Progetto di Programmazione ad Oggetti
"The Exiled"

Luca Casadei, Francesco Pazzaglia, Marco Magnani, Manuel Baldoni

18 febbraio 2024

Indice

Capitolo 1

Analisi

1.1 Requisiti

L'applicazione è un gioco che presenta un personaggio controllato dal giocatore con la possibilità di muoversi nella mappa in 4 direzioni e di combattere contro dei nemici utilizzando magie elementali. I nemici sconfitti potrebbero rilasciare delle cure o dei potenziamenti che favoriscono il giocatore. Per concludere il gioco, il giocatore deve entrare in possesso di 4 cristalli che vengono consegnati una volta sconfitti i 4 boss del gioco che sono nemici più difficili da sconfiggere.

Requisiti funzionali

- Movimento del giocatore.
- Movimento dei nemici.
- Presenza di oggetti ottenibili dal giocatore (Cure, potenziamenti e cristalli).
- Terminazione del gioco una volta raccolti 4 cristalli o se il giocatore viene sconfitto.
- Rilascio degli oggetti da parte dei nemici.
- Possibilità di fare battaglie tra giocatore e nemici.
- Nemici più forti (boss) che se sconfitti rilasciano i cristalli per terminare il gioco.

- Possibilità di utilizzo di magie in battaglia di diverso tipo (Fuoco, Acqua, Fulmine, Erba).
- Aumento di livello tramite guadagno di esperienza sconfiggendo i nemici.

Requisiti non funzionali

- Il gioco deve essere multipiattaforma.
- L'interfaccia deve essere ridimensionabile.

1.2 Analisi e modello del dominio

1.2.1 Inizio del gioco

All'inizio al giocatore viene chiesto di che tipo elementale sarà il personaggio da controllare (vedi *Elenco dei tipi elementali*), quando verrà posizionato sulla mappa gli verrà assegnata una mossa generica di tipo normale, quelle invece del suo stesso tipo potranno essere imparate in maniera automatica all'aumentare di livello.

1.2.2 Movimento

Il giocatore può muoversi di una cella alla volta in una mappa a griglia in 4 direzioni (Nord, Est, Sud, Ovest), mentre si muove il giocatore, si muovono sempre di cella in cella anche i nemici, che non seguono un pattern di movimento specifico ma seguono una direzione casuale, questo a meno che non sia presente il giocatore nel loro range di visibilità (Configurabile da costante), in tal caso seguiranno il giocatore.
Il movimento del giocatore è libero su tutta la mappa, mentre i nemici sono costretti a stare nella zona del loro stesso tipo e non ne potranno uscire anche se in fase di inseguimento del player.

1.2.3 Battaglia

Una battaglia inizia se il giocatore e un nemico si ritrovano nella stessa cella a seguito di un movimento. Per affrontare un nemico in battaglia il giocatore ha a disposizione massimo 4 mosse (non per forza dello stesso tipo del giocatore).

Il combattimento avviene a turni alternati, il primo è il giocatore, poi il nemico e così via finché uno dei due viene sconfitto, se è il giocatore, il gioco termina.
Allo sconfiggere dei nemici viene conferita al giocatore una certa quantità di esperienza (a seconda del nemico sconfitto) che serve ad aumentare di livello e vi è una certa probabilità che il nemico possa rilasciare un oggetto casuale (la probabilità maggiore è però quella di non ricevere nulla).

1.2.4 Oggetti

Qualora venga rilasciato, l'oggetto verrà automaticamente messo nell'inventario del giocatore e potrà essere usato direttamente dalla mappa oppure in battaglia. Ci sono diversi tipi di oggetto, quelli utilizzabili dalla mappa generalmente sono oggetti di cura (rappresentati con il colore verde), altri possono essere oggetti che aumentano il danno del giocatore (colore rosso) oppure la difesa (colore blu-azzurro). Nessun nemico generico potrà rilasciare un *Cristallo*, che è un oggetto esclusivo dei boss e serve per concludere il gioco.

1.2.5 Livello ed esperienza

L'aumento di livello comporta un incremento generale delle statistiche del giocatore, ovvero attacco, difesa e vita di un valore moltiplicativo o incrementale costante. Ogni n livelli verrà presentata al giocatore la possibilità di imparare una nuova mossa, dove n è una costante configurabile. se il giocatore ha già 4 mosse potrà decidere di scambiare quella nuova con una di quelle che conosce già. All'aumentare di livello sarà richiesta sempre più esperienza per passare a quello successivo.

