Relazione Progetto d'esame
Programmazione e Modellazione a Oggetti

Sessione Autunnale 2024/2025

FiveRealms

Saga dei 5 Regni

AUTORI

Mattia Gasperoni
matricola: 329235

Andrea Marchionni
matricola: 326970

Claudio Valentin Gaspar
matricola: 328833

Indice

Capitolo 1

Modifiche

Capitolo 2

Analisi

Il team si propone di sviluppare un gioco basato sul combattimento tra personaggi medievali, arricchito da diverse varianti.
Il titolo dell'applicativo, FiveRealms, nasce dal fatto che il gioco è articolato in cinque livelli principali, corrispondenti a cinque regni differenti.
FiveRealms è un gioco di strategia in cui il giocatore deve guidare i propri personaggi in combattimenti contro nemici che, a ogni round, diventano progressivamente più forti.
Il gioco si fonda su:

- un sistema di personaggi suddivisi in alleati e nemici;
- un sistema di combattimento a turni, basato su griglia e con ordine determinato dalla velocità;
- un sistema di equipaggiamento con armi specializzate e pozioni;
- una struttura a livelli, composta da un tutorial e dai cinque regni principali.

Il flusso del gioco prevede che il giocatore scelga tre personaggi alleati tra quelli disponibili, con cui affrontare una serie di livelli a difficoltà crescente.
Durante ogni livello, i personaggi esplorano la mappa e si scontrano con i nemici.
La progressione avviene attraverso l'accumulo di esperienza, che consente di migliorare le statistiche dei personaggi. Inoltre, l'equipaggiamento ottenuto in battaglia garantisce vantaggi strategici.
Infine, il gioco integra un sistema di salvataggio e caricamento che permette all'utente di interrompere e riprendere la partita in qualsiasi momento.

2.1 Requisiti

2.1.1 Requisiti funzionali

- Il gioco consente di selezionare 3 alleati tra i 5 disponibili da utilizzare in battaglia.
- Sistema di combattimento a turni dinamico, basato sulla velocità individuale dei personaggi.
- Sistema di movimento user-friendly con evidenziazione delle posizioni disponibili sulla mappa.
- Sistema di attacco intuitivo con visualizzazione delle aree di attacco disponibili.
- Evidenziazione in verde del personaggio a cui spetta il turno.
- Sistema di equipaggiamento dinamico con armi di diverse tipologie e pozioni.
- Generazione dei nemici in posizioni semi-casuali con equipaggiamento pseudo-casuale.
- Sistema di esperienza e progressione che migliora le statistiche dei personaggi.
- Sistema di salvataggio e caricamento delle partite.
- Tutorial interattivo opzionale per introdurre le meccaniche di gioco.
- Sistema musicale di sottofondo per rendere l'esperienza più immersiva e coinvolgente.

2.1.2 Requisiti non funzionali

Oltre alle funzionalità principali, l'applicazione dovrà garantire:

- Un'interfaccia chiara e intuitiva, adatta sia a principianti che a giocatori esperti;
- Caricamenti rapidi e una gestione fluida delle animazioni e degli scontri;
- Salvataggi sicuri e integri anche in caso di interruzioni improvvise;
- Possibilità di estendere facilmente il gioco con nuovi livelli, personaggi o modalità;
- Feedback chiaro e immediato sugli eventi di gioco;
- Un'esperienza bilanciata, in cui l'intelligenza artificiale non sia né troppo prevedibile né imbattibile;
- Funzionamento in locale senza necessità di connessione a Internet.

2.1.3 Diagramma dei Casi d'Uso

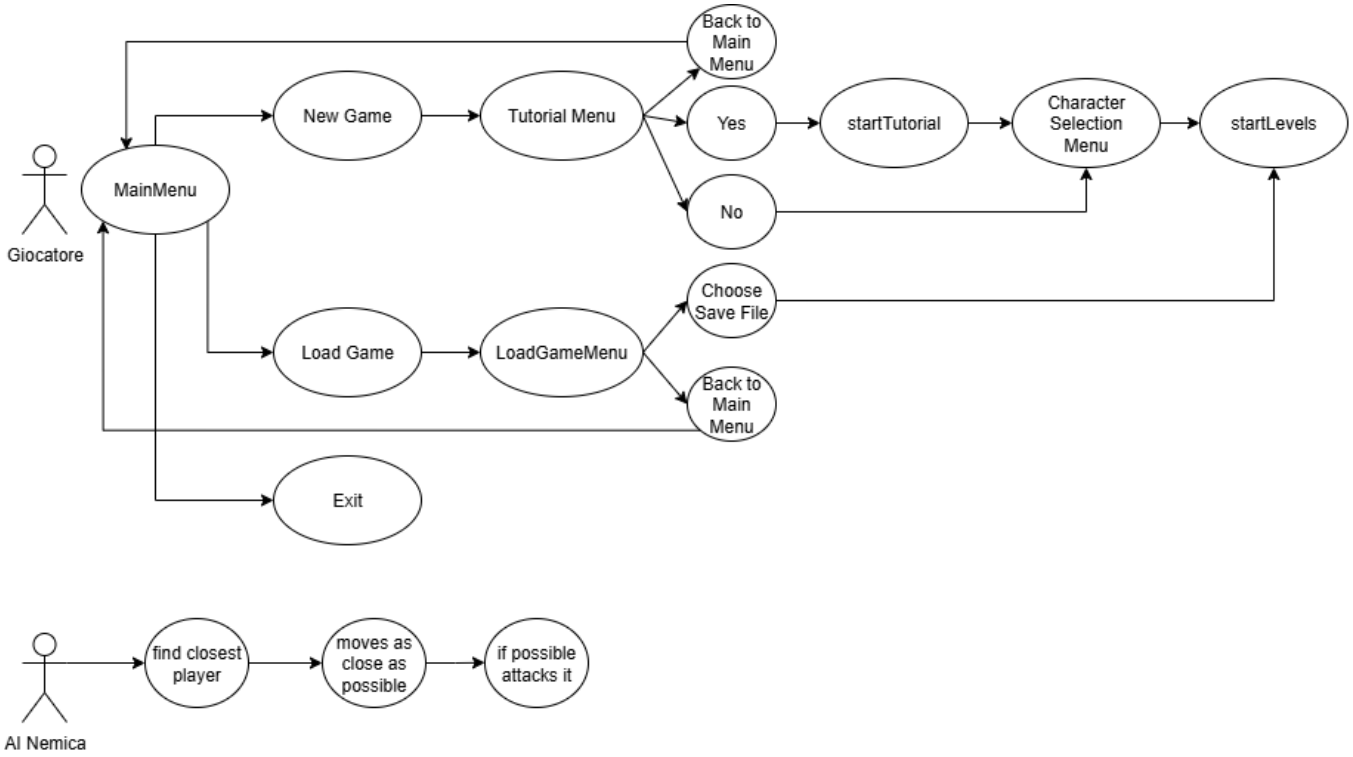

Figura 2.1: Diagramma dei Casi d'Uso

- **Attori coinvolti**
  - **Giocatore** – L'utente che interagisce con il sistema di gioco, navigando nei menu e prendendo decisioni di gioco.
  - **IA Nemica** – L'intelligenza artificiale che gestisce il comportamento dei personaggi nemici.
- **Casi d'uso principali**
  - **Per il Giocatore**
    - Avvia nuova partita – Dal Menu Principale seleziona *New Game*
    - Carica partita – Dal Menu Principale seleziona *Load Game*, accedendo alla schermata *LoadGameMenu* per scegliere un salvataggio.
    - Seguire o saltare il tutorial, decide se avviare la modalità *StartTutorial* o passare al *Character Selection Menu*.
    - Seleziona personaggi – Sceglie i personaggi prima di iniziare il livello.
    - Terminare il gioco – Dal Menu Principale può selezionare *Exit* per chiudere l'applicazione.
  - **Per l'IA Nemica**
    - Individua il giocatore più vicino – Analizza la mappa per determinare la posizione del bersaglio più vicino.
    - Si muove verso il giocatore – Sposta la propria posizione riducendo la distanza.
    - Attacca il giocatore, se possibile – Se a distanza utile, esegue un attacco.

- **Relazioni tra attori e casi d'uso**
  - Il **Giocatore** interagisce con l'intero sistema dei menu e prende decisioni strategiche durante il gioco.
  - L'**IA Nemica** agisce in autonomia, rispondendo alla posizione del giocatore con movimenti e attacchi, secondo una logica predefinita.

2.2 Analisi e Modello del dominio

Il sistema è progettato per gestire un gioco strutturato a livelli, in cui il giocatore affronta sfide dinamiche interagendo con personaggi, mappe e meccaniche di combattimento.
L'architettura mira a favorire la modularità e la futura estendibilità del gioco, facilitando l'aggiunta di nuovi livelli, personaggi o modalità.
L'entità centrale del sistema è rappresentata dalla classe **Game**, responsabile della gestione del flusso generale dell'applicazione.
Essa si occupa dell'inizializzazione dei livelli, dell'aggiornamento dello stato del gioco e del coordinamento tra i componenti principali, come il controller, la mappa e l'interfaccia utente.
Ogni livello è modellato dall'entità `Level`, che incapsula i seguenti elementi principali:

- **Personaggi (`Character`):** rappresentano le entità controllabili o nemiche all'interno del gioco. Ogni personaggio è dotato di caratteristiche generate tramite statistiche pseudo-casuali, rendendolo unico per ogni partita.
- **Mappa di gioco (`LevelMap`):** contiene la configurazione spaziale del livello, inclusa la disposizione degli elementi statici e la posizione iniziale dei personaggi.
- **Ciclo di gioco (Game Loop):** gestisce il susseguirsi delle azioni nel tempo, aggiornando lo stato dei personaggi, le interazioni e le dinamiche del gioco in maniera continua.
- **Interfaccia di combattimento (`BattlePhaseView`):** responsabile della visualizzazione delle fasi di battaglia, sincronizza le animazioni con lo stato del gioco e coordina l'interfaccia utente durante le fasi di azione.

La gestione dei movimenti e delle azioni avviene attraverso la **`BattlePhaseView`**, che garantisce la coerenza tra logica di gioco e rappresentazione visiva.
In sintesi, il modello del dominio è composto dalle entità principali **`Game`**, **`Level`**, **`Character`**, **`LevelMap`** e **`BattlePhaseView`**.
Ciascuna entità è responsabile di una specifica parte del comportamento del gioco, contribuendo a definire un'architettura modulare, facilmente manutenibile ed estendibile.

