A Knight's Tale

Maicol Battistini, Simone Redighieri, Leonardo Viola

13 settembre 2022

Indice

Capitolo 1

Analisi

1.1 Requisiti

Si vuole realizzare un gioco action 2D a livelli, denominato "A Knight's Tale", in cui il giocatore, impersonando un cavaliere, deve sconfiggere tutti i nemici per completare il livello. Il giocatore poi potrà visualizzare una classifica al termine di una partita dove verranno registrati tutti i punteggi ottenuti dai vari giocatori. I giocatori devono registrarsi all'inizio di ogni partita inserendo il proprio nome.

Requisiti funzionali

- Il gioco dovrà contenere un singolo livello nella sua prima versione. Verrà poi ampliato nelle versioni successive del gioco.
- I personaggi/Le entità non potranno oltrepassarsi a vicenda e non potranno oltrepassare gli ostacoli presenti nella mappa di gioco.
- Il personaggio controllato dall'utente dovrà riuscire ad attaccare i nemici.
- Il menù principale dovrà permettere all'utente di iniziare una nuova partita, uscire dal gioco e accedere alla classifica dei giocatori.
- La classifica dovrà visualizzare i nomi e i punteggi dei giocatori ottenuti ad ogni partita.
- I nemici dovranno muoversi autonomamente, mentre il personaggio controllato dall'utente potrà muoversi in base alla volontà del giocatore.
- Dovranno esistere varie tipologie di nemici, che potranno essere più complicate o più semplici da sconfiggere.

Requisiti non funzionali

- Il gioco deve poter essere eseguito sui principali sistemi operativi per computer (Windows, Linux, MacOS).

1.2 Analisi e modello del dominio

Sono presenti 2 tipi di entità che possono muoversi: il personaggio principale e i nemici. Il personaggio controllato dal giocatore dovrà cercare di eliminare tutti i nemici presenti nella mappa: dovrà quindi essere in grado di attaccarli. I nemici, a loro volta, dovranno cercare di inseguire il giocatore per eliminarlo e di conseguenza far perdere l'utente. Ogni entità, che non sia un ostacolo, è dotata di salute e difesa, le quali potranno diminuire in seguito ad un attacco di un'altra entità. Quando la salute di un'entità "finisce", l'entità risulta sconfitta.

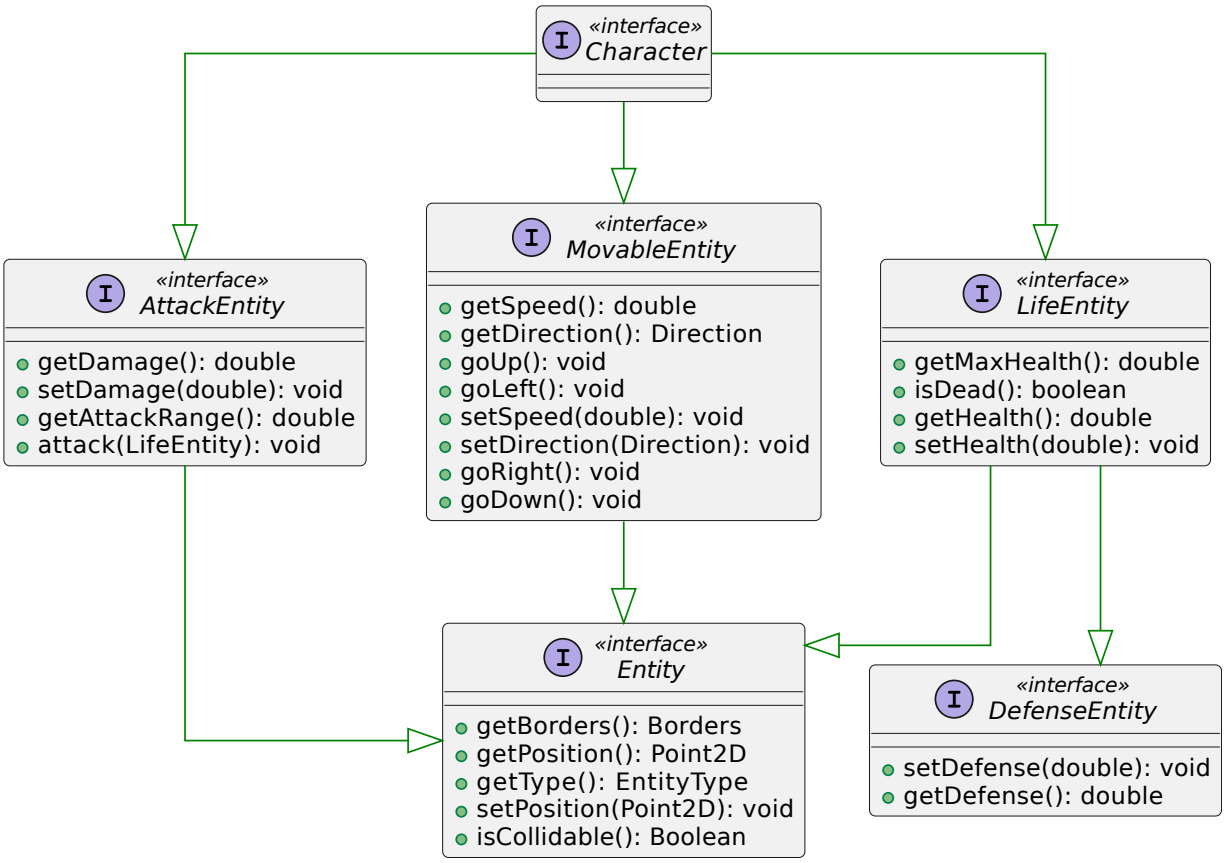

Figura 1.1: Schema UML delle principali entità

La classifica terrà traccia del nome del giocatore e dei punti effettuati in una partita e dovrà essere persistente.

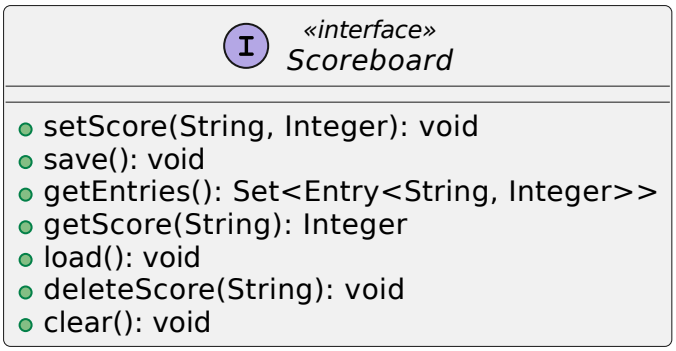

Figura 1.2: Schema UML della scoreboard

Capitolo 2

Design

2.1 Architettura

L'architettura di A Knight's Tale segue il pattern architetturale MVC e utilizza anche il pattern Factory, utilizzato soprattutto per creare rapidamente ogni componente del progetto (Controller, View e Entità (giocatore e nemici)). In particolare, per quanto riguarda la separazione della view dagli altri componenti, esiste una interfaccia generale per la view, che viene estesa da tutte le "schermate" del gioco, e il controller della view, che viene esteso da tutti i controller che sono collegati ad una view. I controller hanno metodi generici e non interagiscono direttamente con la view, così da favorire la reimplementazione della view. Queste poi vengono "unite" insieme tramite le relative factory, senza dover collegare manualmente view e controller.