1.2.6 Nemici

I nemici si dividono in nemici generici, ovvero quelli presenti sulla mappa con capacità di movimento, questi possono essere anche ripetuti e una volta sconfitti non ricompaiono. Ciascuno di questi ha un suo tipo elementale e quindi un'area della mappa a cui è confinato. Per vedere i possibili oggetti rilasciati dai nemici, vedi *Oggetti*.
I 4 boss sono posizionati agli angoli della mappa, e ciascuno di loro si trova nella parte di mappa che corrisponde al tipo elementale che rappresentano, i boss non hanno capacità di movimento e staranno in quella posizione finché non verranno sconfitti dal giocatore.

1.2.7 Elenco dei tipi elementali

I seguenti sono i tipi elementali presenti, ognuno è efficace rispetto ad un altro, quindi quando vengono usate mosse di un certo tipo risultante essere efficace rispetto al tipo di un nemico (non delle mosse che ha, proprio il tipo del nemico), si avranno dei moltiplicatori del danno. Lo stesso vale per i nemici che usano mosse di un tipo efficace rispetto a quello scelto dal giocatore all'inizio (la sua classe, vedi *Inizio del gioco*).
Ad esempio, nel seguente schema si vede che Fulmine è efficace contro Acqua, Erba contro Fulmine etc... Se si capovolge la freccia si notano invece le debolezze di un tipo rispetto ad un altro.

Fulmine	$\longrightarrow$	Acqua
Erba	$\longrightarrow$	Fulmine
Acqua	$\longrightarrow$	Fuoco
Fuoco	$\longrightarrow$	Erba