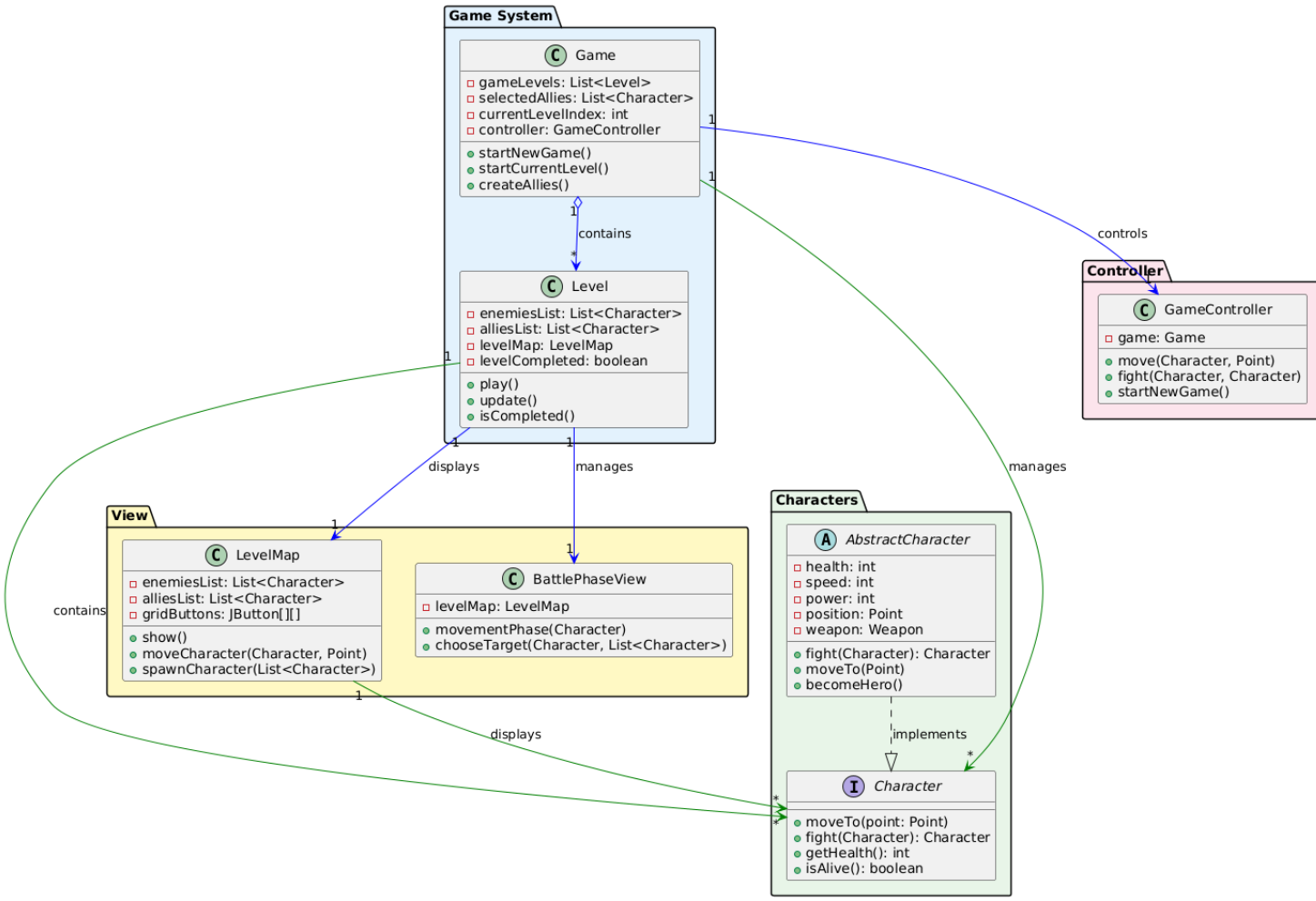

Figura 2.2: Schema UML del dominio

Capitolo 3

Design

3.1 Architettura

Per lo sviluppo è stato adottato il pattern Model-View-Controller.
Il **Model** gestisce lo stato del gioco, offrendo metodi per accedere e modificare i dati ad esempio: **charaters**, **equipment**, **gamestatus** e **point**.
La **View** si occupa invece della rappresentazione grafica delle informazioni contenute nel Model, presentandole all'utente in modo chiaro e intuitivo, tramite l'utilizzo di **mappe** e **menu** iterattivi.
L'interazione dell'utente avviene tramite la View, la loro elaborazione e gestione è affidata al **Controller**, il quale ha anche il compito di aggiornare il Model in base a tali input.

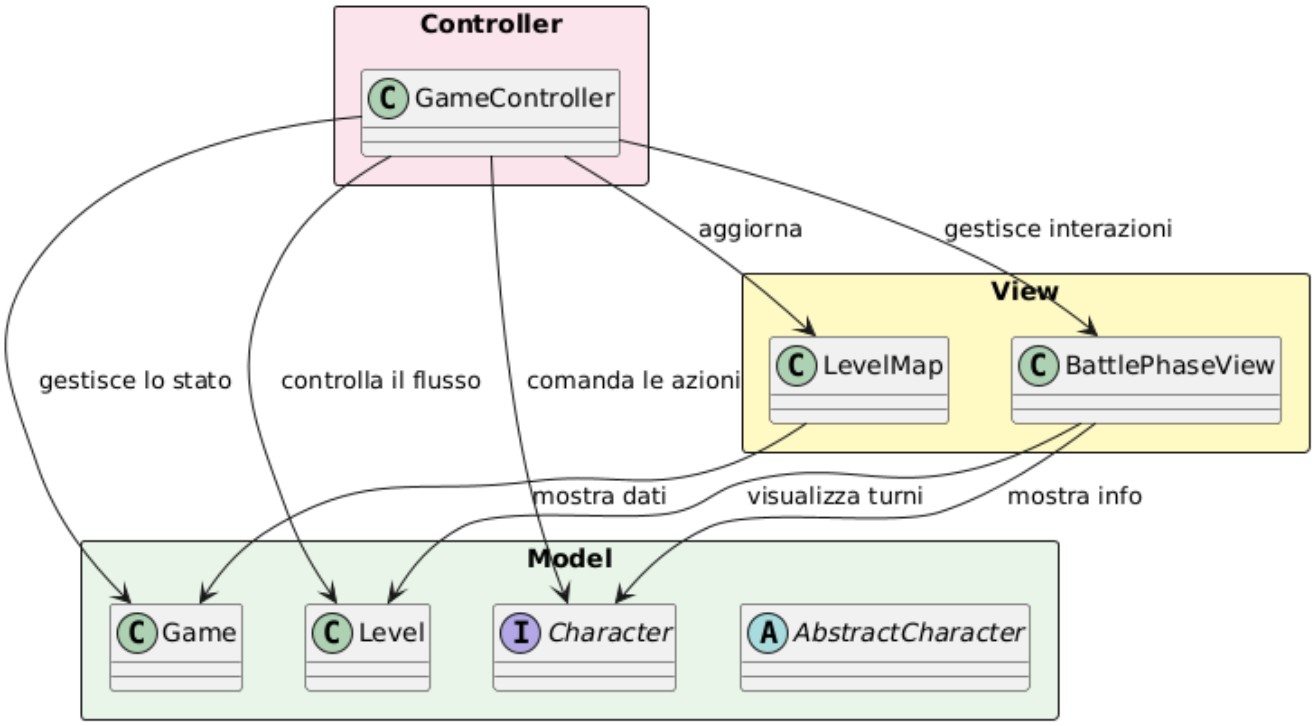

Figura 3.1: MVC design pattern

3.1.1 Model

Il Model si occupa di gestire le metodologie di accesso e modifica dei dati dell'applicazione e del dominio applicativo, determinando coerentemente come le interazioni dell'utente influenzino lo stato corrente dell'applicazione.
Scopo del Model è quindi fornire al Controller un accesso allo stato attuale dell'applicazione.
Per fornire tale funzionalità è stato scelto di utilizzare un'entità Model, responsabile di mantenere lo stato corrente dell'applicazione, che include i principali componenti del sistema: i **personaggi** (come Knight, Wizard, Barbarian), gli elementi di **equipaggiamento** (armi e pozioni), la gestione dello **stato di gioco** (livelli, salvataggi, musica, tutorial).
Immagine uml model, Game.java

3.1.2 View

La View consiste nella schermata dell'applicazione che si occupa della gestione della parte grafica, della User Experience e dell'interazione con l'utente.
È compito delle varie View registrare ed informare il rispettivo Controller di interazioni con l'applicazione da parte dell'utente, in attesa di una sua risposta sul cambiamento dei dati.
È responsabile di rilevare le azioni dell'utente e notificare il Controller affinché le gestisca correttamente.
Si è scelto di progettare View e Controller in modo indipendente dal framework grafico utilizzato, così da rendere semplice l'adozione futura di altre librerie grafiche.
In tal modo, un eventuale cambiamento del framework richiederebbe esclusivamente la riscrittura della View, lasciando inalterati Controller e Model.
La struttura del package `view` è suddivisa in componenti specifici per la visualizzazione delle mappe di gioco (come `LevelMap`, `TutorialMap`, `GridPanel`) e dei menu di navigazione (`MainMenu`, `PauseMenu`, `EndGameMenu`, ecc.), oltre a gestire elementi grafici dinamici come **tooltip** e **banner**.
Questa organizzazione modulare consente una gestione chiara e scalabile dell'interfaccia utente, facilitando l'estensione o la modifica delle visualizzazioni senza impattare sulla logica di controllo o sul modello dati.
Uml della view, Map con AbstractMap; Menu con AbstractMenu, View:

3.1.3 Controller

Il Controller è quella componente cui spetta gestire in maniera conseguenziale le interazioni da parte dell'utente nella View, comunicando quindi al Model il cambiamento avvenuto.
Una volta che il Model avrà completato l'elaborazione della richiesta di cambiamento, il Controller avviserà la propria View, in modo tale che quest'ultima possa aggiornarsi in maniera coerente secondo le regole specificate dal Model.
Come già accennato nella sezione precedente, si è scelto di evitare l'utilizzo di un singolo controller, sia per i motivi sopracitati, sia perché un unico controller sarebbe risultato avere un eccessivo grado di responsabilità, usando infatti dentro al package **controller** 4 tipo di controller:

- **GameController**: gestisce la logica del gameplay, orchestrando le interazioni tra personaggi, livelli e regole di gioco.
- **MapController**: si occupa della gestione della mappa di gioco
- **MenuController**: controlla i menu interattivi, come il menu principale, di pausa e di fine partita, garantendo una navigazione fluida.
- **MusicController**: gestisce la riproduzione e la transizione delle tracce audio durante le diverse fasi del gioco.
- **SaveController**: coordina le operazioni di salvataggio e caricamento dello stato di gioco.

UML dei controller:

3.2 Design dettagliato

3.2.1 Gasperoni

Durante il progetto, mi sono occupato principalmente dello sviluppo della logica di gioco, del coordinamento tra il gioco e la sua interfaccia grafica, e della gestione dei salvataggi. In particolare, ho implementato e gestito le classi principali del progetto, tra cui Game, Level, MusicManager, oltre al controller e alle sue specializzazioni, curando la comunicazione tra logica di gioco e la view. Ho inoltre sviluppato il sistema di salvataggio e caricamento delle partite tramite le classi GameSave e GameSaveManager, in modo da poter interrompere la partita e riprenderla senza perdita di progressi.

Game

La classe **Game** rappresenta il nucleo della logica del gioco e funge da coordinatore tra i vari *manager* specializzati. Invece di inglobare direttamente tutta la logica, delega responsabilità specifiche a componenti dedicate come **CharacterManager**, **LevelManager**, **GameLoopManager**, **TutorialManager**, **GameSaveManager** e **GameStateManager**. Questo approccio aderisce al principio di *Single Responsibility*, migliorando la coesione interna e rendendo il sistema più manutenibile ed estendibile.

L'oggetto **Game** agisce quindi da facciata, incapsulando la complessità delle interazioni tra i sottosistemi e offrendo un'interfaccia semplificata per l'avvio delle varie modalità di gioco (tutorial, nuova partita, caricamento, ecc.).

Per la gestione della view e dell'interfaccia utente, **Game** collabora con il **GameController**, mentre la logica di salvataggio e caricamento è affidata a **GameSaveManager**.

Tra le funzionalità principali troviamo:

- **Coordinamento dei manager:** ogni responsabilità (livelli, personaggi, ciclo di gioco, stato, tutorial, salvataggi) è delegata a un componente specifico;
- **Gestione dei livelli:** ogni livello è modellato come oggetto **GameLevel**, consentendo riuso e scalabilità;
- **Gestione dei personaggi:** selezione, reinizializzazione e sostituzione gestite tramite **CharacterManager**, mantenendo separata la logica del party;
- **Ciclo di gioco non bloccante:** grazie al **GameLoopManager**, che sfrutta `ScheduledExecutorService`, la logica viene aggiornata periodicamente senza bloccare la GUI;
- **Tutorial e scenari guidati:** avviabili attraverso **TutorialManager**, per facilitare l'apprendimento delle dinamiche di gioco;
- **Persistenza dello stato:** caricamento e salvataggio delegati a **GameSaveManager**, garantendo coerenza e riuso del codice di persistenza.

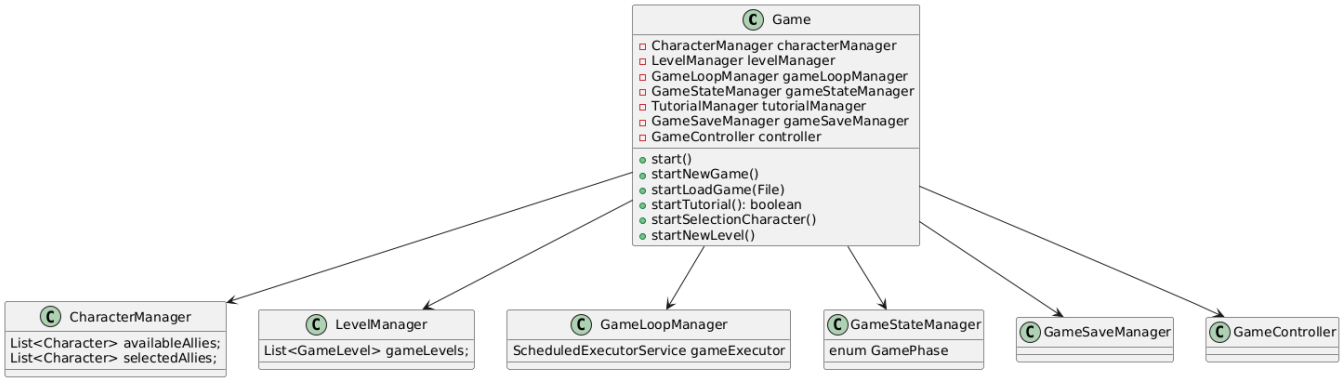

Figura 3.2: UML di Game

GameLevel e GameTutorial

Le classi **GameLevel** e **GameTutorial** implementano l'interfaccia **Level**, che definisce: `boolean play()` e `boolean isCompleted()`.