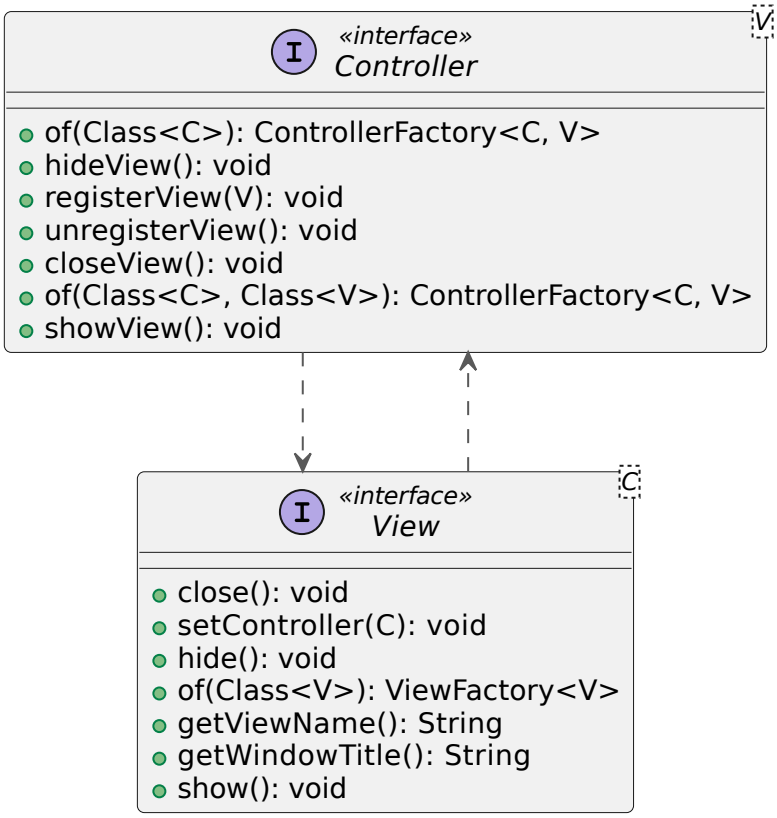

Figura 2.1: Schema UML del collegamento tra View e Controller

Ogni entità aderisce al pattern MVC, in cui è presente un controller che ha il compito di coordinare il funzionamento del model e della view. Le entità "basi", che fungono da ostacoli all'interno della mappa del gioco, hanno un semplice controller per favorire l'interazione tra model e view. I personaggi (giocatore e nemici), invece, estendono i vari compononenti delle entità basi e sono costituiti da un controller più complesso con funzionalità aggiuntive in cui ad ogni aggiornamento del model vengono controllate le collisioni con le altre entità e la finestra di gioco e successivamente viene aggiornata la view.

2.2 Design dettagliato

Maicol Battistini

La mia parte con maggior rilievo all'interno del progetto è stata la creazione delle finestre/view in JavaFX in FXML e riuscire a farle coordinare al meglio con un controller generico (in alcun modo collegato a JavaFX) e al

model per permettere una semplice sostituzione della view. Questo problema è stato risolto creando un sistema simile alla dependency injection, anche se semplificato, e ricorrendo al pattern Factory. In particolare, con l'obiettivo di rendere questa coordinazione utile ai fini del corretto utilizzo di MVC, di separare il più possibile le tre componenti (Model, View e Controller) e di semplificare la sostituzione della parte di view, è stata creata una factory principale (ClassFactory) che si occupa di istanziare una classe che implementa una determinata interfaccia. Controller e view hanno le loro factory (ControllerFactory, ViewFactory), che permettono rispettivamente di instanziare facilmente l'implementazione di una interfaccia di un controller e di una view. Nel caso della factory del controller, è possibile anche specificare l'interfaccia della view, in modo da istanziarla tramite la factory della view e collegarla al controller istanziato.

Problema FXML fornisce un controller per la view che però dipende da JavaFX. Non è quindi possibile utilizzare questo controller con una view sostituita che utilizza, ad esempio, un altro framework per costruire l'interfaccia grafica.

Soluzione Il controller FXML viene utilizzato nel progetto come classe che identifica la view che dialoga con il controller principale dell'applicazione, in modo da utilizzare un controller che non dipenda dal framework della UI.

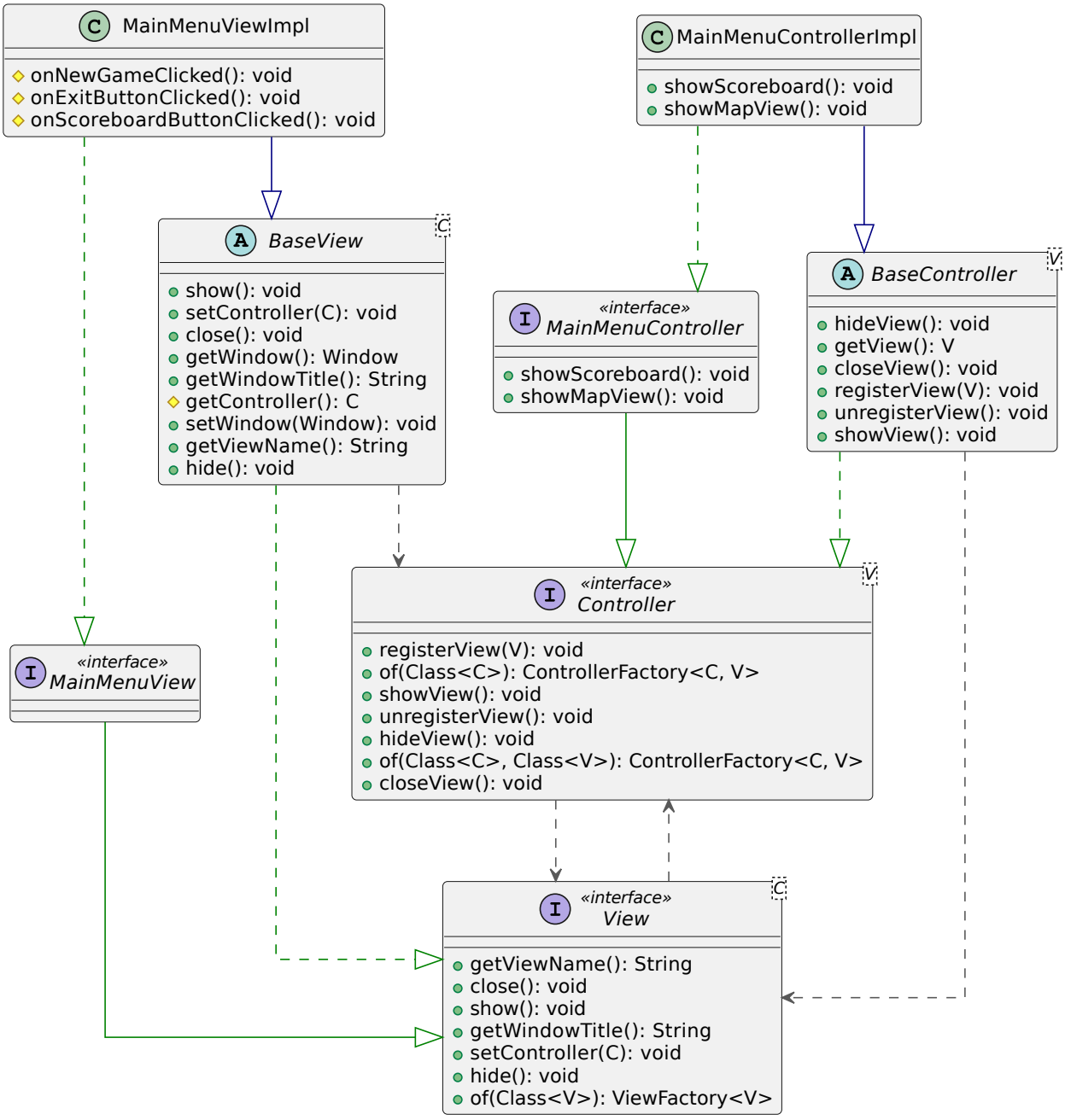

Figura 2.2: Schema UML del menu principale (view e controller). Il controller FXML (chiamato MainMenuViewImpl) delega le azioni da eseguire al controller dell'applicazione (chiamato MainMenuController)

Per la gestione delle scene/schermate mi sono affidato alla libreria SceneOneFX (vedi Note di sviluppo), integrata nella mia classe "helper" Win-

dow. Questa classe permette di interfacciarsi più ad alto livello con SceneOneFX e, soprattutto, indirettamente, per non rischiare di utilizzare questa libreria nel Model o nel Controller, in quanto è possibile utilizzarla correttamente solo nelle view collegate ad essa o a SceneOneFX.