1.2.8 UML

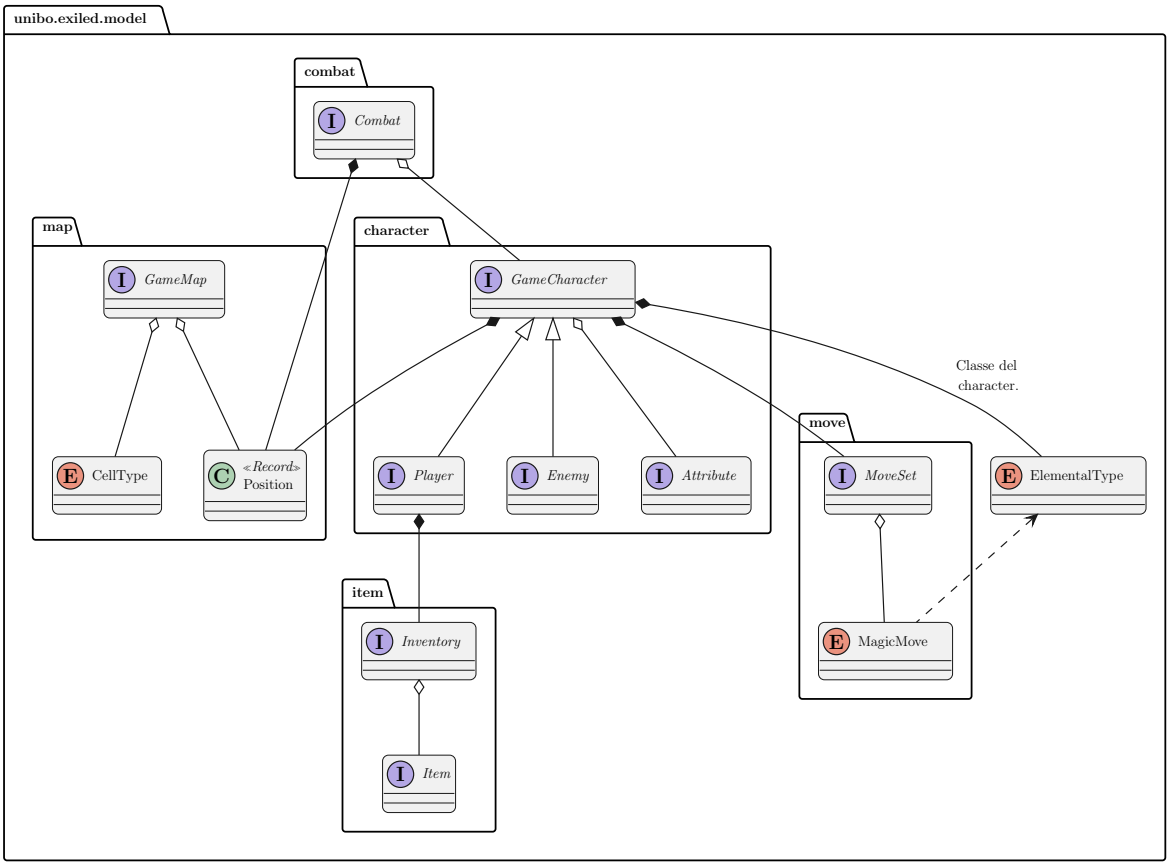

Figura 1.1: Schema UML del dominio.

Capitolo 2

Design

Qui entriamo nel dettaglio della struttura ad alto livello del gioco.

2.1 Architettura

2.1.1 Model entry point

La struttura del progetto segue il modello MVC, questa divisione ha come entry point del model la classe *GameModelImpl* che implementa *GameModel*, essa ha il ruolo di dare accesso agli altri componenti del model, tra cui:

- *CharacterModel*: Model che gestisce il player, nemici e relativi attributi.
- *CombatModel*: Fornisce la gestione del combattimento tra player e nemico.
- *ItemsModel*: Fornisce la gestione degli item del gioco.
- *MapModel*: Fornisce la gestione della mappa e le relative celle.
- *MenuModel*: Fornisce la gestione interna del menu di gioco.

2.1.2 Controller entry point

A questo punto di accesso si collega il controller, che è composto anch'esso da un punto di accesso principale che è *GameController*, questo fornisce accesso alle seguenti sottoparti del controller:

- *CharacterController*: Gestisce il collegamento tra view e model per movimento azioni compibili dalle entità del gioco (nemici e giocatore).

- *CombatController*: Collega la view con il model per il combattimento.
- *ItemsController*: Collega la view con il model per la gestione degli oggetti dell'inventario.
- *MapController*: Collega la view con il model gestendo la mappa e le celle.
- *MenuController*: Collega view e model per la gestione del menu di gioco.

2.1.3 Relazione tra View e Controller

La view viene suddivisa in diverse sottoparti come le precedenti componenti architetturali, l'entry point è la classe *GameView* che è quella che fa partire il motore grafico del gioco. Le sottoparti sono:

- *CharacterView*: Visualizzazione dei nemici e del giocatore.
- *CombatView*: Visualizzazione del combattimento tra un giocatore e un nemico.
- *InventoryView*: Visualizzazione dell'inventario e uso degli item.
- *PlayerClassView*: Visualizzazione iniziale per la scelta della classe del giocatore tra gli elementi disponibili.
- *GameOverView*: Schermata di Game Over.
- *GameVictoryView*: Schermata di vittoria del gioco.
- *NewGameView*: Schermata di avvio.

Alcuni elementi grafici sono stati realizzati con apposite classi di view che hanno quindi un comportamento molto specifico e non comportano necessariamente un collegamento con il controller, ad esempio: *HUDView*, *GameGridView*, *GameMoveChangeView*.

