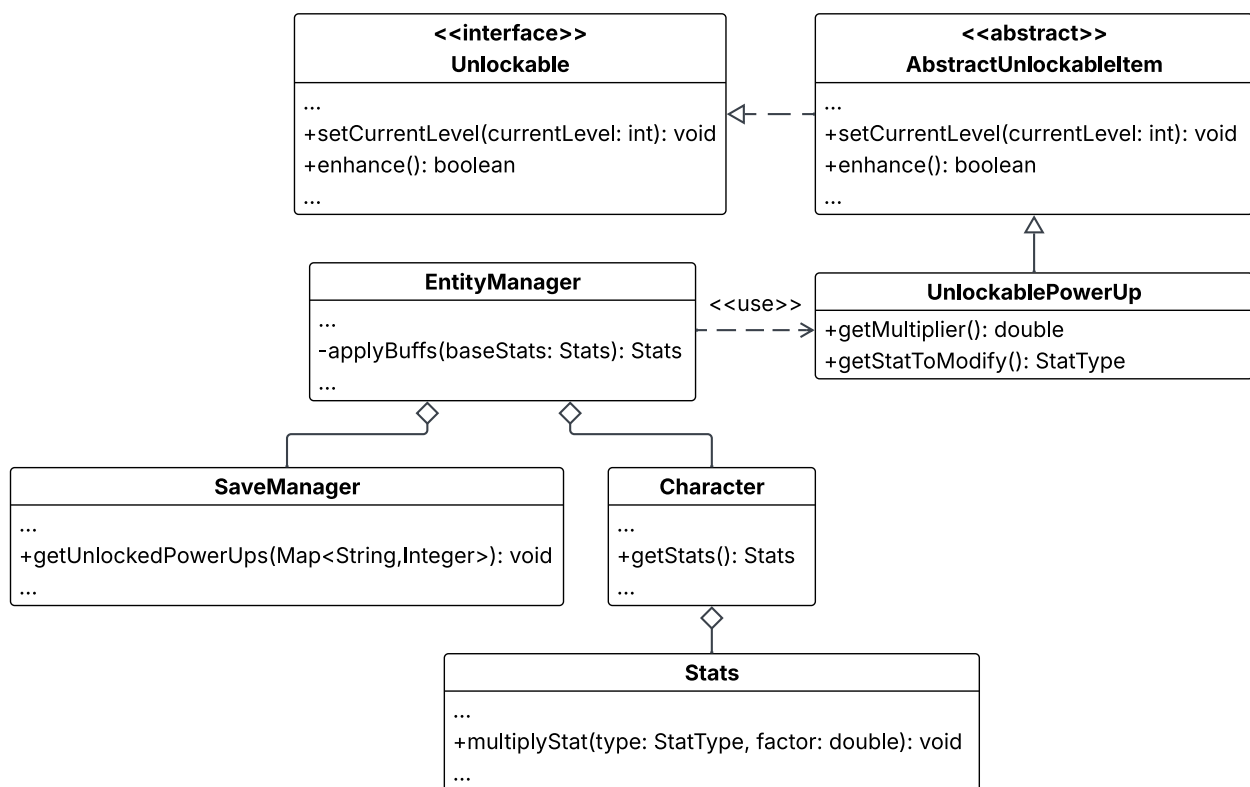
Quando il giocatore seleziona l'opzione **NEW GAME**, viene generato un nuovo file di salvataggio che include dati come:

- i personaggi e i potenziamenti sbloccati;
- la quantità di denaro posseduta;
- i punteggi ottenuti nelle varie partite.

Al termine della sessione di gioco, i metodi del componente **SaveManager** vengono invocati per aggiornare le informazioni nel file di salvataggio corrente.

Al successivo avvio dell'applicazione, selezionando **LOAD GAME**, l'utente può scegliere uno dei salvataggi disponibili. Il nome del salvataggio selezionato viene comunicato dal **Controller** al **SaveManager**, che si occupa di caricare il relativo file. Ho scelto di utilizzare interfacce anziché implementazioni concrete per aderire al DIP (Dependency Inversion Principle) e ridurre l'accoppiamento tra le classi, facilitando così eventuali estensioni e modifiche future del sistema.

Potenziamento delle statistiche del personaggio



Problema Il gioco fornisce la possibilità di potenziare le statistiche del proprio personaggio comprando oggetti nel negozio, semplificando la sopravvivenza nelle successive partite.