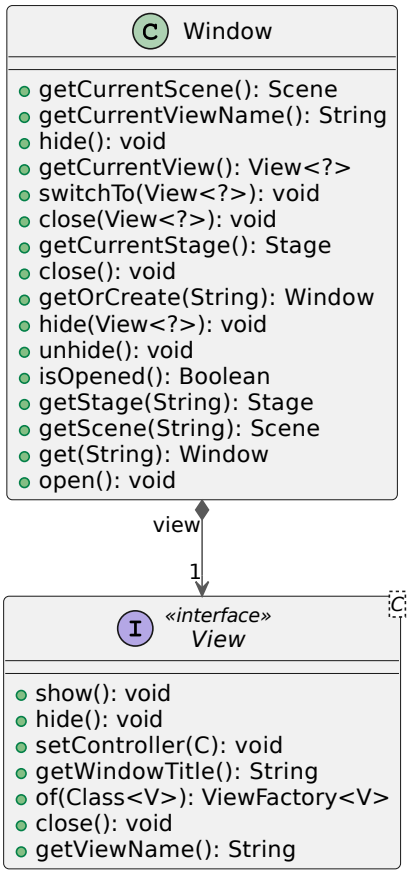

Figura 2.3: Schema UML della classe Window e del collegamento con l'interfaccia View

Simone Redighieri

La mia parte del progetto consisteva nell'implementazione delle varie entità presenti all'interno del gioco per poi concentrarmi nello specifico sullo sviluppo del giocatore, la gestione degli input da tastiera per il movimento e l'attacco del personaggio controllato dall'utente e la gestione delle collisioni con le entità presenti e la finestra di gioco. Avendo notato che tutte le entità (ostacoli e personaggi) hanno caratteristiche in comune, o meglio tutte le caratteristiche delle entità base sono possedute anche dai vari personaggi,

ho deciso di creare un'interfaccia Entity coi metodi principali, come la posizione e la dimensione, per massimizzare il riuso del codice, la quale sarà implementata dalla classe EntityImpl. Inoltre ho deciso di creare delle interfacce per ogni caratteristica di un personaggio (MovableEntity, LifeEntity, AttackEntity) che estendono tutte Entity, in modo da poterle implementare separatamente e a proprio piacimento in futuro per la creazione di nuove entità, e un'interfaccia Character che le estende tutte in quanto un personaggio è un'entità con una salute, in grado di muoversi e di attaccare. L'intefaccia Character viene implementata da una classe astratta BaseCharacter che estende EntityImpl e utilizza il pattern Template Method, ovvero implementa tutti i metodi in comune tra giocatore e nemici, lasciando alle sottoclassi il compito di implementare i metodi rimanenti.

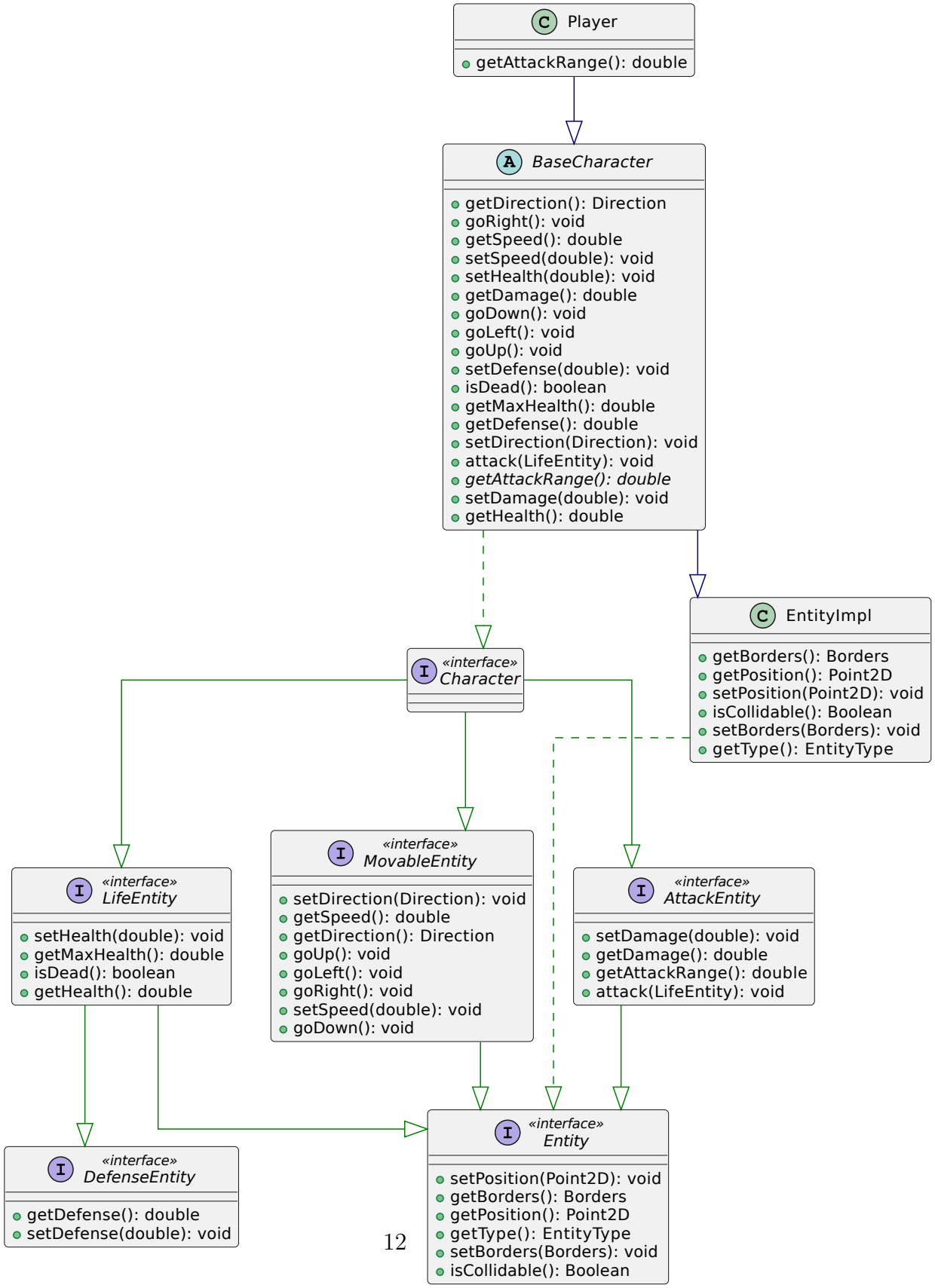

Figura 2.4: Schema UML del Model delle entità e del Player

Per il giocatore ho seguito il pattern MVC. Per il Model è stata creata la classe Player che estende la classe astratta BaseCharacter e si limita ad implementare solo il metodo template *getAttackRange()*. Per quanto riguarda la View ho deciso di creare una classe EntityView per le entità base e una classe CharacterView per i personaggi che implementa l'interfaccia AnimatedEntityView ed estende EntityView. Dato che i metodi per l'aggiornamento dell'immagine erano comuni sia per il giocatore sia per i nemici, ho deciso di impostare questa classe come astratta in modo da non poter essere istanziata e per limitare la duplicazione del codice. La classe PlayerView estende la classe astratta CharacterView e imposta solamente le immagini per il giocatore.