2.1.4 Schema MVC

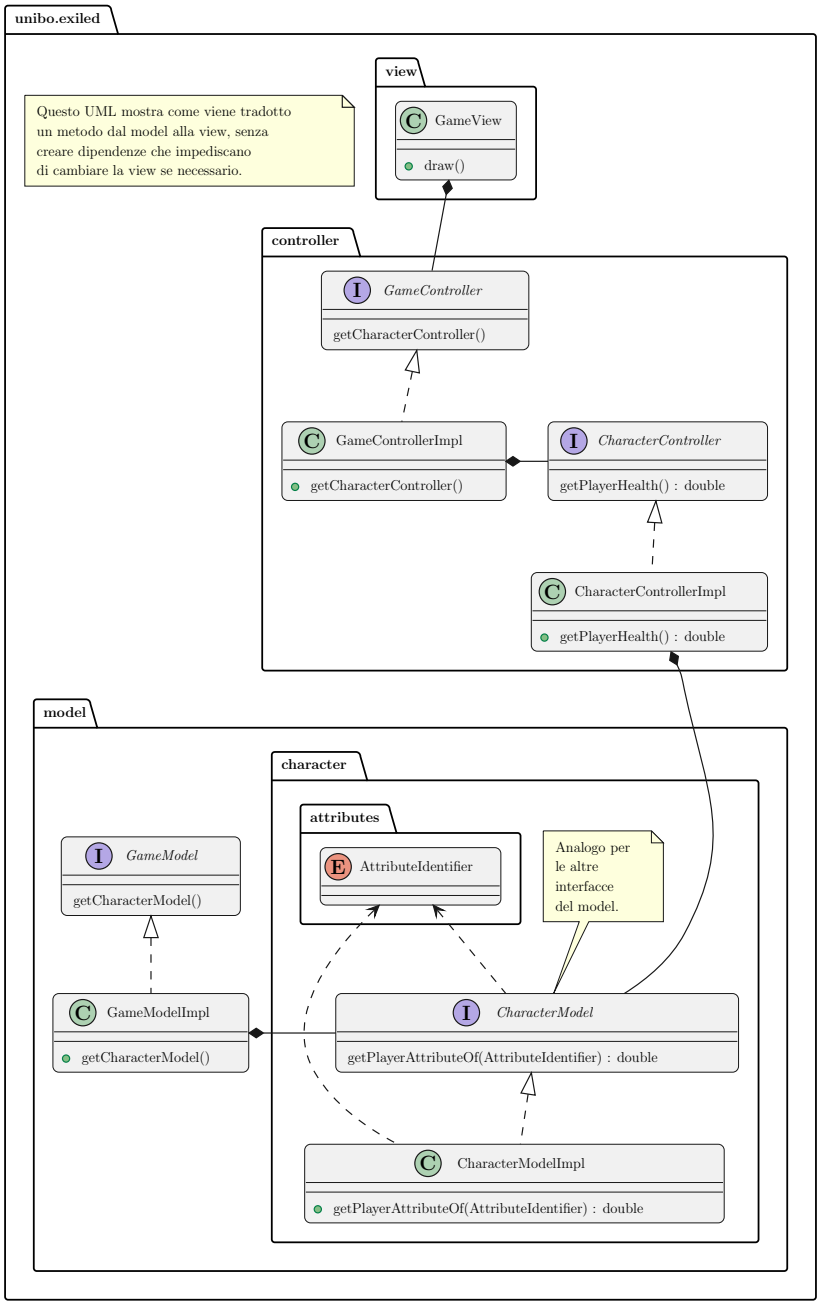

Figura 2.1: Schema UML che rappresenta l'architettura MVC

In conclusione, anche rimpiazzando totalmente il framework di Swing per la view con un altro (es. JavaFX) non è necessario fare alcun tipo di alterazione

al controller e tantomeno al model.

2.2 Design dettagliato

2.2.1 Luca Casadei

Personalmente mi sono occupato di parte della realizzazione delle mosse, della realizzazione di parte dell'entità *GameCharacter*, della realizzazione dei nemici e della loro distribuzione sulla mappa in base al tipo, della creazione della mappa e relativa suddivisione in aree, della creazione ed assegnazione degli attributi dei *GameCharacter*. Per ciascuno di questi elementi del model è stato realizzato un controller apposito per poter utilizzare il pattern MVC.

Entità "Nemico"

Problema: Definire il concetto di nemico e come omologarlo al giocatore per evitare ripetizioni di codice (DRY). Questa difficoltà è riassumibile nel dover suddividere i nemici in comuni e boss finali, divisi ciascuno in base al proprio "tipo elementale" e con la propria quantità di esperienza rilasciata alla sconfitta. Ogni nemico deve inoltre avere un set di mosse personalizzato che varia anch'esso in base al tipo.
Soluzione: La soluzione adottata è stata quella di estendere un concetto di *GameCharacter*, che contenesse tutte le informazioni e le azioni condivise tra giocatore e nemici, e successivamente effettuare la relativa divisione costruendo due interfacce diverse, una per il giocatore e una per il nemico. Per quest'ultimo è stata realizzata un'apposita interfaccia implementata da una classe astratta, che poi verrà utilizzata attraverso un pattern creazionale descritto successivamente per generare e distribuire i nemici sulla mappa. I nemici vengono infine divisi ulteriormente in generici e boss.