GameTutorial è un'implementazione semplice di *Level*, dedicata a mostrare messaggi pop-up per guidare l'utente all'interno del gioco, senza logiche complesse di battaglia. Questo permette di separare le funzionalità didattiche dal resto del gioco, seguendo il principio di *Single Responsibility*.

GameLevel rappresenta invece un livello completo, con gestione delle fasi di battaglia, ordine dei turni, condizioni di vittoria/sconfitta, turni dei personaggi giocatori e IA, e interazione con la mappa e la UI tramite **BattlePhaseView**. La classe incapsula lo stato del livello, le liste di alleati e nemici, l'ordine dei turni e la fase corrente.
Questa struttura segue i principi di *incapsulamento*, *modularità* e *riuso*, permettendo di aggiungere nuove meccaniche o scenari senza modificare la logica centrale del gioco. Inoltre, l'uso di callback e gestione asincrona dei turni rispetta il principio di *separazione dei compiti*, isolando la logica di gioco dalla UI.

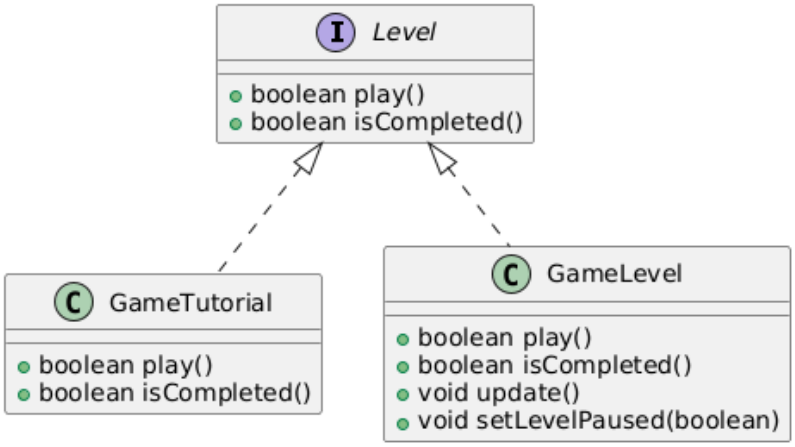

Figura 3.3: UML di Level e delle sue Implementazioni

MusicManager

La classe **MusicManager** si occupa della gestione della musica all'interno del gioco. Essa incapsula le funzionalità di riproduzione e interruzione dei brani audio, centralizzando il controllo delle tracce in un'unica classe, in linea con il principio di *Single Responsibility*.

Internamente, **MusicManager** mantiene una mappa associativa (`Map<String,String>`) che collega i nomi logici dei brani (es. `"level1"`, `"tutorial"`, `"win"`) ai percorsi dei file audio corrispondenti. La riproduzione è gestita tramite la classe `Clip` di `javax.sound.sampled`, consentendo sia l'avvio di tracce singole sia la ripetizione continua in loop.
Tra i metodi principali troviamo:

- **play(trackName, loop):** interrompe eventuali tracce in riproduzione e avvia il brano richiesto, con la possibilità di loop continuo.
- **stop():** ferma la riproduzione corrente, garantendo che non vi siano sovrapposizioni tra tracce diverse.

Questo design favorisce il riuso e l'estendibilità: per aggiungere nuove tracce o modificare la gestione audio non è necessario intervenire sulla logica di gioco, rispettando i principi di *modularità* e *incapsulamento*.

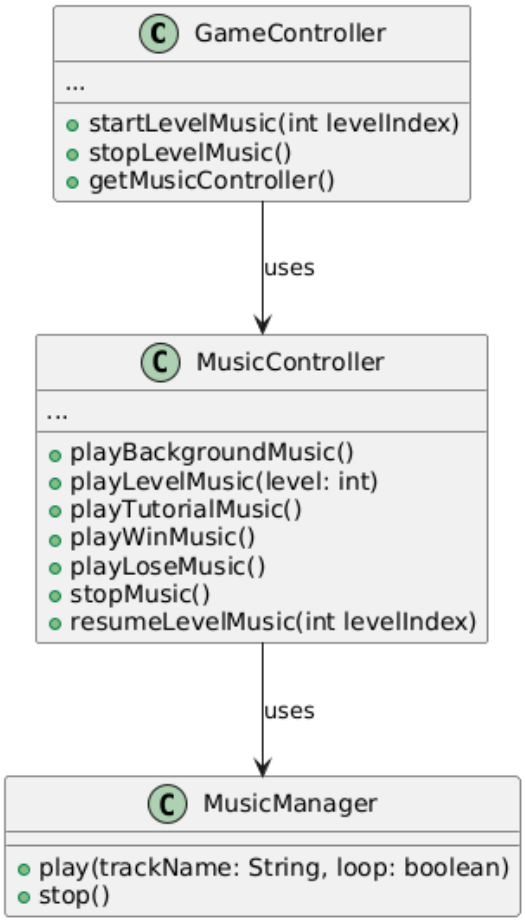

Figura 3.4: UML di MusicManager

Controller

La classe **GameController** rappresenta il controller principale del progetto e incarna pienamente il principio di *delegazione delle responsabilità* e di *separazione dei compiti*. Essa funge da intermediario tra il modello di gioco (la classe `Game`) e i controller specializzati che gestiscono aspetti specifici del gioco, tra cui:

- **MusicController:** gestione della musica adattando la traccia in base al livello corrente o agli esiti della partita;
- **MapController:** gestione delle operazioni sulla mappa, come movimento, combattimenti o rimozioni;
- **SaveController:** gestione del salvataggio e caricamento delle partite;
- **MenuController:** gestione dei menu (principale, di pausa, di fine partita, ecc) e dell'interfaccia utente.

Questa architettura facilita la manutenzione e l'estendibilità del codice, rispettando il *Single Responsibility Principle*: ogni controller ha un compito ben definito, mentre `GameController` coordina la comunicazione tra di essi e fornisce un'interfaccia unificata al resto del sistema.

L'uso dei metodi getter per l'accesso diretto ai controller specializzati permette ulteriori estensioni e favorisce il *principio di apertura/chiusura*, rendendo `GameController` facilmente integrabile in altri componenti dell'applicazione.

Figura 3.5: Enter Caption

Sistema di Salvataggio del Gioco

Il sistema di salvataggio del gioco è composto da due classi principali: `GameSave` e `GameSaveManager`. Queste classi gestiscono lo stato del gioco e permettono di salvarlo, caricarlo e mantenerne traccia in maniera persistente.

GameSave

La classe `GameSave` rappresenta lo stato del gioco in un determinato momento. Contiene le informazioni fondamentali necessarie per riprendere una partita, nello specifico:

- Il livello corrente del gioco (`levelIndex`).
- La lista dei personaggi alleati (`allies`).
- La lista dei personaggi nemici (`enemies`).

Tutte le liste dei personaggi sono immutabili e copiate al momento del recupero tramite i metodi `getAllies()` e `getEnemies()`, garantendo l'integrità del salvataggio.

GameSaveManager

La classe `GameSaveManager` è responsabile della gestione dei salvataggi, fornisce funzionalità per:

- Salvare la partita in un file, con un nome generato automaticamente.
- Caricare una partita da un file specifico oppure l'ultimo salvataggio disponibile.
- Controllare l'esistenza di salvataggi e ottenere la lista dei file salvati, ordinati per data di modifica.
- Eliminare salvataggi specifici.
- Accedere in maniera controllata ai dati del salvataggio attualmente caricato, come livello, alleati e nemici.

In sostanza, `GameSaveManager` funge da intermediario tra il gioco e il sistema di file, mantenendo in memoria lo stato corrente del salvataggio e garantendo operazioni sicure di lettura e scrittura.

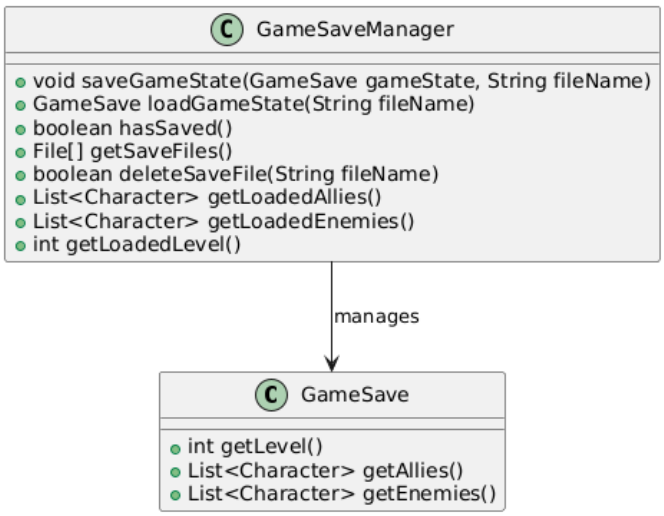

Figura 3.6: UML del saveSystem

3.2.2 Marchionni

Equipment

- **Potions**: Il package **potions** rappresenta tutte quelle pozioni che un personaggio può utilizzare durante la partita e possono essere di 4 tipi:
  - **PotionDefence**
  - **PotionHealth**
  - **PotionPower**
  - **PotionSpeed**

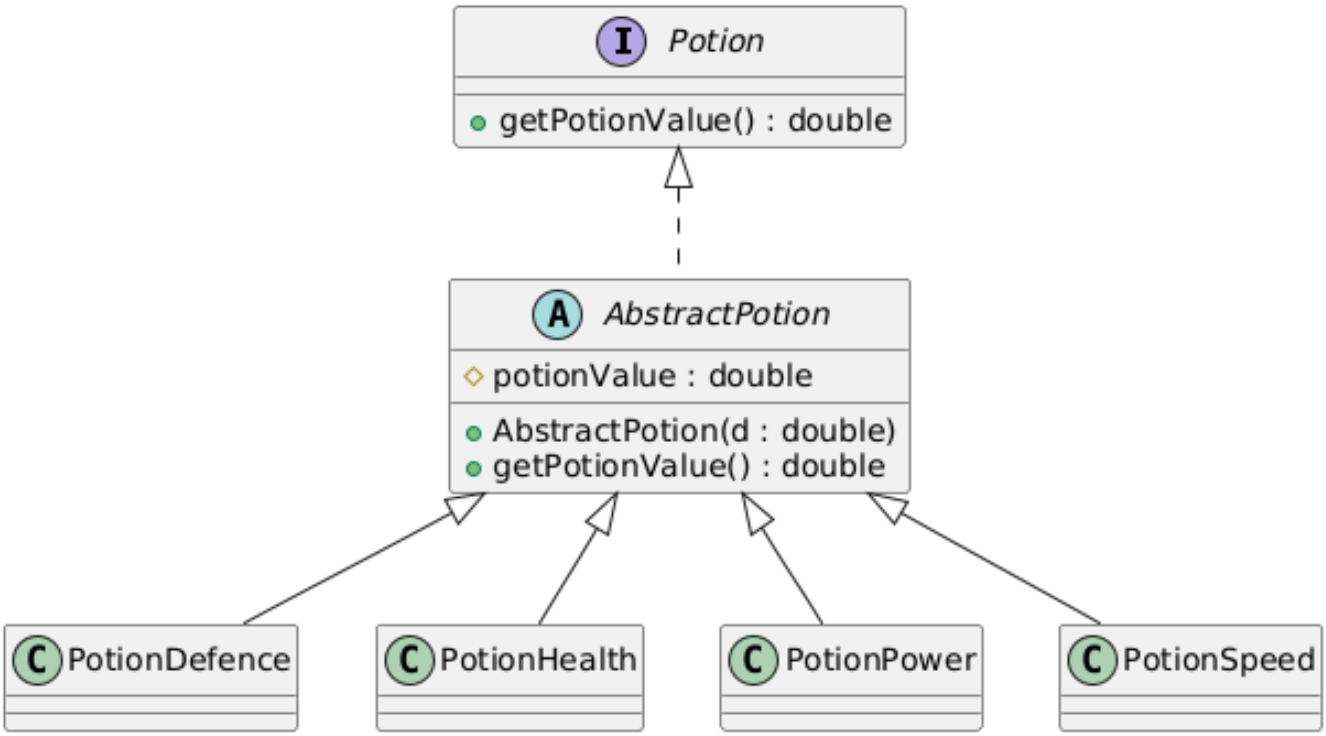

Figura 3.7: UML del package Potions

- **Weapons**: La classe `Weapon` modella le armi disponibili, suddivise in leggere, pesanti, archi e bacchette. Il package **weapons** rappresenta le armi disponibili equipaggiabili da parte dei personaggi e possono essere di vario tipo:
  - **Axe**
  - **LongBow**
  - **LongSword**
  - **ShortBox**
  - **ShortSword**
  - **Spear**
  - **Staff**
  - **Wand**