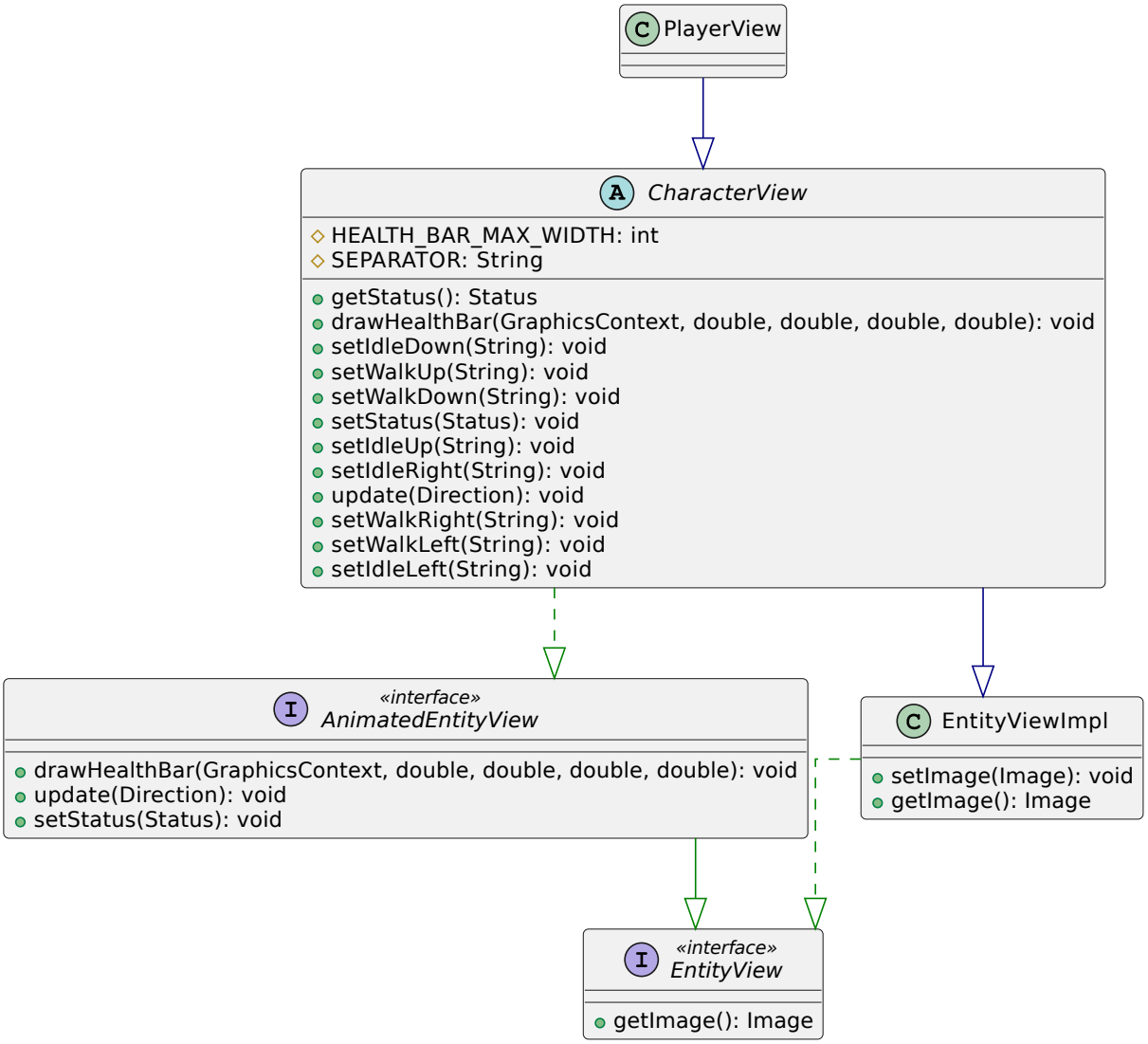

Figura 2.5: Schema UML della View delle entità e del Player

Il controller è stato implementato basandosi sul pattern Template Method poichè ho notato che i metodi per il movimento possono semplicemente chiamare un metodo astratto e protetto *move(Direction d)* che viene implementato dalle sottoclassi. Ho creato un controller per le entità base il quale viene esteso da un controller astratto AbstractController, che implementa l'interfaccia CharacterController, e che a sua volta verrà esteso dal PlayerController implementando il metodo template.