UML dell'entità "Nemico"

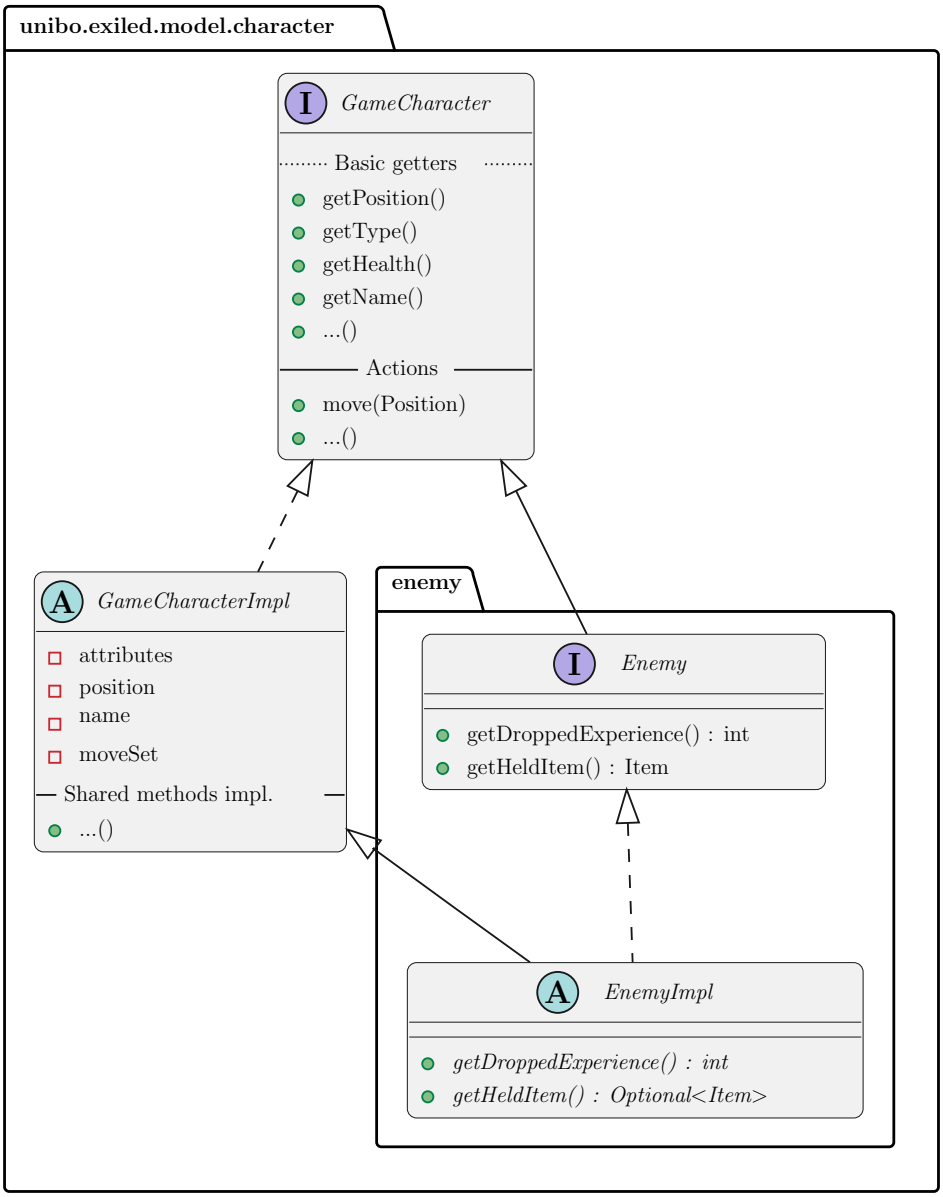

Figura 2.2: Schema UML che rappresenta la gestione dell'entità "Nemico"

La classe *EnemyImpl* contiene metodi astratti in modo tale da poter gestire la singola implementazione a seconda del diverso nemico del gioco, come spiegato successivamente.