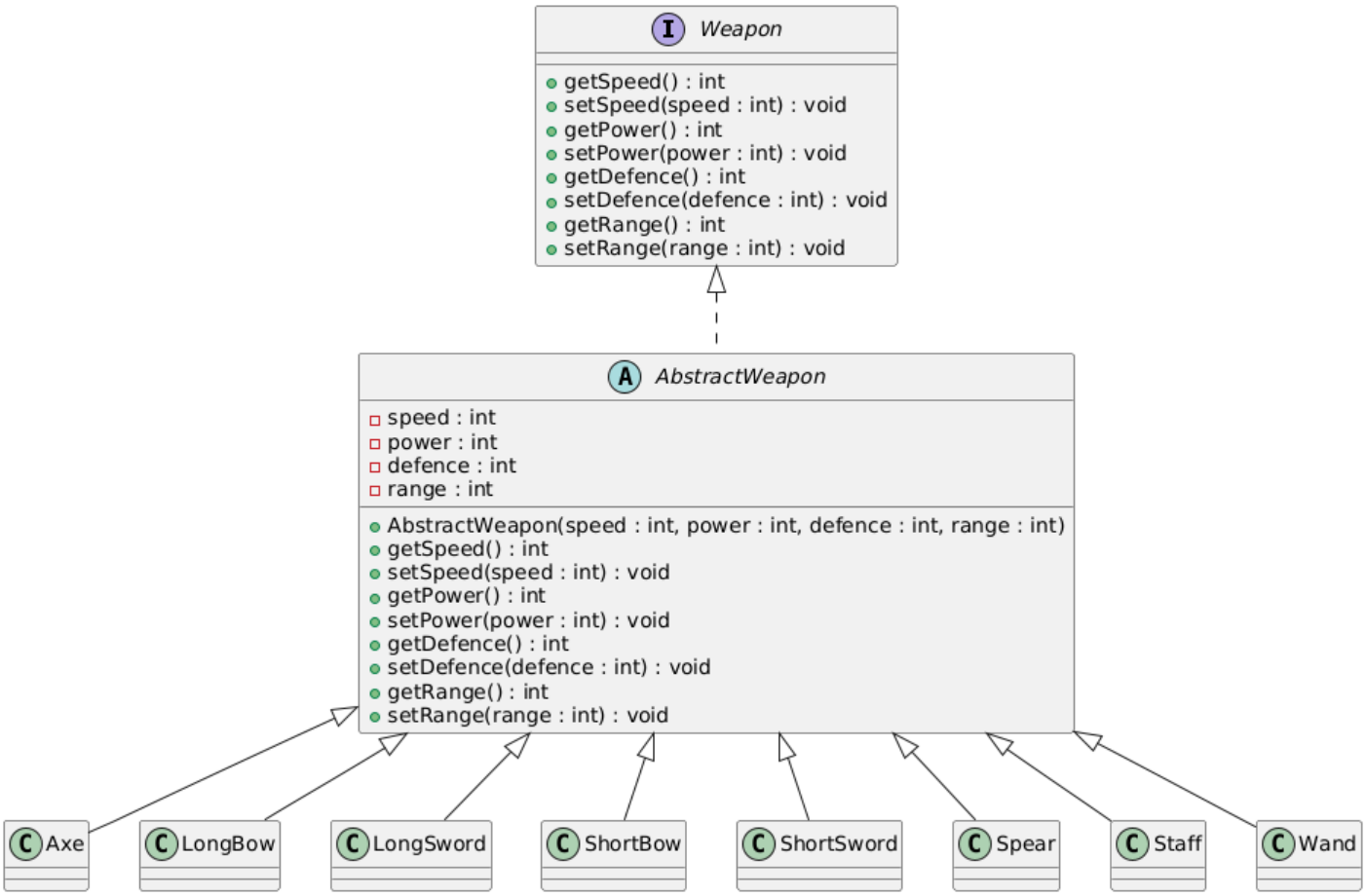

Figura 3.8: UML package Weapons

View

- **Map**: La classe `Map` gestisce la griglia e i movimenti. Il pattern *Template Method* è utilizzato in `AbstractMap`, che definisce i layer grafici (background, griglia, banner informativo e tooltip).

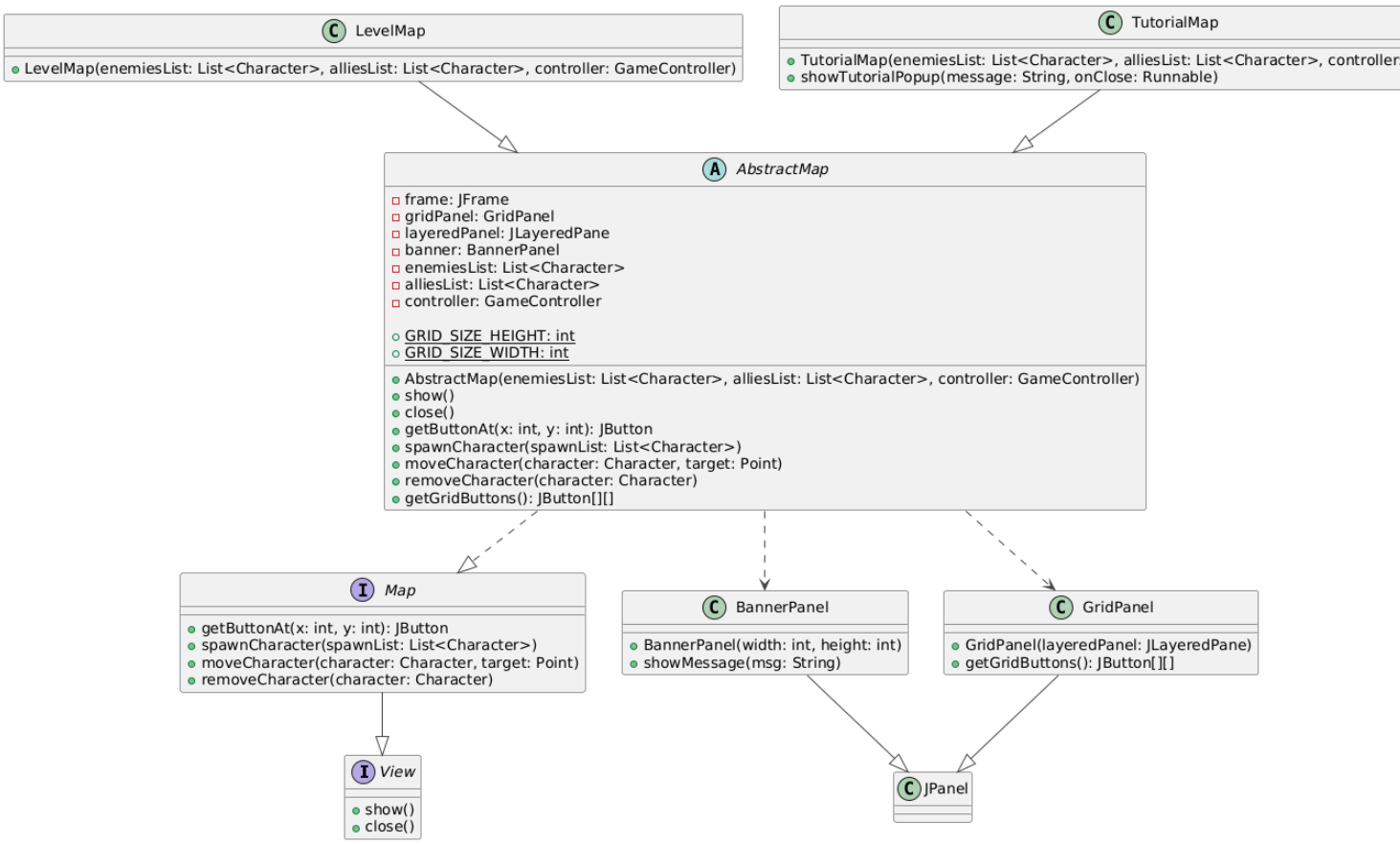

Figura 3.9: UML package Map

- **Menu**: Il package menu rappresenta tutti i menu interattivi user-friendly esposti a schermo per rendere più fluido l'esecuzione del gioco

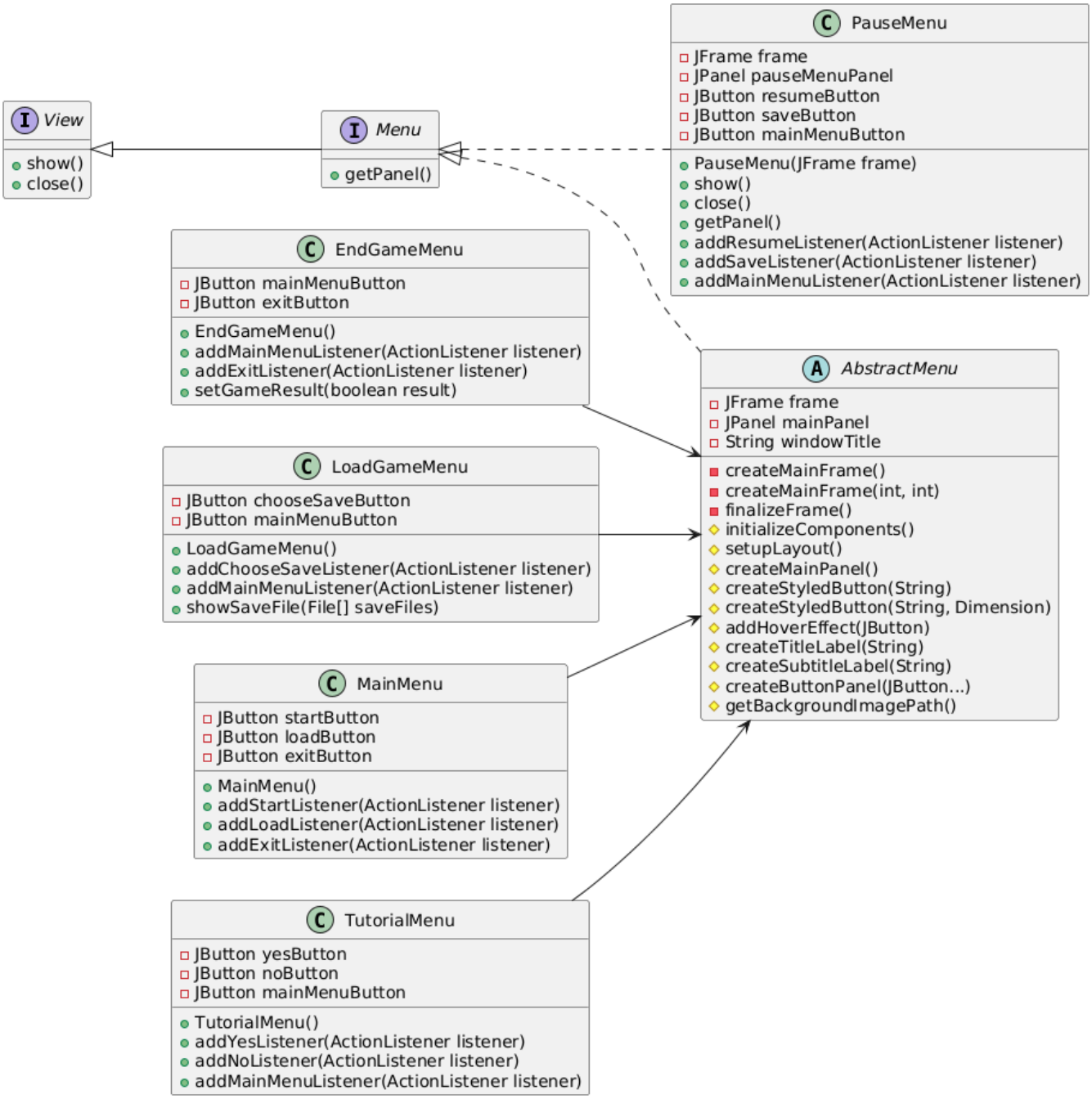

Figura 3.10: UML package Menu

– **SelectionMenu**: racchiude i sottomenu legati alla selezione
  * **AbstractSelectionMenu**: classe astratta che gestisce l'interfaccia e la logica di base per i menu di selezione.
  * **CharacterReplaceMenu**: menu che appare dopo che il livello della partita si conclude, consente di scegliere i personaggi nuovi da far rientrare in squadra.
  * **CharacterSelectionMenu**: gestisce il menu di selezione iniziale dei personaggi da inserire nella squadra.