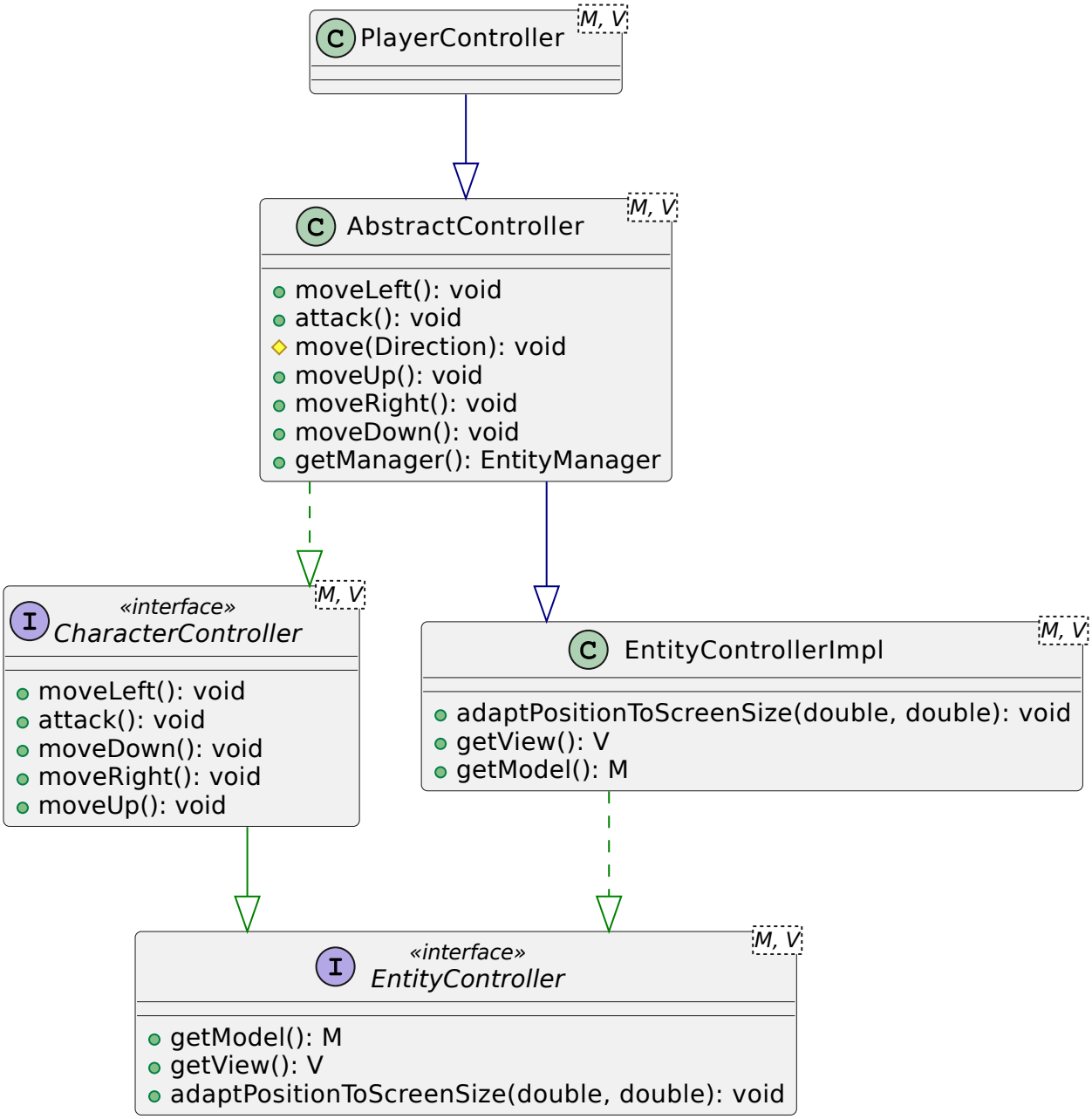

Figura 2.6: Schema UML del Controller delle entità e del Player

Il controller ha il compito di coordinare il model e la view, controllando i movimenti e le azioni e aggiornando l'immagine delle entità di volta in volta.

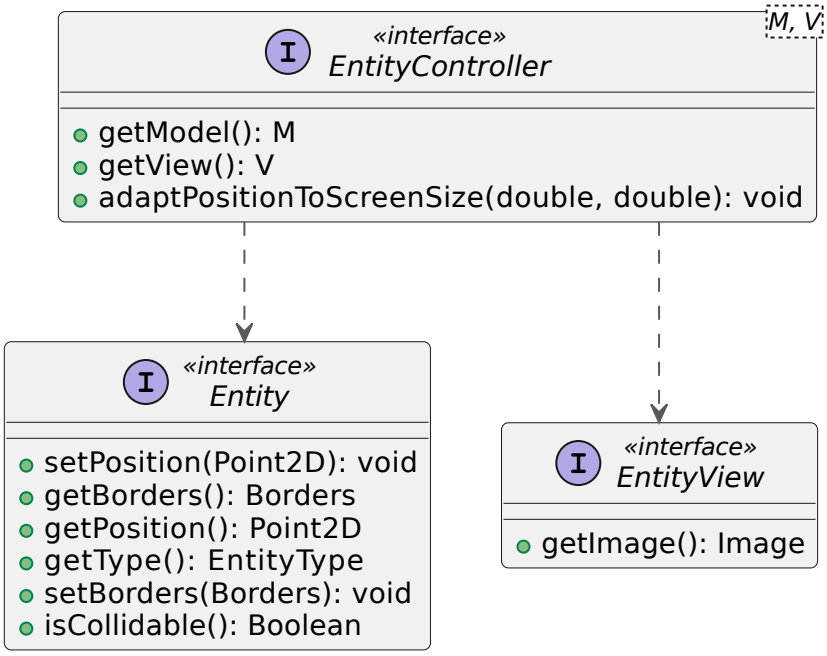

Figura 2.7: Rappresentazione UML del pattern MVC per le entità

Per mantenere salvate tutte le entità presenti nel gioco, è stato creato un EntityManager. Appena un'entità viene creata, questa deve essere aggiunta nella lista delle entità mantenuta nell'EntityManager, e ogni volta che un'entità risulta "sconfitta" e deve essere rimossa dal gioco, bisogna aggiornare la lista rimuovendo l'entità desiderata. L'EntityManager mantiene un riferimento del CollisionManager, il quale ha il compito di controllare le collisioni tra entità e con la finestra del gioco. Per questo motivo tutti i personaggi devono mantenere un riferimento dell'entity manager per controllare le collisioni ad ogni movimento.

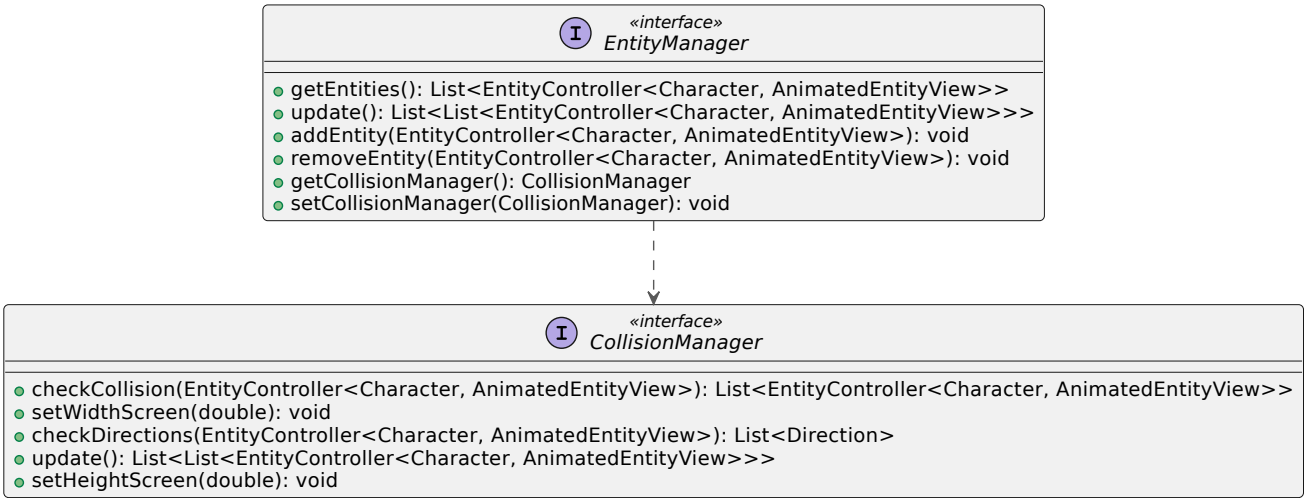

Figura 2.8: Schema UML della gestione delle entità e delle collisioni

Leonardo Viola

La mia parte di progetto consisteva nella creazione dei nemici e del mondo di gioco.

NEMICI

Per quanto riguarda i nemici, che sono delle entità, ho creato le classi Enemy, EnemyView ed EnemyController in modo tale da rispettare il pattern MVC, per la realizzazione di queste classi ho esteso delle classi realizzate da Simone e usate anche per la creazione del player. A differenza del player che si muove grazie all'input dell'utente, i nemici si muovono autonomamente, essendo questa una funzionalità che deve appartiene al model l'ho aggiunta alla classe Enemy. Per creare e gestire tutti i nemici viene utilizzata la classe EnemiesController, che ha il compito di creare i nemici e di gestirli, e mette a disposizione dei metodi che si occupano sia della parte di view che del model di tutti i nemici.