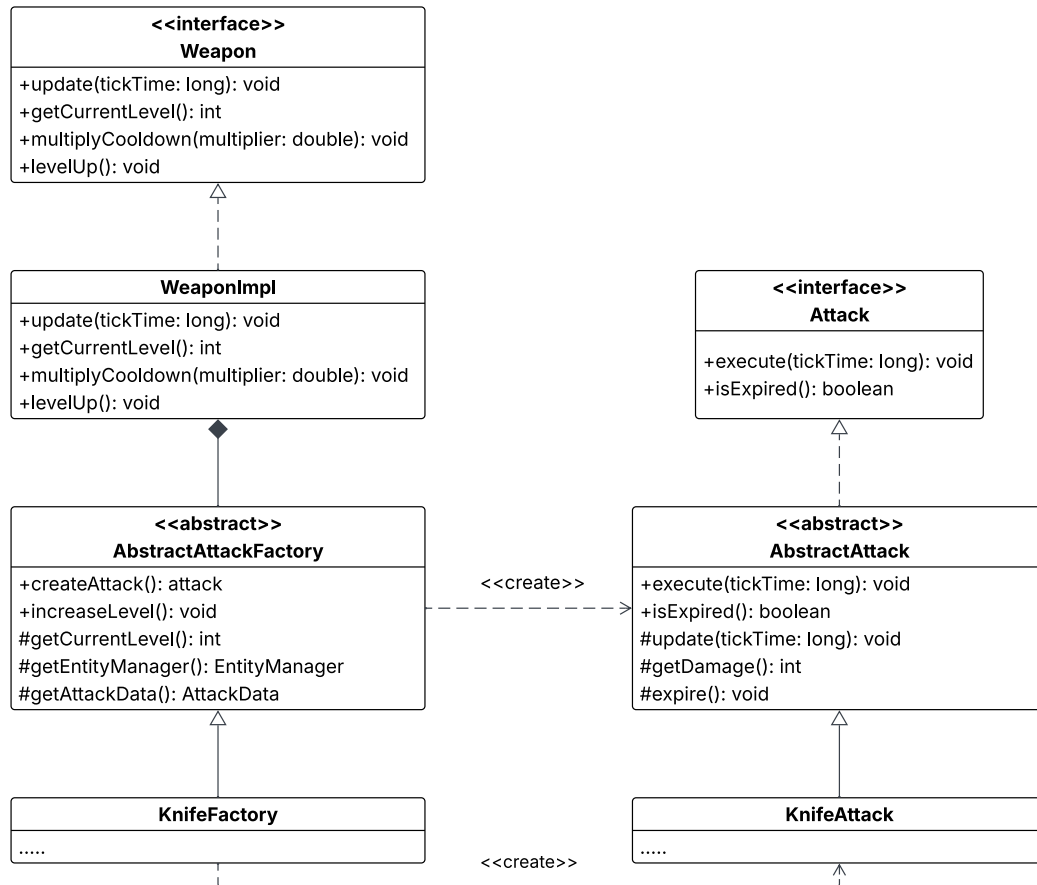
Soluzione Nel gioco ogni power-up è progettato per migliorare una specifica statistica del personaggio, come il danno, la velocità di movimento o la frequenza degli attacchi. Ogni power-up può essere potenziato su più livelli, aumentando progressivamente i benefici in partita.

All'avvio di una nuova partita, l'EntityManager applica automaticamente i bonus forniti dai power-up acquistati dal giocatore tramite il metodo `applyBuffs()`. Per farlo, attraverso il SaveManager, recupera dal salvataggio la lista dei power-up sbloccati e i relativi livelli.

Per ciascun power-up viene quindi calcolato un moltiplicatore in base al livello raggiunto, che viene applicato direttamente alle statistiche base del personaggio prima della sua creazione da parte dell'EntityManager. In questo modo, il personaggio riflette sin da subito i potenziamenti comprati.