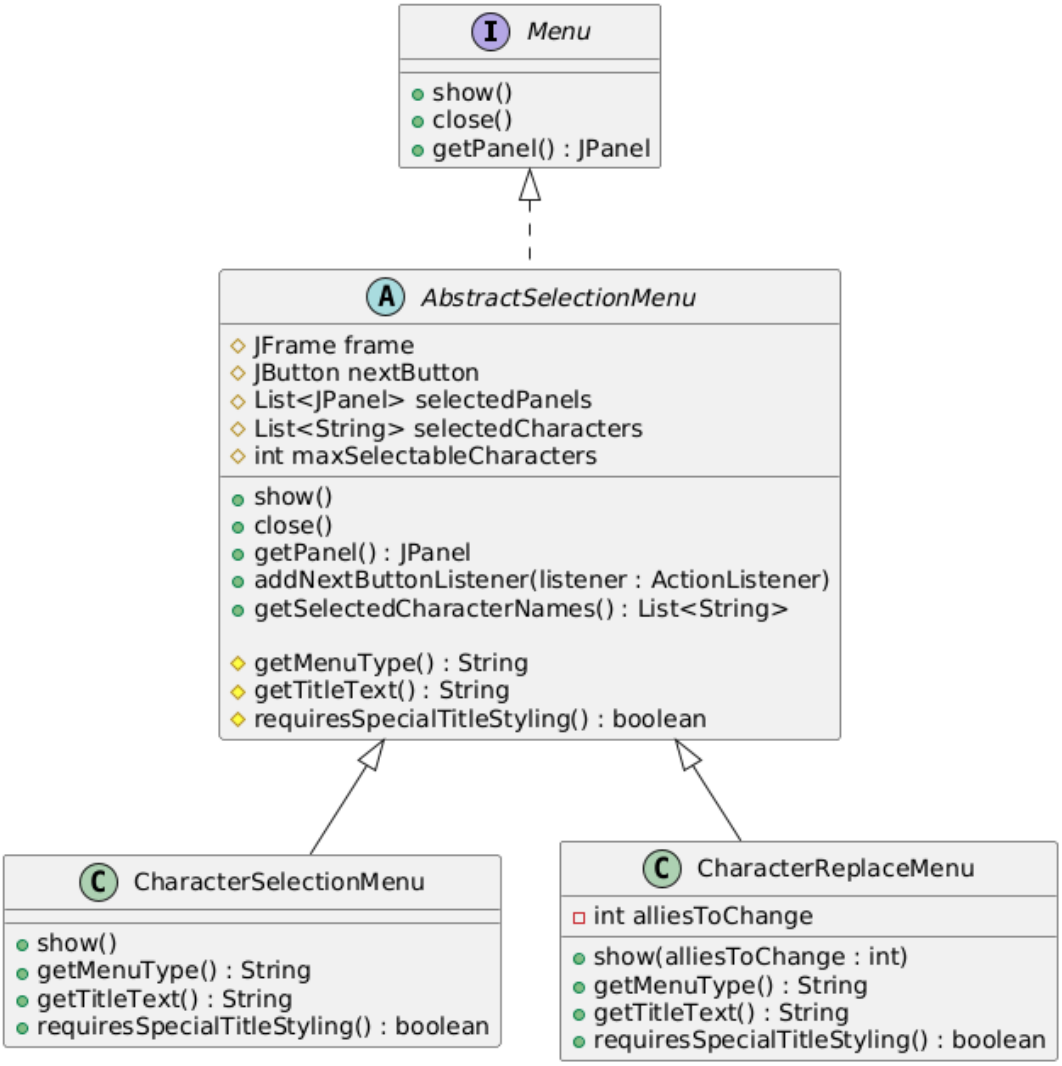

Figura 3.11: UML package Menu

3.2.3 Gaspar

Character

Il sistema di personaggi rappresenta il cuore del gameplay e dell'interazione nell'applicazione.
La progettazione si basa su una gerarchia ben strutturata che separa le responsabilità secondo i principi dell'Object-Oriented Design, garantendo estendibilità e riusabilità del codice.
La classe astratta `AbstractCharacter` funge da base comune per tutti i personaggi, implementando l'interfaccia `Character` e incapsulando le funzionalità condivise come statistiche di combattimento, gestione delle pozioni, sistema di esperienza e meccaniche di movimento.
Questo approccio segue il principio **DRY** (*Don't Repeat Yourself*), centralizzando la logica comune ed evitando duplicazioni di codice.
Per la specializzazione dei personaggi, il sistema utilizza l'ereditarietà attraverso classi concrete (`Archer`, `Barbarian`, `Juggernaut`, `Knight`, `Wizard`) che definiscono statistiche specifiche e equipaggiamenti caratteristici.
Ogni classe implementa un costruttore che genera valori casuali entro range predefiniti, garantendo varietà nel gameplay pur mantenendo il bilanciamento.
Ogni tipo di personaggio (detto anche Character o "classe" in gergo di RPG) può essere alleato o nemico, ed esiste per ognuno anche una sottoclasse di tipo Boss che dimostra l'estensibilità del sistema: essi ereditano dai personaggi base e implementano meccaniche speciali attraverso l'override di metodi chiave.
Questo pattern consente di aggiungere nuovi comportamenti senza modificare il codice esistente, rispettando il principio Open/Closed.
Tra le funzionalità principali del sistema troviamo:

- **Sistema di combattimento dinamico**: calcolo del danno basato su potenza e difesa, con controattacchi automatici e gestione della mortalità;
- **Progressione del personaggio**: sistema di esperienza con level-up automatico che incrementa tutte le statistiche;
- **Integrazione sistema armi**: ogni personaggio dispone di una di due possibili armi, che influenzano potenza, difesa, velocità e definiscono il range, ovvero la distanza di attacco;
- **Integrazione sistema delle pozioni**: supporto per quattro tipologie (salute, potenza, difesa, velocità) con applicazione automatica durante il combattimento;
- **Serializzazione intelligente**: supporto completo per il salvataggio con reinizializzazione automatica degli elementi transienti (come icona/immagine).

L'architettura garantisce un sistema flessibile e facilmente estendibile, permettendo l'aggiunta di nuovi tipi di personaggio e meccaniche senza compromettere la stabilità del codice esistente.

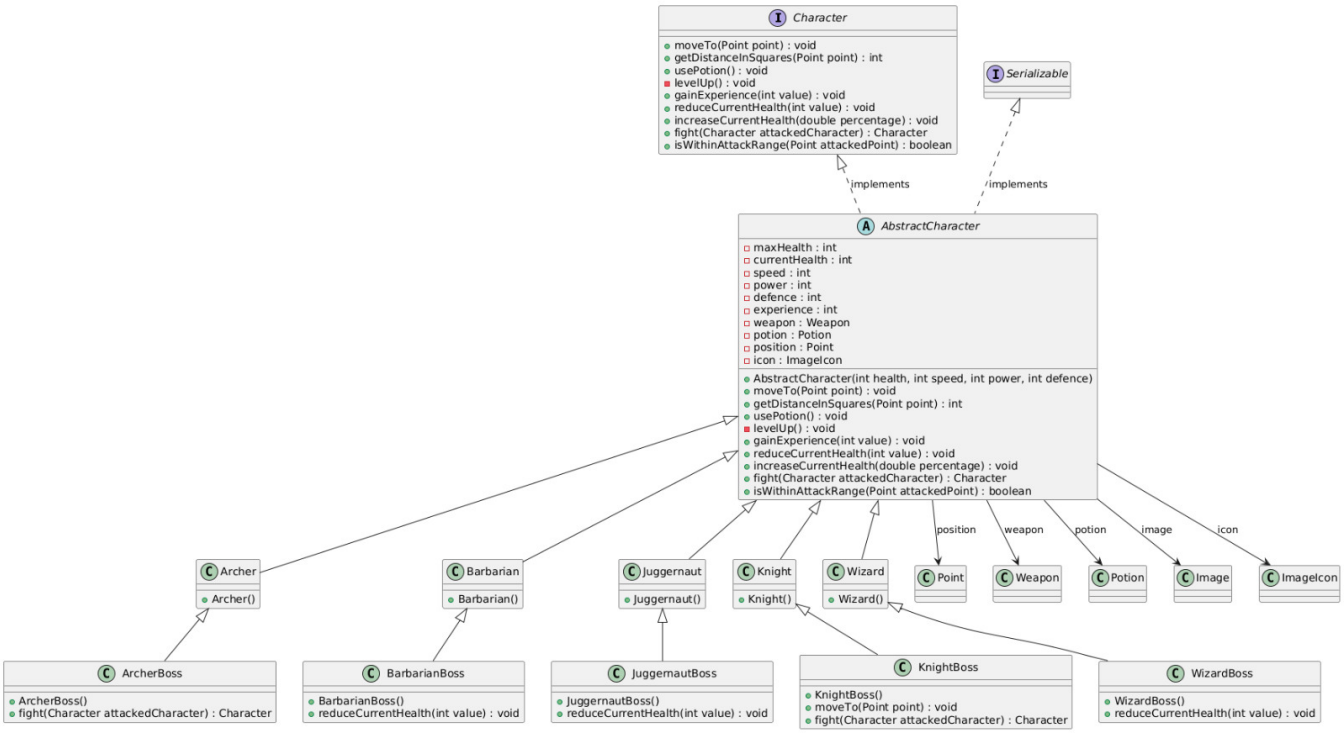

Figura 3.12: UML del package Characters

BattlePhaseView

La classe `BattlePhaseView` rappresenta il cuore dell'interfaccia utente durante le fasi di combattimento, gestendo l'interazione tra il giocatore e la griglia di gioco.
Questa componente implementa il pattern Model-View-Controller, fungendo da ponte tra la logica di gioco (`GameController`) e la rappresentazione visuale (`LevelMap`), garantendo una separazione netta delle responsabilità.
Il design della classe segue il principio di Single Responsibility, concentrandosi esclusivamente sulla gestione delle interazioni di combattimento senza occuparsi della logica di gioco sottostante.
La collaborazione con `GameController` per le operazioni di movimento e combattimento e con `LevelMap` per la gestione visuale dimostra l'applicazione del principio di Dependency Injection, rendendo il sistema modulare e testabile.
L'architettura utilizza un approccio event-driven attraverso l'uso di `ActionListener` dinamici, memorizzati in mappe che associano posizioni geografiche (`Point`) alle relative azioni.
Questo pattern consente di gestire interazioni complesse mantenendo il codice organizzato e garantendo la pulizia delle risorse dopo ogni fase.
La gestione dell'interfaccia utente implementa meccanismi di feedback visivo sofisticati, utilizzando diverse colorazioni per guidare il giocatore: grigio per le posizioni di movimento valide, rosso scuro per l'area di attacco e rosso per i nemici attaccabili.
L'uso di `AtomicBoolean` garantisce la thread-safety e previene azioni multiple accidentali.

Tra le funzionalità principali del sistema troviamo:

- **Gestione delle fasi di movimento**: calcolo dinamico delle posizioni raggiungibili basato sulla velocità del personaggio, con evidenziazione visuale e gestione dei listener per l'interazione;
- **Sistema di targeting**: identificazione automatica dei nemici nel raggio d'azione, con filtraggio basato sulle statistiche delle armi e feedback visivo immediato;
- **Interfaccia non bloccante**: utilizzo di callback (`Runnable`) per gestire il flusso asincrono delle operazioni, mantenendo la responsività dell'interfaccia;
- **Gestione sicura delle risorse**: pulizia automatica degli `ActionListener` con sincronizzazione per prevenire memory leak e interferenze tra fasi successive;
- **Integrazione con il sistema di personaggi**: accesso diretto alle proprietà dei personaggi per calcoli di movimento e combattimento senza violazione dell'incapsulamento;
- **Feedback informativo**: aggiornamenti in tempo reale dei banner e messaggi di stato per guidare l'esperienza utente.