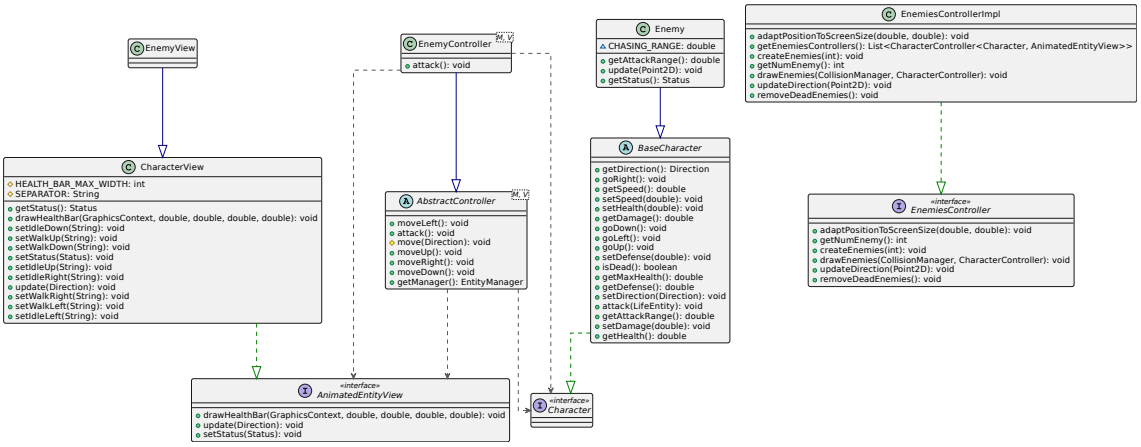

Figura 2.9: Schema UML della gestione dei nemici

MONDO DI GIOCO

Per il mondo di gioco utilizzo il file map.fxml per creare la struttura della finestra che è composta principalmente da un canvas, e poi attraverso le classi che ho realizzato disegno delle immagini nel canvas per mostrare il mondo di gioco e le diverse entità (player e nemici). Il mondo di gioco è costituito da tante immagini, ogni immagine rappresenta un componente del mondo di gioco (es: albero, prato, muro, ecc...), ovviamente solo alcuni di questi componenti saranno visti come degli ostacoli (il prato non sarà un ostacolo perché può essere calpestato dai nemici e dal player, invece un albero è un ostacolo). Anche in questo caso ho usato il pattern MVC, in particolare per quanto riguarda la parte di view uso le classi CrossableTile e SolidTile, per differenziare le tile (immagini che compongono il mondo di gioco) calpestabili da quelle non calpestabili. Nel model ho deciso di tenere traccia solo delle SolidTile per questo ho creato la classe ObstacleEntity, che terrà traccia della posizione e delle dimensioni degli ostacoli presenti nel mondo di gioco. Quindi nella parte di view tengo traccia di tutte le tile usate nel mondo di gioco (calpestabili e non), invece nella parte di model ho tenuto traccia solo delle tiles non calpestabili (che rappresentano gli ostacoli), ho fatto questa scelta perché non ho bisogno di sapere posizione e dimensione delle tiles che non rappresentano ostacoli. Ho usato una classe SpwanerImpl che serve a "spawnare", cioè a far comparire nel mondo di gioco degli ostacoli, infatti questa classe viene usata per lo spawn degli alberi nella mappa, ed ogni volta che il gioco viene avviato, la posizione degli alberi sarà diversa.

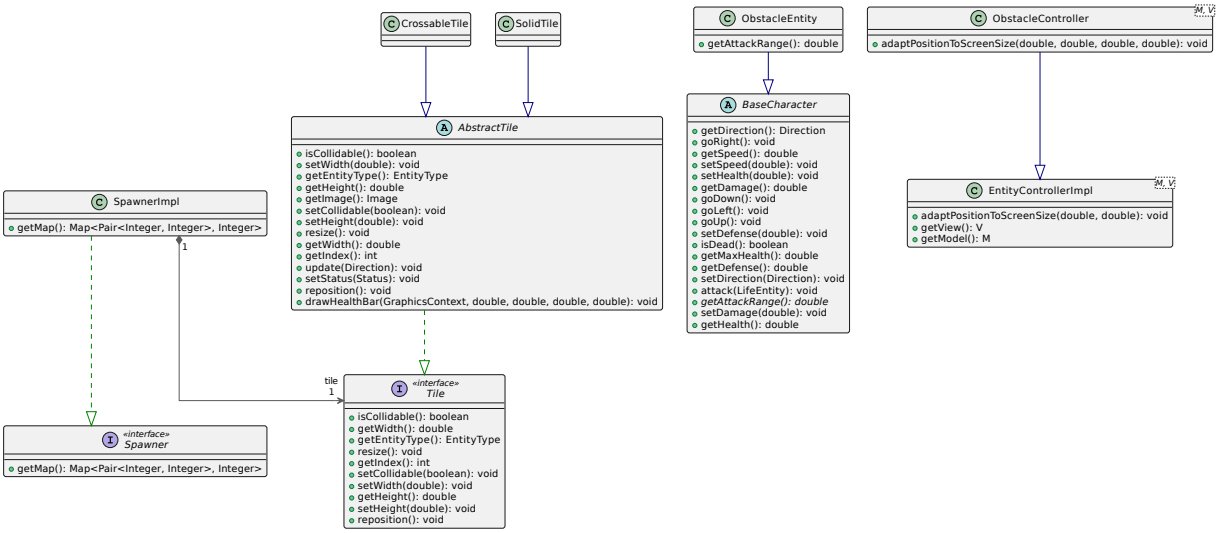

Figura 2.10: Schema UML del pattern MVC della mappa

Dopo aver parlato della parte di model e di view del mondo parliamo della parte di controller, che è composta da due classi: **MapViewimpl** che sarebbe il controller del file map.fxml e nel pattern MVC appartiene alla parte di View, e possiamo definirlo come un controller della view, e **MapControllerImpl** che è il vero controller. La differenza sostanziale tra queste due classi è che MapViewImpl ha una visione solo sulla view ed utilizza metodi e componenti della libreria javafx, mentre ciò non avviene in MapControllerImpl (perchè ho separato la parte di view da quella di controller) e a sua volta ha una visione più globale del mondo di gioco. In MapViewImpl è presente il gameloop perchè è realizzato usando una classe di javafx ed è presente anche un Thread che si occupa di effettuare altre azioni in contemporanea al gameloop.