2.2.3 Testa Alessandro

Gestione di armi e attacchi

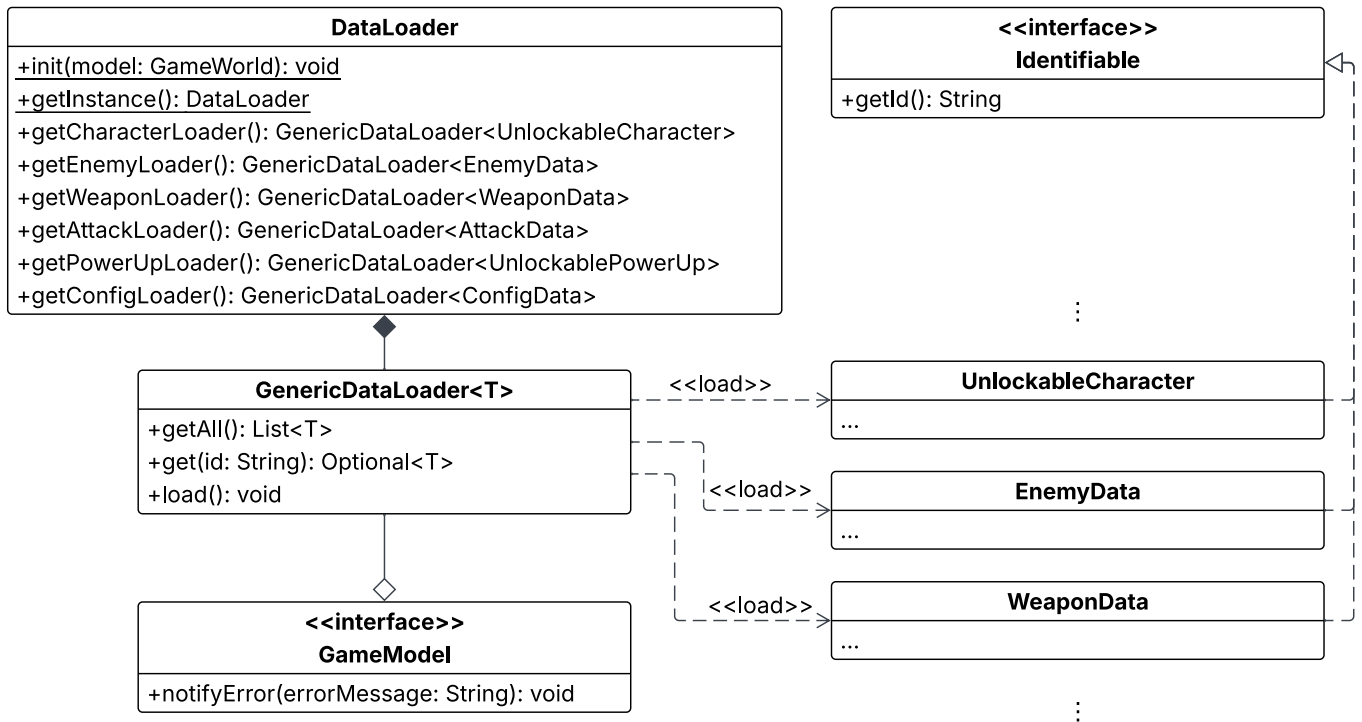


Problema Il sistema deve supportare diversi tipi di armi con tipi di attacco differenti (proiettili direzionali, attacchi ad area, attacchi che seguono i nemici). Le armi devono inoltre gestire cooldown, livellamento e statistiche modificabili.

Soluzione Il sistema delle armi è stato progettato utilizzando il pattern Factory Method per la creazione degli attacchi. Il pattern Factory Method è usato nella classe astratta AbstractAttackFactory che definisce il metodo createAttack(), implementato diversamente da ogni factory concreta (KnifeFactory, MagicWandFactory, GarlicFactory, SantaWaterFactory). È stato utilizzato anche il pattern Template Method nella gerarchia AbstractAttack: il metodo execute() definisce l'algoritmo comune (aggiornamento e chiamata

a `update()`), mentre ogni attacco implementa un comportamento specifico nel metodo `update()`. La classe `WeaponImpl` implementa l'interfaccia `Weapon` e gestisce il cooldown e il livellamento, delegando la creazione degli attacchi alla propria `AbstractAttackFactory`. Questa separazione rispetta il Single Responsibility Principle e il Dependency Inversion Principle: la `weapon` si occupa solo della gestione dei tempi e del livellamento, mentre la `factory` gestisce la creazione degli attacchi specifici, inoltre la `weapon` dipende dall'astrazione `AbstractAttackFactory` e non dalle sue implementazioni permettendo di cambiare il tipo di attacco senza modificare il codice della `weapon`. Il metodo `update()` della `weapon` gestisce il tempo tra un attacco e l'altro e utilizza la `factory` per generare nuovi attacchi. Il pattern `Factory Method` è stato applicato attraverso la classe astratta `AbstractAttackFactory` che definisce il metodo `createAttack()`, implementato diversamente da ogni `factory` concreta (`KnifeFactory`, `MagicWandFactory`, `GarlicFactory`, `SantaWaterFactory`). È stato utilizzato il pattern `Template Method` nella gerarchia `AbstractAttack`: il metodo `execute()` definisce l'algoritmo comune (aggiornamento e chiamata a `update()`), mentre ogni attacco implementa un comportamento specifico nel metodo `update()`. Inoltre il sistema di level-up delega la logica di potenziamento degli attacchi alle implementazioni concrete della `factory`. Tutto questo permette di aggiungere nuove armi creando una nuova `factory` e la classe attacco senza modificare la logica delle `Weapon`.