L'architettura garantisce un'esperienza utente fluida e intuitiva, permettendo estensioni future del sistema di combattimento senza compromettere la stabilità dell'interfaccia esistente.

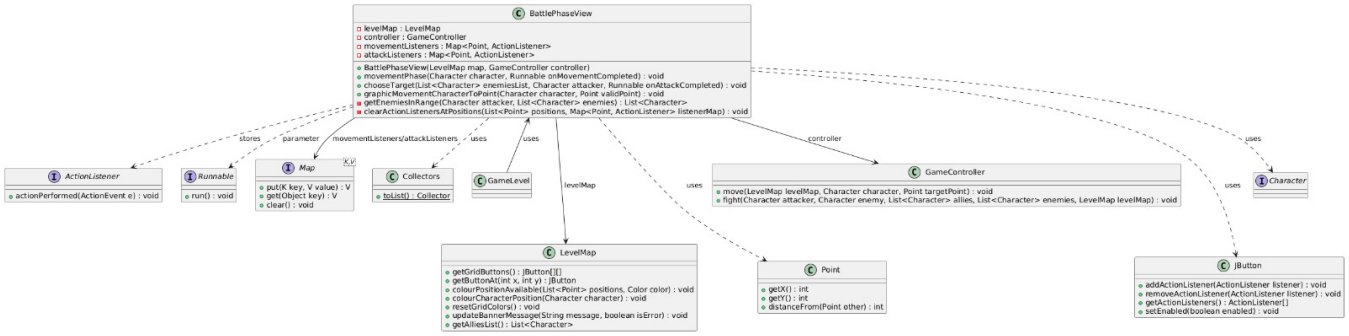

Figura 3.13: UML BattlePhaseView

Capitolo 4

Sviluppo

4.1 Testing automatizzato

Il testing automatizzato è stato implementato per verificare le funzionalità core del sistema *Five-Realms*. Per ogni classe di test sono riportati gli obiettivi, gli scenari, l'implementazione e i risultati attesi. I test sono stati realizzati utilizzando JUnit 5.

4.1.1 GameTest

Character Creation and Management Tests

- **testCreateAllies**
  - **Obiettivo:** Verificare la corretta creazione e inizializzazione degli alleati.
  - **Scenario:** Creazione della lista di alleati in un nuovo gioco.
  - **Implementazione:** Creazione degli alleati tramite il metodo `game.createAllies()` e controllo della presenza dei tipi specifici (Barbarian, Archer, Knight, Wizard, Juggernaut).
  - **Risultati attesi:** La lista non deve essere nulla, deve contenere esattamente 5 alleati, ognuno dei tipi previsti.
- **testCharacterSelection**
  - **Obiettivo:** Verificare la selezione e la configurazione dei personaggi.
  - **Scenario:** Selezione di 2 personaggi e impostazione delle loro posizioni.
  - **Implementazione:** Aggiunta di personaggi selezionati alla lista e chiamata di `becomeHero()` e `setPosition()`.
  - **Risultati attesi:** Tutti i personaggi devono essere alleati, vivi, con posizione non nulla, e il numero totale non deve superare il massimo consentito.
- **testAddSelectedCharacters**
  - **Obiettivo:** Verificare l'aggiunta di nuovi personaggi alla selezione esistente.
  - **Scenario:** Aggiunta di ulteriori personaggi a una lista già inizializzata.
  - **Implementazione:** Uso di `game.addSelectedCharacters()`.
  - **Risultati attesi:** La lista finale deve contenere tutti i personaggi previsti.

- **testCharacterMovement**
  - **Obiettivo:** Verificare il corretto posizionamento e movimento dei personaggi.
  - **Scenario:** Spostamento di un personaggio da una posizione iniziale a una finale.
  - **Implementazione:** Chiamata a `setPosition()` e verifica delle coordinate.
  - **Risultati attesi:** Le posizioni devono corrispondere a quelle impostate e rientrare nei limiti della mappa.
- **testCharacterStats**
  - **Obiettivo:** Validare le statistiche dei personaggi.
  - **Scenario:** Creazione degli alleati e verifica dei loro attributi.
  - **Implementazione:** Controllo dei valori di salute, potenza, difesa e salute massima.
  - **Risultati attesi:** Tutti i valori devono essere coerenti e positivi, con salute iniziale uguale a quella massima.

Game State and Flow Tests

- **testGameState**
  - **Obiettivo:** Controllare la correttezza dello stato iniziale del gioco.
  - **Scenario:** Creazione di una nuova istanza di gioco.
  - **Implementazione:** Verifica di costanti e dimensioni della mappa.
  - **Risultati attesi:** Tutti i valori devono essere positivi e coerenti con la definizione del gioco.
- **testGameInitialization**
  - **Obiettivo:** Verificare l'inizializzazione dei manager del gioco.
  - **Scenario:** Nuovo gioco.
  - **Implementazione:** Controllo dei manager tramite `assertNotNull`.
  - **Risultati attesi:** Tutti i manager devono essere inizializzati correttamente.
- **testLevelManagement**
  - **Obiettivo:** Verificare la gestione dei livelli.
  - **Scenario:** Impostazione di vari indici di livello validi.
  - **Implementazione:** Uso di `setCurrentLevelIndex()` e verifica dei valori.
  - **Risultati attesi:** Gli indici devono essere aggiornati correttamente e rientrare nei limiti.
- **testCharacterReplacementWorkflow**
  - **Obiettivo:** Verificare il flusso di sostituzione dei personaggi.
  - **Scenario:** Modifica del flag di attesa per la sostituzione.
  - **Implementazione:** Impostazione e verifica del flag tramite getter/setter.
  - **Risultati attesi:** Il flag deve riflettere correttamente lo stato corrente.
- **testTutorialMode**

- **Obiettivo:** Controllare l'esecuzione della modalità tutorial.
- **Scenario:** Avvio del tutorial.
- **Implementazione:** Chiamata a `game.startTutorial()`.
- **Risultati attesi:** Il metodo deve restituire un valore booleano senza errori.

Combat and Game Mechanics Tests

- **testCombat**
  - **Obiettivo:** Verificare il calcolo del danno in combattimento.
  - **Scenario:** Attacco tra due personaggi.
  - **Implementazione:** Calcolo del danno e confronto con la salute del difensore.
  - **Risultati attesi:** Danno positivo, salute aggiornata correttamente.
- **testCombatConsistency**
  - **Obiettivo:** Controllare la coerenza del calcolo del danno.
  - **Scenario:** Attacchi ripetuti tra due personaggi.
  - **Implementazione:** Uso di `@RepeatedTest` per verificare la consistenza.
  - **Risultati attesi:** La salute del difensore deve diminuire coerentemente ad ogni attacco.
- **testVictoryDefeat**
  - **Obiettivo:** Verificare le condizioni di vittoria e sconfitta.
  - **Scenario:** Vari scenari di battaglia (solo alleati, solo nemici, entrambi).
  - **Implementazione:** Controllo delle liste di alleati e nemici.
  - **Risultati attesi:** Identificazione corretta dello stato di vittoria, sconfitta o battaglia in corso.

Edge Cases and Error Handling Tests

- **testInvalidCharacterSelection**
  - **Obiettivo:** Gestire selezioni nulle o non valide.
  - **Scenario:** Selezione iniziale vuota e tentativo di impostare `null`.
  - **Implementazione:** Controllo delle eccezioni con `assertThrows` e verifica del recupero dello stato.
  - **Risultati attesi:** Deve lanciare `NullPointerException` e permettere il recupero dello stato con liste valide.
- **testEmptyCharacterSelection**
  - **Obiettivo:** Verificare la gestione di selezioni vuote.
  - **Scenario:** Impostazione di una lista vuota di personaggi selezionati.
  - **Implementazione:** Uso di `setSelectedCharacters()` con lista vuota.
  - **Risultati attesi:** La lista selezionata deve essere vuota ma non nulla.

- **testMaxAlliesSelection**
  - **Obiettivo:** Verificare il rispetto del limite massimo di alleati.
  - **Scenario:** Selezione di un numero di alleati pari e superiore al massimo consentito.
  - **Implementazione:** Creazione di liste con numero variabile di alleati.
  - **Risultati attesi:** Il sistema deve gestire correttamente sia il limite massimo che i casi di superamento.
- **testCharacterPositionSetting**
  - **Obiettivo:** Verificare l'impostazione di posizioni valide e non valide.
  - **Scenario:** Impostazione di posizioni all'interno e fuori dai confini della mappa.
  - **Implementazione:** Uso di `setPosition()` con coordinate valide e non valide.
  - **Risultati attesi:** Le posizioni devono essere impostate correttamente, con avvisi per coordinate fuori confine.

Performance and Resource Tests

- **testGameInitializationPerformance**
  - **Obiettivo:** Verificare le prestazioni dell'inizializzazione del gioco.
  - **Scenario:** Creazione multipla di istanze di gioco.
  - **Implementazione:** Uso di `@Timeout` per limitare il tempo di esecuzione.
  - **Risultati attesi:** Tutte le inizializzazioni devono completarsi entro 5 secondi.
- **testCharacterCreationPerformance**
  - **Obiettivo:** Verificare le prestazioni della creazione dei personaggi.
  - **Scenario:** Creazione ripetuta di gruppi di alleati.
  - **Implementazione:** Loop di creazione con controllo del tempo.
  - **Risultati attesi:** La creazione di 50 gruppi di alleati deve completarsi entro 3 secondi.
- **testMemoryCleanup**
  - **Obiettivo:** Verificare il corretto cleanup delle risorse.
  - **Scenario:** Selezione di personaggi e chiusura del gioco.
  - **Implementazione:** Chiamata a `game.closeAll()` dopo operazioni.
  - **Risultati attesi:** La chiusura deve completarsi senza errori e i manager devono rimanere accessibili.

Save/Load System Tests

- **testSaveManagerInitialization**
  - **Obiettivo:** Verificare l'inizializzazione del sistema di salvataggio.
  - **Scenario:** Creazione di un nuovo gioco.
  - **Implementazione:** Controllo del manager di salvataggio.

  - **Risultati attesi:** Il manager di salvataggio deve essere inizializzato correttamente.

- **testInvalidSaveFileHandling**
  - **Obiettivo:** Verificare la gestione di file di salvataggio non validi.
  - **Scenario:** Tentativo di caricamento da file inesistente.
  - **Implementazione:** Chiamata a `game.startLoadGame()` con file fittizio.
  - **Risultati attesi:** Il sistema deve gestire il caso senza lanciare eccezioni.

4.1.2 MapTest

Map Initialization and Basic Operations Tests

- **testMapCreation**
  - **Obiettivo:** Verificare la corretta creazione e inizializzazione della mappa di gioco.
  - **Scenario:** Creazione di una nuova mappa di livello con liste vuote di alleati e nemici.
  - **Implementazione:** Istanziazione di LevelMap e verifica dello stato iniziale.
  - **Risultati attesi:** La mappa non deve essere nulla e le liste di alleati e nemici devono essere vuote.
- **testSpawnCharacter**
  - **Obiettivo:** Verificare il corretto spawning dei personaggi sulla mappa.
  - **Scenario:** Creazione di un personaggio eroe e aggiunta alla lista degli alleati.
  - **Implementazione:** Utilizzo di `map.spawnCharacter()` e verifica dell'occupazione della posizione.
  - **Risultati attesi:** La posizione del personaggio deve risultare occupata sulla mappa dopo lo spawning.
- **testMoveCharacter**
  - **Obiettivo:** Verificare il movimento corretto dei personaggi sulla mappa.
  - **Scenario:** Spostamento di un personaggio da una posizione iniziale a una target.
  - **Implementazione:** Utilizzo di `controller.move()` e verifica della nuova posizione.
  - **Risultati attesi:** Il personaggio deve essere nella nuova posizione target e questa deve risultare occupata.
- **testRemoveCharacter**
  - **Obiettivo:** Verificare la rimozione corretta dei personaggi dalla mappa.
  - **Scenario:** Rimozione di un personaggio dalla mappa e dalla lista degli alleati.
  - **Implementazione:** Utilizzo di `controller.remove()` e verifica della lista alleati.
  - **Risultati attesi:** La lista degli alleati deve essere vuota dopo la rimozione del personaggio.