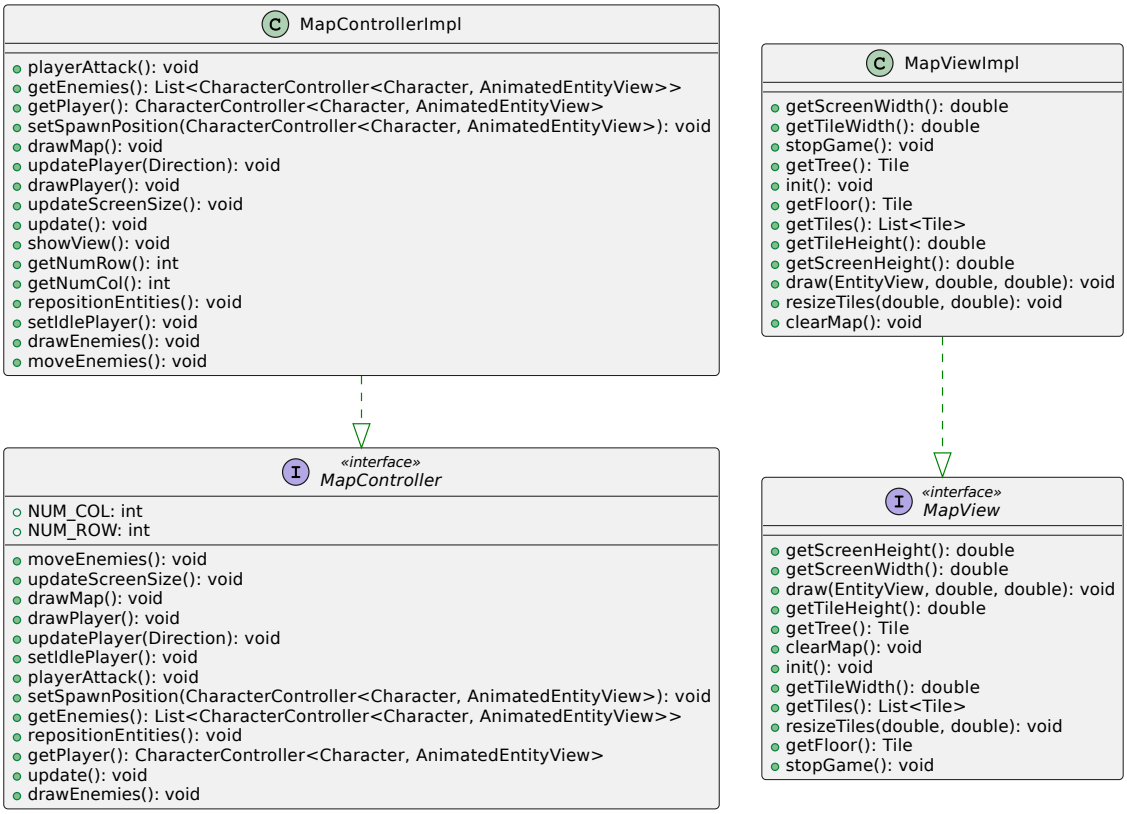

Figura 2.11: Schema UML della gestione della mappa

Sia per la realizzazione dei nemici che per la realizzazione del mondo di gioco ho cercato di separare al meglio le parti di model, di view e di controller. Inoltre ho fatto molta attenzione a non usare classi e metodi di javafx nel model e nel controller.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

I test automatici sono stati sviluppati con JUnit 5 e AssertJ, che fornisce delle "Assertions" più OOP-oriented di quelle di JUnit. I test riguardano la maggior parte dei componenti del progetto, tra cui tutte le view, i model, le factory. Sono stati effettuati test manuali sui 3 principali sistemi operativi presenti sul mercato (Windows, Linux (in particolare Ubuntu) e MacOS), così come avviati anche i test automatici su questi ultimi al fine di verificare la compatibilità del progetto con tutti i sistemi. Per il testing dell'interfaccia grafica ci si è affidati a TextFX per rendere il tutto automatico, sia su sistemi con monitor, sia su sistemi "headless" (ad esempio le Github Actions, che abbiamo utilizzato sul nostro repository Github per eseguire i test ad ogni push) che non possiedono un monitor collegato. *Si noti che, a causa di limitazioni del sistema headless di Github actions, è normale che alcuni test riguardanti la GUI di JavaFX falliscano frequentemente, cosa che invece non accade se eseguiti su un sistema headed (con monitor) e che il testing su MacOS è limitato in quanto nessun componente del gruppo possiede un Mac per poter effettuare i test (si è pertanto provato i test su macchina virtuale, che però potrebbero non risultare reali).*

3.2 Metodologia di lavoro

Ci aspettiamo, leggendo questa sezione, di trovare conferma alla divisione operata nella sezione del design di dettaglio, e di capire come è stato svolto il lavoro di integrazione. **Andrà realizzata una sotto-sezione separata per ciascuno studente** che identifichi le porzioni di progetto sviluppate, separando quelle svolte in autonomia da quelle sviluppate in collaborazio-

ne. Diversamente dalla sezione di design, in questa è consentito elencare package/classi, se lo studente ritiene sia il modo più efficace di convogliare l'informazione. Si ricorda che l'impegno deve giustificare circa 40-50 ore di sviluppo (è normale e fisiologico che approssimativamente la metà del tempo sia impiegata in analisi e progettazione).

Maicol Battistini

La parte di lavoro a me assegnata riguardava la creazione del menù principale, la gestione del punteggio (chiamato nel gioco "Scoreboard") in tutte le sue parti (Model, View e controller), la creazione della schermata di Gioco finito, l'aggiunta dell'HUD in-game (ovvero la barra della vita di una entità (ad esempio giocatore o nemico)) e l'aggiunta della statistica di difesa alle entità che hanno una vita (quindi un personaggio).
L'architettura è stata inizialmente stabilita insieme a tutto il gruppo, gettando le basi per il progetto finale. Verso la fine del progetto poi, sono state unite le modifiche di ognuno tramite Git, avendo lavorato su branch differenti. Una volta unite le modifiche sono stati effettuati i ritocchi finali (fix dei warning dei linter, quali Checkstyle, PMD e Spotbugs, test finali per verificare il corretto funzionamento del progetto su tutti i dispositivi).

Simone Redighieri

La parte di lavoro a me assegnata consisteva nell'implementazione del Player, ovvero il personaggio principale controllato dall'utente, la gestione degli input da tastiera per il movimento e l'attacco, e la gestione delle collisioni. La fase di progettazione iniziale è stata un po' complessa inizialmente, principalmente perchè non riuscivo bene a capire come strutturare al meglio il lavoro e separare le parti di model da quelle di view. In fase di implementazione, invece, ho riscontrato qualche problema nel comprendere bene il corretto funzionamento di JavaFX, che sono riuscito a risolvere servendomi della documentazione, e soprattutto nello sviluppo della view del giocatore, nello specifico il cambio immagine e l'animazione del player secondo la direzione in cui si muoveva. L'utilizzo di DVCS GIT è stato fondamentale poichè ci ha permesso di lavorare separatamente e autonomamente sulle parti assegnate e di monitorare lo sviluppo del lavoro degli altri componenti del team. Abbiamo deciso di procedere creando delle proprie branch, in cui fare il push delle modifiche necessarie al corretto funzionamento dell'applicazione, le quali sono state unite successivamente in un unico branch principale.

Leonardo Viola

La mia parte di lavoro iniziale comprendeva lo sviluppo del nemico e di altre classi da sviluppare insieme a Simone che sarebbero state utilizzate sia per il nemico che per il player, a meno di un mese dalla scadenza dopo l'uscita di Elia dal gruppo è stata assegnata a me tutta la sua parte, ovvero la creazione del mondo di gioco, quindi alla fine mi sono occupato della gestione dei nemici e della creazione e gestione del mondo di gioco. Per quanto riguarda la parte dei nemici ho fatto grande uso delle altre classi sviluppate da Simone, invece per lo sviluppo e gestione del mondo di gioco ho fatto uso sia di classi sviluppate da Simone sia da Maicol, di grande aiuto è stato git che ci ha permesso di rendere più facile la nostra collaborazione e condivisione di codice. Dato che io mi sono occupato della gestione della mappa, ho dovuto sviluppare delle classi in cui vengono gestiti diversi aspetti, dalle entità alla barra della salute, ecc... , e per lo sviluppo di queste classi ho collaborato molto insieme agli altri, per capire bene come unire le nostre singole parti. Un problema che ho riscontrato nella mia parte di progetto causato dall'organizzazione della struttura del progetto, riguarda gli ostacoli, che sono entità che non si muovono (possono essere riposizionate in seguito al ridimensionamento della finestra) e non hanno vita, però nonostante ciò quando sono andato ad unire la mia parte di progetto con quella di Simone ho dovuto, per motivi di compatibilità con le classi e metodi da lui sviluppati, far estendere all'interfaccia Tile (interfaccia che rappresenta la parte di view degli elementi che compongono la mappa) l'interfaccia AnimatedEntityView che rappresenta la view di entità animate (e non è il caso delle entità che compongono la mappa), inoltre la classe ObstacleEntity (che rappresenta il model delle entità ostacoli) estende la classe BaseCharacter che gestisce entità che possono muoversi, attaccare, ecc..., quindi per questo motivo ad esempio nella classe ObstacleEntity è presente il metodo getAttackRange() che non dovrebbe esserci perchè gli ostacoli non attaccano, ed è sempre per questo motivo che nella classe AbstractTile sono presenti dei metodi vuoti che riguardano entità animate.