Gestione di caricamento dati



Problema Nel gioco è necessario caricare molti tipi diversi di dati da file JSON: personaggi, nemici, armi, attacchi e power-up. Il problema è evitare di scrivere una classe di caricamento separata per ogni tipo di dato, evitare duplicazione di codice e rendere il sistema di caricamento dati facilmente estendibile.

Soluzione Per risolvere questo problema, è stato utilizzato il pattern Singleton per la classe **DataLoader**, che al suo interno contiene un'istanza di **GenericDataLoader** per ogni tipo di dato da caricare. Il sistema utilizza i Generics attraverso la classe **GenericDataLoader<T>**: questa classe può caricare tutti i tipi di dati che implementano l'interfaccia **Identifiable**. L'implementazione usa i generics bounded con `<T extends Identifiable>`, garantendo che tutti gli oggetti caricati abbiano un ID accessibile tramite il metodo `getId()`. All'interno di **GenericDataLoader**, il metodo `load()` implementa il processo di caricamento, mentre il parsing viene effettuato con la libreria Gson. Per ottimizzare le performance, i dati vengono caricati solo quando necessario e memorizzati in una mappa per evitare riletture da file inutili. La gestione degli errori è centralizzata: tutti gli errori vengono notificati al **GameModel** tramite `notifyError()`.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

In questo progetto si è scelto di testare le implementazioni delle principali interfacce del Model attraverso una suite di test automatici basata su `JUnit`. A causa dei vincoli di tempo e della maggiore complessità associata, non è stato invece possibile implementare test automatici per le componenti Controller e View.

A seguire una breve descrizione dei vari test realizzati:

- **TestPositionable**: verifica la corretta gestione delle operazioni spaziali fondamentali, inclusi il posizionamento delle entità nello spazio di gioco e il calcolo delle distanze tra oggetti posizionabili;
- **TestCollidable**: valida l'implementazione del sistema di rilevamento delle collisioni, assicurando che il controllo dei raggi di collisione e la gestione degli eventi di contatto tra entità funzionino correttamente;
- **TestMovable**: testa la logica di movimento delle entità mobili, verificando il corretto aggiornamento delle posizioni in base alla velocità e direzione, nonché la gestione delle traiettorie future per la prevenzione delle collisioni;
- **TestLiving**: controlla l'implementazione delle meccaniche vitali delle entità viventi, inclusa la gestione dei punti salute, i meccanismi di danno e guarigione, e gli stati di attacco delle entità;
- **TestCollectible**: verifica che gli oggetti collezionabili mantengano correttamente il loro valore associato e che il sistema di raccolta modifichi appropriatamente lo stato dell'oggetto una volta collezionato;

- **TestAttack:** valida il ciclo di vita completo degli attacchi, controllando la corretta gestione della durata temporale, l'esecuzione periodica della logica di attacco e la terminazione automatica al raggiungimento del tempo limite;
- **TestScore:** esamina la logica di calcolo e aggiornamento del punteggio, verificando che i diversi eventi di gioco (eliminazioni, livelli raggiunti, tempo di sessione) contribuiscano correttamente al punteggio finale.

3.2 Note di sviluppo

3.2.1 Bassi Ciro

Utilizzo di `lambda expression` e `stream`

Utilizzate frequentemente, principalmente nella creazione e gestione di collezioni di oggetti. Di seguito alcuni esempi significativi:

- **Lambda expression:** in questo esempio ([permalink](#)), utilizzata in `EntityManager` per iterare rapidamente sulla lista di nemici e aggiornare il comportamento di ciascuno di essi;
- **Stream:** in questo esempio ([permalink](#)), utilizzato in `DataBuilder` (`Controller`) per trasformare la lista di nemici ottenuta dal `Model`, in una lista di oggetti `LivingEntityData`, adatti ad essere utilizzati dalla `View`.

Utilizzo di `Optional`

Utilizzato, in questo esempio ([permalink](#)), durante la creazione di un nemico all'interno della classe `EnemyFactory`. Durante la creazione, il metodo `createEnemy()` tenta di assegnare una posizione libera al nemico; tuttavia, per quanto improbabile, può accadere che non venga trovata alcuna posizione valida nemmeno dopo un massimo di 10 tentativi. In tal caso viene restituito un `Optional` vuoto, indicando il fallimento nella creazione del nemico.