Creazione dei nemici

Problema: Generare i nemici in modo scalabile (l'aggiunta di un nuovo nemico al gioco deve essere meno complesso e tedioso possibile).
Soluzione: Come introdotto precedentemente, il nemico è rappresentato da una classe astratta *EnemyImpl* che viene sfruttata attraverso l'uso del pattern *Factory Method*. Ci sono diversi tipi di nemico nel gioco, quelli generici e i Boss. La classe che fa uso di questo pattern è *EnemyFactory* e la relativa implementazione *EnemyFactoryImpl*.
Alternative: Un'alternativa possibile era quella di creare una classe che ereditasse da *EnemyImpl* per ogni nemico presente nel gioco ed assegnarli manualmente le mosse, l'esperienza rilasciata ecc..., in questo caso il codice sarebbe risultato meno scalabile e avrebbe reso l'aggiunta di nemici tediosa.
Segue uno schema UML che rappresenta il metodo generazionale dei nemici (Esclusa la divisione tra nemici generici e Boss che viene trattata separatamente in una sottosezione successiva).

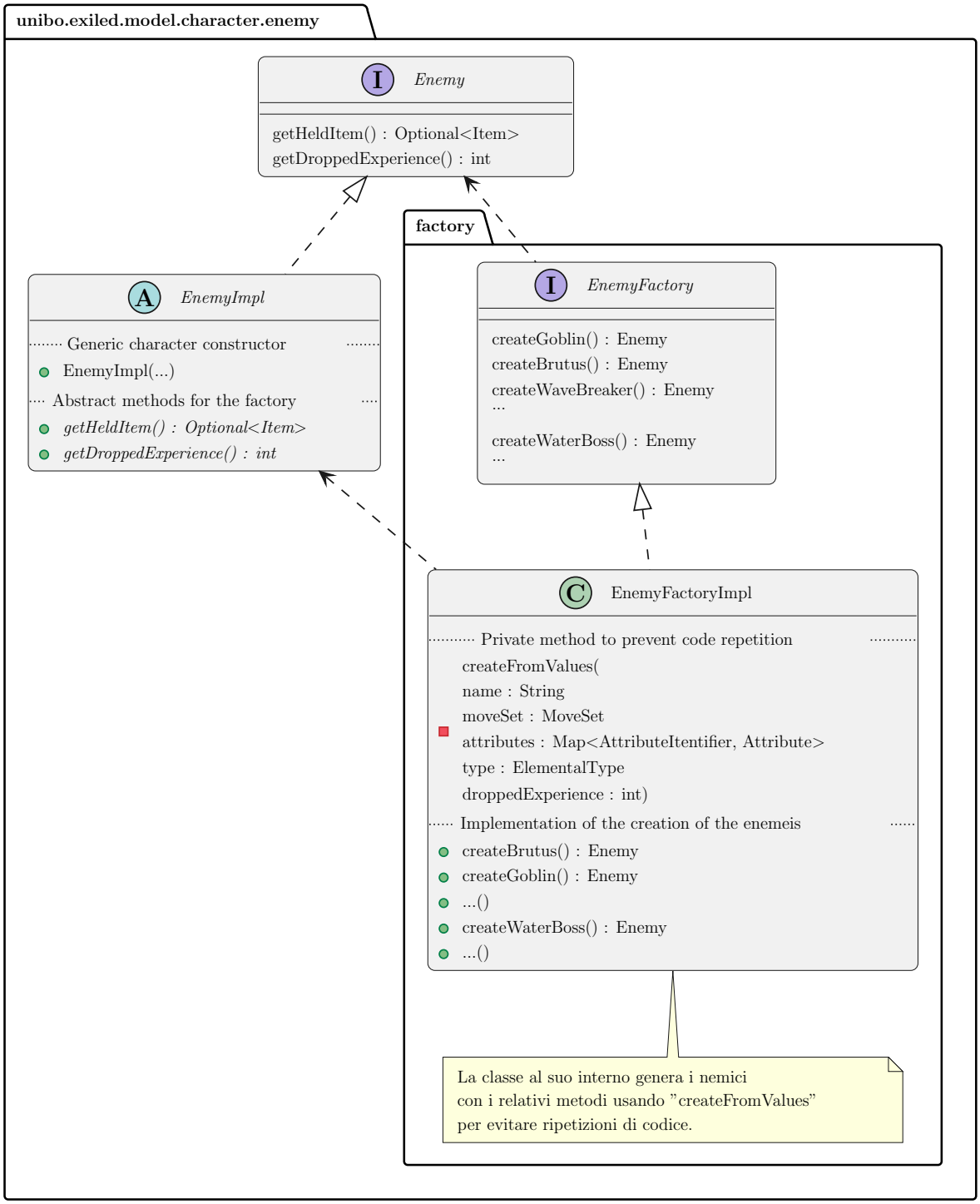

Figura 2.3: Schema UML della factory di nemici (Boss esclusi)