3.3 Note di sviluppo

Maicol Battistini

- **Progettazione di classi con generici**: utilizzati per collegare la tipizzazione di Controller e View (schermata della UI).

- **Lambda expressions**: Utilizzate nella view della scoreboard e nella ClassFactory
- **Stream**: Utilizzato nella ClassFactory per effettuare operazioni sugli elementi di un array
- **JavaFX**: Libreria utilizzata per realizzare la UI del progetto (buona parte della view)
- **MaterialFX**: Componenti "custom" per JavaFX che seguono lo stile Material Design di Google.
- **SceneOneFX**: Utility che semplifica la creazione e la gestione di scene di JavaFX. Utilizzato per gestire meglio la finestra di JavaFX e il passaggio da una scena/schermata all'altra.
- **Appdirs**: Utility per ottenere rapidamente i percorsi speciali del sistema (es. dati delle app) in modo da ottenere il percorso in cui salvare i dati persistenti (ad esempio scoreboard.json)
- **Classgraph**: Scannerizzatore del classpath. Utilizzato per trovare ed ottenere la classe che implementa una certa interfaccia nella ClassFactory.
- **Google Guava**: Utility varie per Java. Utilizzata nel caricamento del file FXML in modo dinamico: viene costruito il nome del file in base al nome della classe della view convertito in snake case grazie a Guava
- **JsonBeans**: Lettura e scrittura di file JSON. Utilizzata per scrivere facilmente le informazioni della scoreboard in un file JSON tramite serializzazione del model.
- **AssertJ**: Assertions per JUnit più OOP-oriented rispetto a quelle incluse in JUnit.
- **TextFX**: Adattamento dei test JUnit per poter testare gli elementi grafici di JavaFX. Utilizzato per effettuare test sulle interfacce grafiche create con JavaFX.
- **Monocle**: Binari precompilati di Monocle, una implementazione delle finestre e la loro gestioni per sistemi embedded (senza monitor). Utilizzato per poter avviare correttamente i test nelle Github Actions.
- **Github actions**: Utilizzate per testare (tramite JUnit), controllare i warning dei linter e creare un JAR ad ogni push del codice sul repo Github

Simone Redighieri

- **Progettazione di classi con generici**: utilizzati nei controller per avere view e model generici.
- **Stream**: utilizzate quando possibile per comodità e leggibilità.
- **Lambda expressions**: utilizzate principalmente negli Stream.
- **JavaFX**: libreria principale del progetto.
- **AssertJ**: utilizzato nei test con JUnit

Leonardo Viola

- **Progettazione di classi con generici**: utilizzati sopratutto per usare le classi sviluppate da Simone.
- **Stream**: utilizzate sopratutto nei controller per gestire le diverse entità.
- **Lambda expressions**.
- **JavaFX**: libreria principale del progetto.
- **AssertJ**: utilizzato nei test con JUnit

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Maicol Battistini

Mi ritengo soddisfatto della parte da me svolta, soprattutto della parte di comunicazione tra view e controller, pur essendo separati a livello implementativo. Questo progetto mi ha permesso di migliorare le mie capacità di lavoro in gruppo e di aumentare la mia conoscenza del mondo Java, in particolare della vastità di librerie esistenti online. Ho avuto un pò di difficoltà soprattutto nel coordinare i lavori durante la fase finale del progetto, dove sono saltati fuori i bug e i problemi più grandi e importanti.

Simone Redighieri

Nonostante le difficoltà riscontrate inizialmente mi ritengo soddisfatto del lavoro svolto. Prima di iniziare a lavorare concretamente al progetto, mi sentivo in parte insicuro, poichè non avevo mai affrontato prima un progetto di tali dimensioni. Grazie al tempo impiegato, al confronto e scambio di idee con i membri del gruppo, ho iniziato a prendere confidenza con il lavoro. La realizzazione di un progetto così complesso sia per il coordinamento in gruppo sia per il seguito pratico di realizzazione, credo sia stata un'esperienza utile e formativa, perchè mi ha permesso di sperimentare a lungo le mie competenze, acquisendo consapevolezze nuove.

Leonardo Viola

All'inizio ero un po' titubante all'idea di dover sviluppare un progetto con altri, perché fino ad ora ero abituato a lavorare da solo, ma comunque era

un qualcosa che volevo fare perché sapevo che mi sarebbe stato utile per il futuro. Durante lo sviluppo del progetto ho incontrato diverse difficoltà, all'inizio non sapevamo bene cosa fare, ed eravamo un pò spaesati, poi siamo riusciti ad organizzarci e alla fine c'è l'abbiamo fatta. Inizialmente la mia parte del progetto era diversa da quella di cui poi mi sono occupato, infatti ad un mese dalla consegna dopo l'uscita dal gruppo di Elia a me è stata assegnata la parte di creazione del mondo, e ho incontrato delle difficoltà perché nei mesi precedenti non mi ero mai focalizzato sulla parte di view e di creazione del mondo, ma mi ero focalizzato di più sulla parte dei model e controller, quindi nonostante mi sia stata assegnata una parte di cui non conoscevo il funzionamento, alla fine ce l'ho fatta anche se a causa mia abbiamo consegnato in ritardo, e mi scuso per questo, ma nell'ultimo periodo prima della consegna ho avuto dei problemi che non erano previsti, e che non c'entravano nulla con il progetto. Alla fine sono felice di come è andata anche perché ho utilizzato git, che ho scoperto essere un qualcosa di molto utile per chi programma.

4.2 Difficoltà incontrate e commenti per i docenti

Le maggiori difficoltà incontrate dal team di sviluppo sono state:

- La suddivisione del lavoro, in quanto cercando di separare il più possibile le parti assegnate, cercando di dare ad ognuno una parte autonoma, si ricadeva spesso in parti comuni.
- Coordinazione nello sviluppo delle parti in comune.
- Realizzazione di una struttura del progetto adeguata per seguire il pattern MVC
- Ritardo nella consegna causato dalla parte assegnata a Viola, siccome è stata completata in ritardo di un pò di giorni rispetto alla deadline.

Appendice A

Guida utente

All'avvio del gioco viene presentato il menu principale, in cui è possibile avviare una nuova partita, aprire la scoreboard (tabella punteggi dei giocatorI) o uscire dal gioco. All'avvio di una nuova partita si controllerà un cavaliere all'intereno di una mappa intento a decimare gli zombie nemici. È possibile utilizzare i seguenti comandi da tastiera:

- **Tasto D**: muove il giocatore a destra.
- **Tasto A**: muove il giocatore a sinistra.
- **Tasto W**: muove il giocatore verso l'alto.
- **Tasto S**: muove il giocatore verso il basso.
- **tasto SPACE**: il giocatore attacca davanti a sé.

A fine gioco (quando il giocatore viene sconfitto o tutti i nemici vengono sconfitti vengono presentati i punti totalizzati, con la possibilità di salvarli nella scoreboard inserendo il proprio nome e premendo il pulsante "SALVA". La restante parte della finestra è identica al menu principale, con l'eccezione che il pulsante ESCI è sostituito dal pulsante "MENU PRINCIPALE" che permette di ritornare al menu principale del gioco.

Infine, la finestra della scoreboard permette di visualizzare in una tabella tutti i punteggi ottenuti dai giocatori e di filtrarli.

Appendice B

Esercitazioni di laboratorio

B.0.1 Maicol Battistini

- Laboratorio 04: https://virtuale.unibo.it/mod/forum/discuss.php?d=12345#p123456
- Laboratorio 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=90125#p141139
- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=87880#p138075
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=88829#p138965
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=89272#p140290
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=90125#p141139
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=91128#p141948