3.2.2 Utilizzo di `GridBagLayout`

Il layout `GridBagLayout`, utilizzato all'interno di `AbstractBasePanel`, in particolare per il metodo `addComponent()`, è stato sviluppato riadattando codice proveniente da risorse online.

3.2.3 Foschini Paolo

Utilizzo di `lambda expression` e `stream`

Di seguito alcuni esempi:

- **Lambda expression:** in questo esempio ([permalink](#)) viene usata per ordinare i punteggi in modo rapido e leggibile;
- **Stream:** in questo esempio ([permalink](#)) viene utilizzato per ottenere la lista dei nomi dei file di salvataggio presenti nella cartella.

3.2.4 Testa Alessandro

Utilizzo di lambda expression e stream

- **Lambda expression:** ([permalink](#)) lambda che ritorna `UnlockableCharacter` dato l'id.
- **Stream:** ([permalink](#)) Lo stream filtra le `Weapon` del personaggio e ritorna quella con l'id specificato (se è presente).

Utilizzo di Generics Bounded

Utilizzati nella classe `GenericDataLoader` ([permalink](#)).

Utilizzo di Optional

Usato per la lettura di dati da file, nel caso in cui non esista alcun dato con l'id specificato ([permalink](#)).

Utilizzo della libreria Gson

Usata per la lettura e il parsing di dati da file JSON ([permalink](#)).

Utilizzo di Key Bindings in Swing

Per l'implementazione della gestione degli input da tastiera nella classe `InputHandler` è stata utilizzata la funzionalità `Key Bindings` di Swing. Per l'implementazione è stata usata questa fonte: <https://docs.oracle.com/javase/tutorial/uiswing/misc/keybinding.html> Precedenti implementazioni non funzionavano su macOS, per questo è stata usata `Key Bindings`.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Bassi Ciro

Ritengo che il lavoro sia complessivamente accettabile: il gioco rispetta i requisiti stabiliti in fase iniziale e offre un'esperienza tutto sommato simile a quella del videogioco originale ("Vampire Survivors"). Se però dovessi ricominciare dall'inizio, rivedrei probabilmente alcune scelte progettuali. In particolare, alcune funzionalità potevano essere distribuite meglio tra le classi, evitando classi con ruoli eccessivamente diversi. Ad esempio, l'implementazione iniziale di `GameViewImpl` accorpava funzionalità troppo diverse, rendendo il codice molto lungo e poco manutenibile; la successiva scelta di suddividere in diversi componenti si è rivelata essere molto più efficace. Iniziando da subito con questo approccio, avremmo sicuramente evitato di perdere tempo in refactoring. Mi sono accorto, inoltre, dell'utilità di rispettare i principi OOP, necessari per massimizzare la flessibilità del codice. Ho anche apprezzato l'utilizzo di alcuni pattern, in particolare il Template Method. Un'altra scelta progettuale, condivisa con gli altri membri del gruppo, che ritengo interessante è stata quella di utilizzare file JSON per memorizzare i dati di configurazione e le informazioni relative alle diverse entità. Questo ha reso estremamente semplice e veloce l'aggiunta di nuove entità o la modifica di quelle esistenti, senza dover intervenire sul codice, migliorando così la scalabilità del progetto. Concludendo, credo che questo progetto, pur non privo di margini di miglioramento, rappresenti una base sufficientemente ben progettata e facilmente estensibile per eventuali futuri aggiornamenti.

4.1.2 Foschini Paolo

Non sono pienamente soddisfatto del mio contributo a questo progetto. In alcune occasioni credo di non aver scritto il codice nel modo più corretto, e questo potrebbe aver complicato il lavoro dei miei compagni. Ritengo che, almeno inizialmente, non ho mostrato particolare dedizione al progetto e questo si è riflesso sui risultati. Allo stesso tempo, sono molto contento del gruppo che si è formato e di come siamo riusciti a coordinarci e collaborare in modo efficace. Questo progetto mi ha permesso di imparare molto, sia dal punto di vista del lavoro di squadra che da quello della programmazione ad oggetti.

4.1.3 Testa Alessandro

Il progetto si è concluso con risultati complessivamente positivi. L'architettura finale presenta una struttura che rispetta in larga parte i principi SOLID, garantendo al sistema un buon livello di estendibilità. Tuttavia, la frequenza elevata dei refactoring ha comportato alcuni compromessi sulla qualità del codice, portando occasionalmente a soluzioni non completamente ottimali; per questo, a posteriori, penso sarebbe stato opportuno dedicare maggiore attenzione alla fase di analisi e progettazione iniziale. Nonostante questi aspetti, il risultato complessivo è soddisfacente e in linea con le mie aspettative.