Visual and UI Tests

- **testColourPositionAvailable**
  - **Obiettivo:** Verificare la colorazione delle posizioni disponibili sulla mappa.
  - **Scenario:** Colorazione di una lista di posizioni disponibili con un colore specifico.
  - **Implementazione:** Utilizzo di `map.colourPositionAvailable()` e verifica della validità delle coordinate.
  - **Risultati attesi:** Tutte le posizioni colorate devono avere coordinate valide all'interno dei confini della mappa.
- **testColourCharacterPosition**

- **Obiettivo:** Verificare la colorazione della posizione del personaggio sulla mappa.
- **Scenario:** Colorazione della posizione corrente di un personaggio eroe.
- **Implementazione:** Utilizzo di `map.colourCharacterPosition()` e verifica dello stato della posizione.
- **Risultati attesi:** La posizione del personaggio deve essere valida e occupata sulla mappa.

4.2 Metodologia di lavoro

Il progetto è stato sviluppato seguendo un approccio collaborativo e iterativo, caratterizzato da una forte integrazione tra i membri del team durante tutte le fasi di sviluppo.
La metodologia adottata ha favorito sia il lavoro individuale sia la collaborazione, garantendo coerenza architetturale e qualità del codice prodotto.
La fase iniziale è stata dedicata all'analisi e alla progettazione dell'architettura, durante la quale tutti i membri del team hanno lavorato a stretto contatto per definire attraverso diagrammi UML e sessioni di brainstorming la struttura del sistema.
Questa fase collaborativa ha permesso di stabilire una visione condivisa del progetto e di identificare le interfacce tra i diversi moduli.
La collaborazione è stata particolarmente intensa durante le fasi iniziali di implementazione e durante l'integrazione finale di tutti i moduli, momenti cruciali per garantire il corretto funzionamento del sistema nel suo insieme.

4.2.1 Workflow

Per organizzare il lavoro è stato utilizzato un documento condiviso in cui venivano tracciati gli step raggiunti, quelli in corso di sviluppo e gli obiettivi futuri.
Questo strumento ha permesso di mantenere una visione d'insieme sullo stato di avanzamento del progetto e di coordinare efficacemente le attività individuali.
La comunicazione tra i membri del team è stata molto importante e cruciale, che ha facilitato il confronto costante su questioni tecniche, la risoluzione di problematiche emergenti e la condivisione di decisioni progettuali.

4.2.2 Version Control

Il controllo di versione è stato gestito attraverso Git, utilizzando un repository condiviso, dove il branch principale era **main**, dal quale sono state fatte le release.
Ogni sviluppatore ha seguito le best practice per i commit, utilizzando messaggi descrittivi che documentassero chiaramente le modifiche apportate.
Questo approccio ha garantito la tracciabilità delle modifiche e ha facilitato l'integrazione del codice sviluppato dai diversi membri del team.

4.2.3 Gasperoni

Realizzazione completa del package **gameStatus**, responsabile della gestione della logica dello stato di gioco, dei livelli e del ciclo principale di aggiornamento, e del package **controller**, incaricato della gestione centralizzata delle interazioni tra i componenti del sistema.

Il mio contributo si è concentrato su due aspetti principali del progetto:

- **Package *gameStatus***: implementazione delle classi **Game**, **GameLevel** e **GameStateManager**, responsabili della gestione dei livelli, dell'aggiornamento continuo del gioco tramite timer, del salvataggio e del caricamento dello stato, e della gestione degli eventi di gioco (movimenti, combattimenti e interazioni tra personaggi). Ho inoltre progettato il sistema di salvataggio integrato, articolato nelle classi **GameSave** e **GameSaveManager**, per la memorizzazione dello stato del gioco e dei progressi dei giocatori, inclusi alleati, nemici e parametri di livello.
- **Package *controller***: sviluppo della classe **GameController**, che funge da coordinatore principale tra i vari moduli del gioco, inclusi **MusicController**, **MapController**, **MenuController** e **SaveController**. Gestione dell'integrazione tra i diversi componenti, garantendo coerenza tra la logica di gioco e l'interfaccia utente, con metodi per il caricamento, la pausa e la ripresa dei livelli, oltre alla comunicazione tra sistemi grafici, logici e di input.

Durante lo sviluppo ho collaborato con il team per l'integrazione dei vari package del progetto, assicurando che la logica di gioco, il sistema di salvataggio e il controller fossero coerenti con l'architettura complessiva e con l'esperienza utente finale.

4.2.4 Marchionni

Realizzazione completa del package **view** che rappresenta l'interfaccia utente e del package **equipment** con i rispettivi sistemi di pozioni e armi equipaggiabili.
Il mio contributo si è focalizzato su due componenti fondamentali del progetto, per quanto riguarda l'interfaccia grafica, il package **view**, suddiviso nel package **map** per la gestione della visualizzazione e interazione con le mappe di gioco (includendo classi come `AbstractMap`, `GridPanel`, `LevelMap`) e nel package **menu** per tutti i menu interattivi del sistema (come `MainMenu`, `EndGameMenu`, `LoadGameMenu` e il sottosistema di selezione personaggi).
Parallelamente, ho progettato e implementato il sistema di equipaggiamento attraverso il package **equipment**, articolato nel package **potions** per la gestione delle pozioni utilizzabili dai personaggi (`PotionHealth`, `PotionPower`, `PotionDefence`, `PotionSpeed`) e nel package **weapons** per le diverse tipologie di armi equipaggiabili (`Sword`, `Bow`, `Axe`, `Staff`, `Spear`, `Wand`).
Durante lo sviluppo ho collaborato con il team per l'integrazione delle interfacce tra moduli, garantendo coerenza nell'architettura e nell'esperienza utente del sistema complessivo.

4.2.5 Gaspar

Realizzazione completa del package **character** che rappresenta i personaggi di gioco e della classe `BattlePhaseView` che rappresenta la gestione della parte grafica di movimento e attacco durante il turno del giocatore, e loro integrazioni in altri package e classi, oltre che al bilanciamento dei valori delle statistiche usate dai personaggi e dai loro equipaggiamenti.
Questo include l'interfaccia `Character`, la classe astratta `AbstractCharacter`, le classi concrete `Archer`, `Knight`, `Barbarian`, `Wizard`, `Juggernaut`, le classi boss `ArcherBoss`, `KnightBoss`, `BarbarianBoss`, `WizardBoss`, `JuggernautBoss`, le quantità e tipi di nemici che appaiono in ogni livello, le immagini associate ai background dei livelli ed ai personaggi (trovate online e modificate con TokenStamp), il metodo `startAITurn()` che gestisce la logica degli NPC (*Non-Playable Character*) e la classe `BattlePhaseView`.
Durante lo sviluppo ho collaborato con il team per l'integrazione di queste classi e metodi all'interno di quelle che ne necessitano uso, per mantenere quanto possibile coerenza nell'architettura e nell'esperienza utente del sistema complessivo.

Le seguenti componenti sono state realizzate attraverso un lavoro collaborativo tra tutti i membri del team:

- **Package View**: Data l'estensione e la complessità dell'interfaccia grafica, il lavoro è stato suddiviso equamente tra i membri del team, mantenendo però un approccio collaborativo per le componenti fondamentali. In particolare, sono state sviluppate insieme le classi astratte `AbstractMenu`, `AbstractSelectionMenu`, `AbstractMap` e `BattlePhaseView`, così come le interfacce `Menu`, `View` e `Map`. Questa scelta ha garantito coerenza nell'implementazione dell'interfaccia utente e ha facilitato l'integrazione tra le diverse viste del sistema.

- **Package gameStatus**: Rappresentando il cuore del flusso di gioco, questo package è stato sviluppato interamente attraverso la collaborazione di tutti i membri del team per garantire una perfetta integrazione tra le componenti. Particolare attenzione è stata dedicata alle classi `Game` e `Level`, che gestiscono la logica principale del gioco, e alle classi relative al sistema di salvataggio e caricamento (`GameSave` e `GameSaveManager` nel package `saveSystem`), fondamentali per la persistenza dello stato di gioco.

- **Package Controller**: Il Controller, essendo responsabile della coordinazione tra la logica e l'interfaccia utente, è stato sviluppato collaborativamente per assicurare una corretta gestione del flusso di controllo e una adeguata separazione delle responsabilità secondo il pattern MVC adottato.

Questa metodologia di lavoro ha permesso di lavorare in maniera efficiente con la necessità di mantenere coerenza architetturale, risultando in un prodotto finale ben integrato e di qualità.

4.3 Note di sviluppo

4.3.1 Gasperoni

Durante lo sviluppo del progetto ho fatto ampio uso di concetti della **programmazione orientata agli oggetti (OOP)** e di strumenti Java per strutturare in modo modulare e coerente il codice. Le principali tecnologie e strumenti utilizzati sono:

- **Timer e multithreading**: utilizzati nel package `gameStatus` per aggiornare il ciclo di gioco senza bloccare la GUI, permettendo la gestione sicura di pause e riprese dei livelli.
- **Stream**: impiegate per elaborare liste di personaggi, nemici, alleati e oggetti, semplificando operazioni di filtraggio, ordinamento e trasformazioni dei dati in modo funzionale e leggibile.
- **Lambda expressions**: utilizzate nel package `controller` per rendere il codice più conciso e leggibile, in particolare per l'implementazione di callback, strategie e operazioni su collezioni.
- **Event Handling**: utilizzato nel `MenuController` per gestire in modo modulare i click e le interazioni dell'utente con i menu del gioco.
- **Collezioni principali**:
  - `List<Character>`: per gestire le liste di alleati e nemici;
  - `PriorityQueue<Character>`: per determinare l'ordine dei turni dei personaggi;
  - `Map<String, String>`: per gestire lo stato e le tracce della musica in `MusicManager`;
  - `enum`: per rappresentare i vari stati del gioco e dei livelli.

4.3.2 Marchionni

- **Strutture dati principali utilizzate**: Sono state utilizzate principalmente le collezioni del framework Java, in particolare **List** e la sua implementazione **ArrayList** per gestire sequenze ordinate di elementi, e **Map** per associare chiavi uniche a valori, facilitando la ricerca e l'organizzazione dei dati.

- **Stream**: Utilizzate estensivamente per la gestione e filtraggio delle liste di personaggi, in particolare nel metodo **isPositionOccupied()** della classe **AbstractMap** per verificare se una posizione è già occupata da alleati o nemici attraverso operazioni di filtering e matching sui flussi di dati.

```
public boolean isPositionOccupied(Point point) {
    if (point == null) return false;

    return this.alliesList.stream()
        .filter(ally -> ally != null && ally.getPosition() != null)
        .anyMatch(ally -> ally.getPosition().equals(point))
        ||
        this.enemiesList.stream()
        .filter(enemy -> enemy != null && enemy.getPosition() != null)
        .anyMatch(enemy -> enemy.getPosition().equals(point));
}
```

- **Lambda expressions**: Utilizzate in combinazione con **Stream** nella classe **AbstractMap**, particolarmente per il controllo delle posizioni dei personaggi nelle liste di alleati e nemici attraverso operazioni di filtering e mapping.

- **Java Swing avanzato**: Implementazione di interfacce grafiche nella classe **AbstractMap** utilizzando **JLayeredPane** per la gestione di layer multipli, **Timer** per la comparsa e scomparsa dei banner delle statistiche dei personaggi e messaggi a schermo, **JWindow** associata al tooltip del personaggio specifico, **JPanel** per visualizzare a schermo il pannello di gioco, **JFrame** e **JButton**.

- **Event Handling**: Gestione avanzata di eventi multipli (**ActionListener**, **MouseListener**, **MouseMotionListener**) sui componenti della griglia nella classe **AbstractMap** e nelle classi del package **menu**, con implementazione di pulizia completa degli eventi nel metodo **removeAllEvent()** per evitare memory leak durante la chiusura delle mappe.