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Bassi Ciro

Una delle principali difficoltà è stata la fase di progettazione iniziale. Non è stato semplice prevedere fin da subito tutte le funzionalità e i dettagli architetturali: molti problemi imprevisti sono emersi durante l'implementazione, rendendo necessarie numerose modifiche alle scelte progettuali prese in precedenza e, in alcuni casi, hanno portato a soluzioni non ottimali o a imprecisioni nell'organizzazione del codice. Allo stesso tempo, ho trovato molto stimolante il lavoro di gruppo. Il confronto con gli altri membri ha portato idee e soluzioni a cui probabilmente non avrei mai pensato da solo.

4.2.2 Foschini Paolo

All'inizio ho riscontrato alcune difficoltà nel mantenere costanza nello sviluppo del progetto. Credo che questo sia dovuto al fatto che, nelle fasi iniziali, è difficile percepire concretamente i risultati del proprio lavoro: ci si concentra soprattutto sulla progettazione e sull'implementazione delle prime classi, senza vedere ancora un prodotto visibile o funzionante. Con il progredire del progetto, però, la situazione è cambiata: man mano che prendeva forma, è cresciuta anche la mia motivazione e il desiderio di portarlo avanti al meglio.

Un'ulteriore difficoltà è stata conciliare lo sviluppo del progetto con la frequenza delle lezioni del nuovo semestre, il che ha richiesto un notevole impegno in termini di organizzazione e gestione del tempo. Devo riconoscere che lavorare in gruppo è stato un elemento fondamentale: avere qualcuno con cui confrontarsi, che incoraggia e stimola a continuare, si è rivelato un grande aiuto per completare il progetto al meglio.

Appendice A

Guida utente

A.1 Selezione del salvataggio

Nella schermata iniziale A.1 l'utente può scegliere se iniziare una nuova partita o caricare un salvataggio esistente. Premendo il pulsante **NEW GAME** viene creato un nuovo salvataggio, mentre il pulsante **LOAD GAME** consente di selezionare e caricare un salvataggio già esistente A.2.

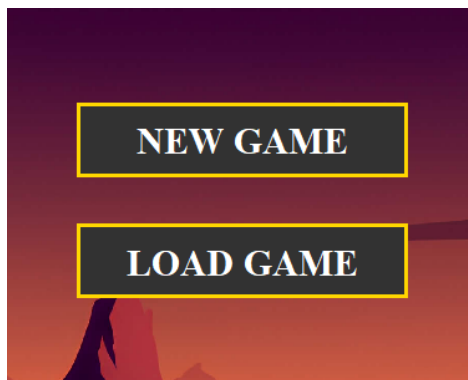


Figura A.1: Schermata iniziale

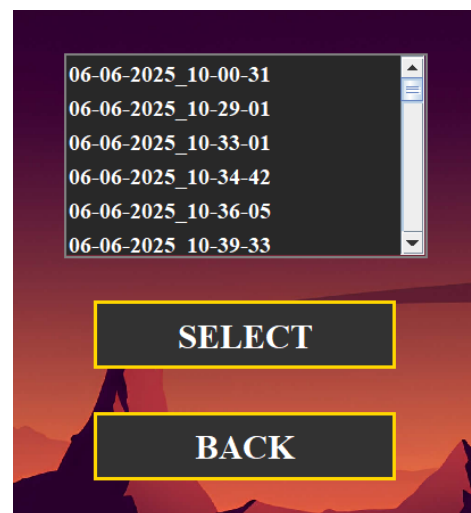


Figura A.2: Selezione salvataggio

A.2 Menu principale

Dopo aver premuto **NEW GAME**, si arriva al menù principale A.3.

- **SCOREBOARD**: consente di visualizzare i punteggi ottenuti nelle partite associate al salvataggio attualmente caricato;
- **SHOP**: permette di accedere al negozio del gioco, suddiviso in due sezioni: Characters, dove è possibile acquistare nuovi personaggi giocabili, e Power Ups, dedicata all'acquisto di potenziamenti;
- **LOAD SAVE**: torna alla schermata di selezione dei salvataggi.