- **Graphics e Image Processing**: Utilizzo di **java.awt.Graphics** nella classe **AbstractMap** per il ridimensionamento dinamico delle immagini di background nel metodo **initializeBackgroundMap()** e la gestione delle trasparenze, con scaling automatico basato sulle dimensioni del pannello.

```
private void initializeBackgroundMap() {
    String backgroundFile = "images/background/background" + numLevel + ".jpg";
    ImageIcon backgroundImage = new ImageIcon(backgroundFile);

    if (backgroundImage.getIconWidth() > 0 && backgroundImage.getIconHeight() > 0) {
        Dimension panelSize = this.layeredPanel.getSize();
        int panelWidth = panelSize.width;
        int panelHeight = panelSize.height;

        Image image = backgroundImage.getImage();
        Image resizedImage = image.getScaledInstance(panelWidth, panelHeight,
            Image.SCALE_SMOOTH);
        backgroundImage = new ImageIcon(resizedImage);
    }
}
```

- **Dynamic UI Layout**: Implementazione di layout responsivi che si adattano alle dimensioni dello schermo nella classe **AbstractMap** (metodo **initializeFrame()**), calcolando dinamicamente posizioni e dimensioni dei componenti basandosi sulla risoluzione del display dell'utente.

```
private void initializeFrame() {
    // Set window dimensions based on screen resolution
    Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
    int width = (int) (screenSize.getWidth() * 0.6);  // 60% of screen width
    int height = (int) (width * 3.0 / 4.0);           // 4:3 aspect ratio
    this.frame.setSize(width, height);

    // Center the window on screen
    this.frame.setLocationRelativeTo(null);
}
```

- **Tooltip Management**: Sistema avanzato di gestione dei tooltip per i personaggi implementato nella classe **CharacterTooltipManager** e utilizzato in **AbstractMap**, con aggiornamento dinamico delle informazioni nel metodo **updateToolTip()** e cleanup automatico per prevenire sovrapposizioni e memory leak.

4.3.3 Gaspar

- **Event Handling**: Gestione avanzata di eventi (`ActionListener` di tipo click sui pulsanti della griglia nella classe `BattlePhaseView`), con implementazione di pulizia completa degli eventi nei metodi dedicati per evitare memory leak.
- **Lambda**: Utilizzate per impostare gli `ActionListener` di `BattlePhaseView`.
- **Strutture dati principali utilizzate**: Sono state utilizzate principalmente le collezioni del framework Java, in particolare `List` e la sua implementazione `ArrayList` per gestire sequenze ordinate di elementi, e `Map` per associare chiavi uniche a valori, facilitando la ricerca e l'organizzazione dei dati, con la sua implementazione `HashMap`.
- **Stream**: Utilizzate estensivamente per la gestione e filtraggio delle liste di personaggi, come nei metodi `startAITurn()` della classe `GameLevel`, `fight()` della classe `AbstractCharacter`, e `chooseTarget()` di `BattlePhaseView`.

```
private void startAITurn() {
    try {
        Thread.sleep(1500);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("AI Turn");
    Character victim = alliesList.stream()
            .min(Comparator.comparing(charac ->
                charac.getDistanceInSquares(currentAttacker.getPosition())))
            .orElse(null);
    if (victim == null) {
        this.currentTurnState = RoundState.TURN_COMPLETED;
        return;
    }
    Set<Point> occupiedPositions = new HashSet<>();
    alliesList.stream().forEach(character -> occupiedPositions.add(character.getPosition()));
    enemiesList.stream().forEach(character ->
        occupiedPositions.add(character.getPosition()));
    movementPhaseManager.graphicMovementCharacterToPoint(currentAttacker,
        this.availablePositions.stream()
            .filter(point -> currentAttacker.getDistanceInSquares(point) <=
                (currentAttacker.getSpeed() / AbstractCharacter.SPEED_TO_MOVEMENT))
            .filter(point -> !occupiedPositions.contains(point))
            .filter(point -> victim.getDistanceInSquares(point) <=
                currentAttacker.getRange()) // Within attack range
            .min(Comparator.comparing(point -> currentAttacker.getDistanceInSquares(point)))
                // Closest to current position
            .orElseGet(() -> // If no immediate attack positions available, simply chase
            availablePositions.stream()
            .filter(point -> currentAttacker.getDistanceInSquares(point) <=
                (currentAttacker.getSpeed() / AbstractCharacter.SPEED_TO_MOVEMENT))
            .filter(point -> !occupiedPositions.contains(point))
            .min(Comparator.comparing(point -> victim.getDistanceInSquares(point))) //
                Closest to victim
            .orElse(currentAttacker.getPosition())
                    ));
    try {
        this.controller.fight(currentAttacker, victim, alliesList, enemiesList, levelMap);
    } catch (IllegalArgumentException e) {
```

```
        // Handle the case where even the closest enemy is still out of attack range after
            movement
        System.out.print("␣No␣enemy␣in␣attack␣range.␣Attack␣phase␣cancelled.");
    }
    this.currentTurnState = RoundState.TURN_COMPLETED;
}

public Character fight(Character attackedCharacter) throws IllegalArgumentException {
    Character deadCharacter = null;
    if(this.isAllied() == attackedCharacter.isAllied())
        throw new IllegalArgumentException("You␣cannot␣attack␣someone␣belonging␣to␣your␣own␣
            faction!");
    if (!this.isWithinAttackRange(attackedCharacter))
        throw new IllegalArgumentException("You␣cannot␣attack␣someone␣outside␣of␣your␣
            weapon's␣attack␣range!");
    // Use potion if beneficial (not health potion when at max health)
    if(this.hasPotion() && !(this.getPotion() instanceof PotionHealth &&
        this.getCurrentHealth() == this.getMaxHealth())) {
        this.usePotion();
    }
    attackedCharacter.reduceCurrentHealth(this.getPower() - attackedCharacter.getDefence());
    if (attackedCharacter.isAlive() && attackedCharacter.isWithinAttackRange(this))
        this.reduceCurrentHealth(attackedCharacter.getPower() - this.getDefence());
    if (!attackedCharacter.isAlive()) {
        deadCharacter = attackedCharacter;
        this.gainExperience(AbstractCharacter.EXP_LEVELUP_THRESHOLD/3);
        //50% chance of getting a potion, if so get one of the four randomly
        switch(rand.nextInt(0,9)) {
        case 5:
            this.setPotion(new PotionHealth());
            break;
        case 6:
            this.setPotion(new PotionDefence());
            break;
        case 7:
            this.setPotion(new PotionPower());
            break;
        case 8:
            this.setPotion(new PotionSpeed());
            break;
        }
    }
    if (!this.isAlive()) {
        deadCharacter = this;
        attackedCharacter.gainExperience(AbstractCharacter.EXP_LEVELUP_THRESHOLD/2);
    }
    return deadCharacter;
}

public void chooseTarget(List<Character> enemiesList, Character attacker, Runnable
    onAttackCompleted) {
    System.out.println("\n" + attacker.getClass().getSimpleName() + "␣(HP:␣" +
        attacker.getCurrentHealth() + ",␣DMG:␣" + attacker.getPower() +
            ")␣select␣who␣you␣want␣to␣attack␣->␣");

    // Get enemies we can attack
        List<Character> reachableEnemies = getEnemiesInRange(attacker, enemiesList);

        if (reachableEnemies.isEmpty()) {
                System.out.println("␣No␣enemies␣in␣attack␣range.␣Attack␣phase␣cancelled.");
```

```
            if (onAttackCompleted != null) onAttackCompleted.run();
            return;
    }
    else {
    this.levelMap.updateBannerMessage("Movement␣completed,␣
        "+attacker.getClass().getSimpleName()+"␣choose␣a␣target", false);
    }

    // Get positions of enemies we can attack
    List<Point> enemyPositions = reachableEnemies.stream()
                                            .map(Character::getPosition)
                                            .collect(Collectors.toList());
    // Get all positions within attack range
    List<Point> positionsInAttackRange = new ArrayList<>();
    JButton[][] buttonGrid = this.levelMap.getGridButtons();

for (int row = 0; row < buttonGrid.length; row++) {
    for (int col = 0; col < buttonGrid[row].length; col++) {
        Point point = new Point(row, col);

        if (attacker.isWithinAttackRange(point)) {
            positionsInAttackRange.add(point);
        }
    }
}

    // Highlight attackable enemies in red
    levelMap.colourPositionAvailable(positionsInAttackRange, new Color(139, 0, 0, 80));
        // dark red
    levelMap.colourPositionAvailable(enemyPositions, new Color(255, 0, 0, 80)); // red
for (Character enemy : enemiesList) {
    synchronized (this.attackListeners) {
        Point enemyPosition = enemy.getPosition();

        JButton button = levelMap.getButtonAt(enemyPosition.getX(),
            enemyPosition.getY());
        ActionListener attackListener = click -> {
            /* for debug
            System.out.println(" has targeted  " + enemy.getClass().getSimpleName() +
                               " (HP: " + enemy.getCurrentHealth() + ", DEF: " +
                                   enemy.getDefence() + ")");
                               */

            this.controller.fight(attacker, enemy, levelMap.getAlliesList(),
                enemiesList, levelMap);
            /* for debug
            System.out.println("\nPost attack " + enemy.getClass().getSimpleName() +
                               " has " + enemy.getCurrentHealth() + " hp");*/

            this.clearActionListenersAtPositions(enemyPositions, this.attackListeners);
            // Unblock the execution flow
            if (onAttackCompleted != null) onAttackCompleted.run();
        };
        // Remove any old listeners
        for (ActionListener al : button.getActionListeners()) {
            button.removeActionListener(al);
        }

        this.attackListeners.put(enemyPosition, attackListener);
        button.addActionListener(attackListener);
```

```
            button.setEnabled(true);
                }
    }
}
```

Character

L'attacco in particolare, gestito dal metodo `fight()`, agisce nel modo più efficiente e sicuro possibile per gestire questo step fondamentale del gioco, controllando prima se il personaggio attaccato è della stessa fazione dell'attaccante (attaccare in questo caso è proibito) e che non si stia attaccando al di fuori del proprio range di attacco.
Il metodo procede poi a decidere se ha senso usare la propria pozione se il personaggio attaccante ne ha una (non ha senso usare una pozione di cura se si è a salute massima), per poi andare a infliggere il danno appropriato al nemico.
Il calcolo del danno è dato numericamente dal proprio totale di *Power* meno il totale di *Defense* nemico, che può risultare in un valore zero ma il metodo stesso normalizza il valore nei casi in cui vada sotto zero.
Se il proprio nemico sopravvive a questo danno, contrattacca, infliggendo danno all'attaccante.
Infine, si controlla se uno dei due personaggi è morto durante il combattimento: se il nemico è morto, l'attaccante ha una possibilità di guadagnare una pozione e guadagna esperienza; se l'attaccante è morto, il suo nemico guadagna esperienza.
Se uno dei due è morto, esso è il valore di ritorno della funzione che altrimenti ritorna `null` (in questo modo, il controller che gestisce le liste di personaggi può capire chi rimuovere se necessario).

AI Nemica

Il metodo `startAITurn()` di `GameLevel` fa ampio uso delle stream: innanzitutto per trovare il personaggio avversario più vicino (attualmente si tratta del personaggio del giocatore più vicino, dato che l'**IA** è implementata solo per i nemici, ma potrebbe facilmente essere riadattata a funzionare per entrambe le fazioni se necessario, grazie al design modulare del metodo) tramite l'operazione `min()` su un comparatore che valuta la distanza tra il personaggio IA in questione e ciascun personaggio della fazione opposta.
Ottenuto questo personaggio, filtra tutte le possibili posizioni finali su cui muoversi, escludendo tutte quelle più distanti del suo range di movimento, esclude tutte quelle già occupate, per poi escludere tutte quelle da cui non potrebbe attaccare la sua vittima scelta.
Se rimangono posizioni, allora sceglie quella più vicina alla propria (passo fondamentale per evitare che classi *ranged* come `Archer` e `Wizard` non si avvicinino più del necessario ad altri, cosa che non avrebbe senso), prendendone una a caso se ci sono posizioni di distanza uguale tra loro.
Se non esistono posizioni a questo step, allora va in modalità *chase*, cercando semplicemente la posizione più vicina alla propria vittima e che rientra dentro il suo range di movimento, altrimenti filtrando come prima. Se tale posizione non esistesse per qualche motivo, allora rimane sulla propria posizione.

BattlePhaseView

Viene usato da e contenuto dentro `GameLevel`.
Il metodo `movementPhase()` serve per evidenziare le caselle in cui il personaggio del giocatore può muoversi e applicare `ActionListener` cliccabili per il movimento su quelle caselle; poi gestisce la rimozione di tutti i listener dopo che uno è stato utilizzato.
Il metodo `chooseTarget()` fa lo stesso per le caselle entro la portata d'attacco dopo il movimento,

aggiungendo in particolare i mouse click listener a qualsiasi nemico attaccabile e rimuovendo tutti questi listener dopo che uno è stato utilizzato.

Sorgenti

https://github.com/andrymarchio13/FiveRealms