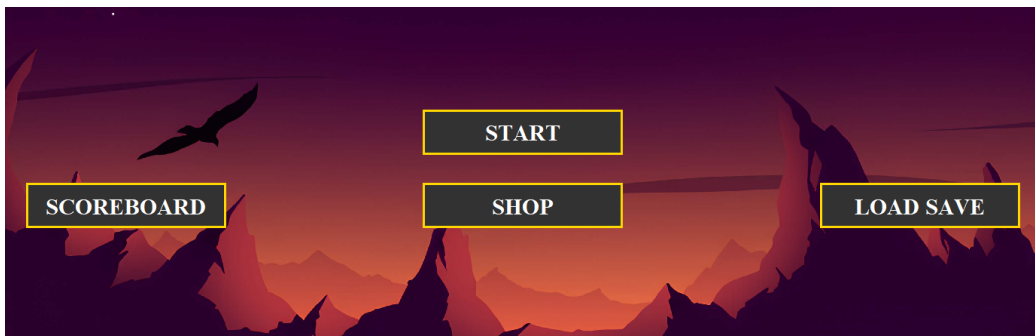
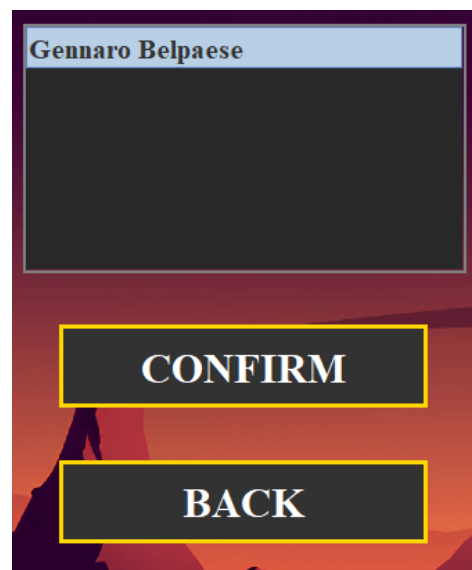
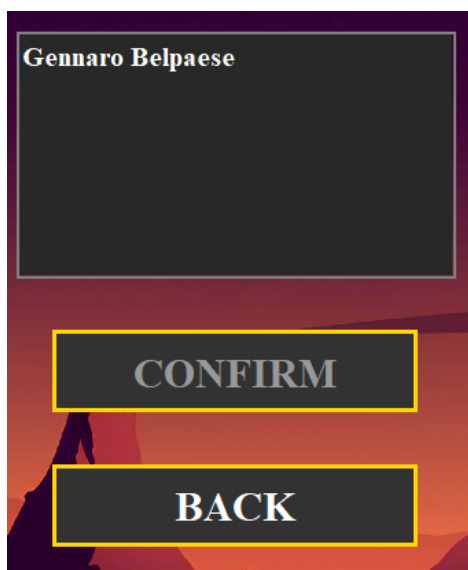


Figura A.3: Menù principale





A.3 Selezione del personaggio

Nella schermata iniziale l'utente può scegliere il personaggio da utilizzare durante la partita, tra quelli sbloccati. Una volta scelto, si può cominciare la partita con il tasto **CONFIRM** o tornare al menu principale con il tasto **BACK**.



A.4 Partita

Durante la partita, il personaggio può essere controllato utilizzando le frecce direzionali oppure i tasti W, A, S e D. È possibile equipaggiare fino a un massimo di tre armi, inclusa quella iniziale assegnata al personaggio scelto. Le armi disponibili appartengono a quattro categorie principali:

-  **Knife**, arma di partenza di Gennaro Belpaese, spara nella direzione in cui il personaggio sta guardando. Potenziando quest'arma diminuirà il tempo tra il lancio di un coltello e quello successivo;
-  **Garlic**, arma di partenza di Poe'Ratcho, è una zona circolare attorno al personaggio. Potenziando quest'arma aumenterà l'area di danno;
-  **Santa Water**, arma di partenza di Suor Clerici, è una piccola zona circolare che spunterà in un punto casuale della mappa. Potenziando quest'arma aumenterà il danno;
-  **Magic Wand**, arma di partenza di Imelda Belpaese, spara sempre un colpo al nemico più vicino al personaggio. Potenziando quest'arma diminuirà il tempo tra un colpo e il successivo.

Ogni arma ha le sue caratteristiche uniche, tra cui il cooldown (il tempo di attesa tra un attacco e l'altro), il danno e la dimensione degli attacchi. Per ottenere nuove armi o potenziare quelle già acquisite, il giocatore deve salire di livello raccogliendo i punti esperienza lasciati dai nemici sconfitti (vedi sezione successiva). Al raggiungimento di un nuovo livello, viene mostrata una schermata che permette di scegliere un'arma da sbloccare o potenziare, come in figura A.4. Dopo aver effettuato la selezione, è sufficiente premere il tasto **CHOOSE** per tornare alla partita.

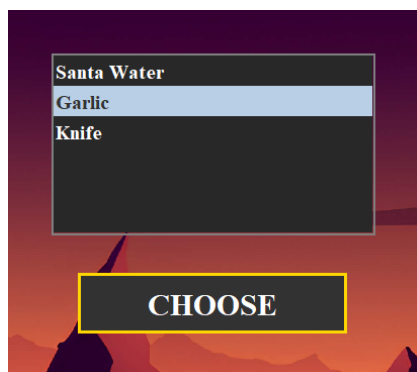


Figura A.4: Selezione nuova arma

Quando un nemico viene sconfitto, esiste una probabilità che rilasci un oggetto collezionabile. Questi possono essere di tre tipologie:




-  **Coin:** necessario per poter acquistare potenziamenti e nuovi personaggi nel negozio;
-  **Experience gem:** necessaria per far progredire il personaggio di livello, permettendogli di utilizzare nuove armi o potenziare quelle già in suo possesso.
-  **Food:** cura una parte della vita del personaggio, mostrata dalla barra rossa sotto al personaggio, come in figura A.5.



Figura A.5: Schermata partita

Premendo il tasto **ESC** durante la partita, il gioco verrà messo in pausa e verrà mostrata una schermata A.6 con due opzioni: **RESUME**, per riprendere la partita da dove si era interrotta, e **EXIT**, per tornare al menu principale.

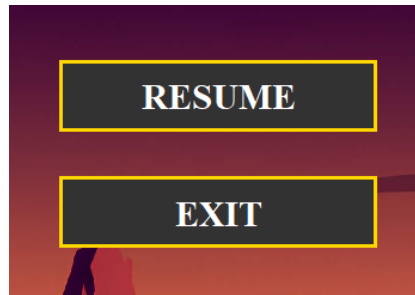


Figura A.6: Menù di pausa

Al termine della partita, viene visualizzata una schermata A.7 di riepilogo con le statistiche e il punteggio ottenuto. Selezionando il tasto **CONTINUE**, si ritorna al menu principale.

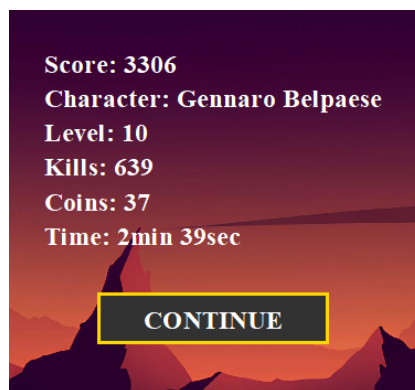


Figura A.7: Menù di fine partita

A.5 Shop

Nella sezione Shop, divisa tra **CHARACTERS** e **POWERUPS**, premendo sull'oggetto di interesse verranno mostrate a schermo le informazioni relative ad esso, tra cui immagine, descrizione e prezzo, come in figura A.8. Il pulsante **BUY** rimarrà disattivato fino a quando non si avranno le monete necessarie per comprare l'oggetto selezionato.



Figura A.8: Esempio di informazioni relative all'item

A.6 Scoreboard

Il pannello Scoreboard mostra l'elenco delle partite concluse relative al salvataggio corrente, come in figura A.9. Per ciascuna partita vengono visualizzate informazioni come il punteggio finale, la durata, il personaggio utilizzato e il livello massimo raggiunto. Le partite sono ordinate in ordine decrescente di punteggio.

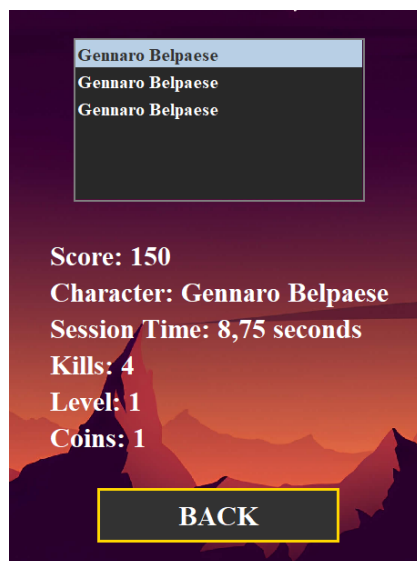


Figura A.9: Scoreboard

A.7 Salvataggi

I salvataggi vengono memorizzati nella cartella `HOME` dell'utente. Per trasferirli su un altro computer è sufficiente copiare la cartella `vampire-io_save` A.10 nella cartella `HOME` del nuovo computer.

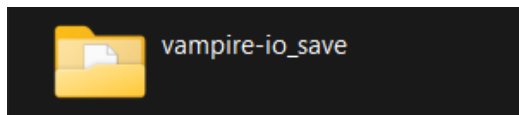


Figura A.10: Cartella dei salvataggi