Problema: Divisione dei boss dai nemici generici, per evitare che l'oggetto rilasciato al giocatore venga scelto casualmente.
Soluzione: In questo caso è stato necessario considerare i boss come una specializzazione di nemici generici, quindi come sottoclassi di *EnemyImpl* ma con una loro classe specifica dedicata che consenta di scegliere manualmente il tipo di oggetto rilasciato alla sconfitta. Segue uno schema UML che aggiunge il concetto di nemico.

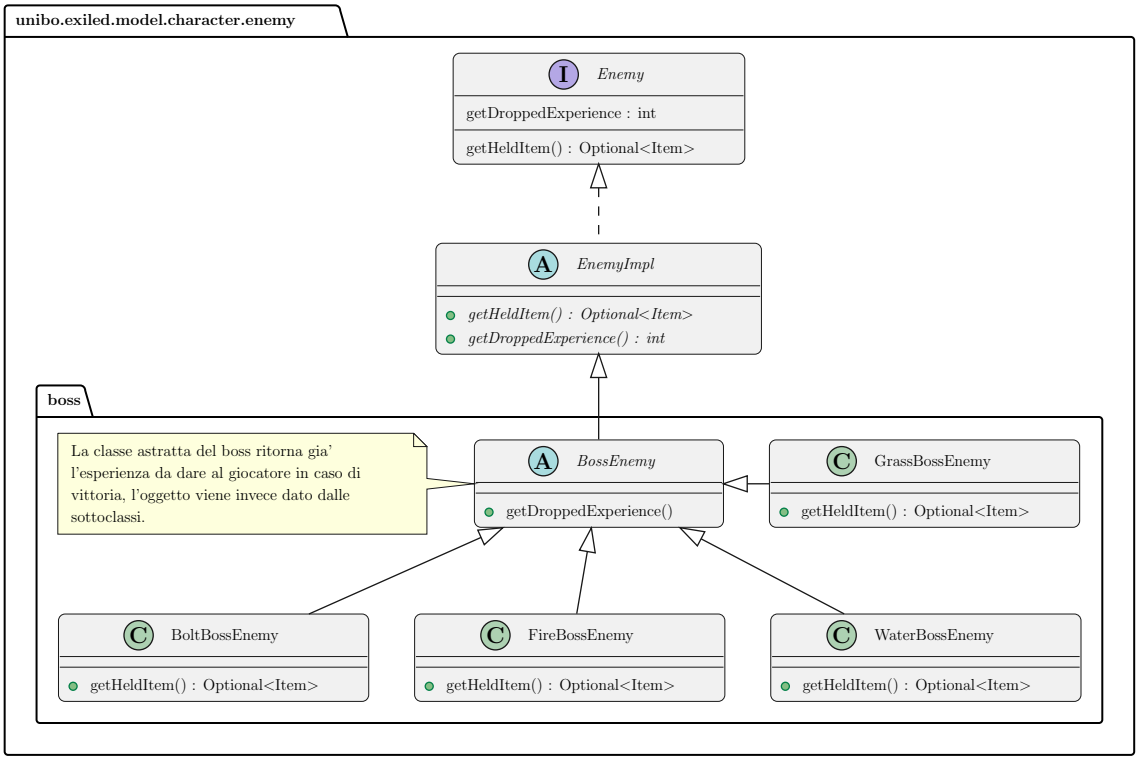

Figura 2.4: Schema UML della gestione dei Boss.

In maniera analoga nella "Factory" di nemici descritta precedentemente vengono utilizzate le classi dei boss per la loro generazione.

Raggruppare i nemici in collezione

Problema: Come raggruppare assieme tutti i nemici generati per poterli disporre sulla mappa?
Soluzione: In questo caso ho creato un'interfaccia apposita *EnemyCollection* che segue il pattern Iterator per poter scorrere tutti i nemici indipen-

dentemente da come venga implementata la collezione. Segue uno schema UML della struttura.

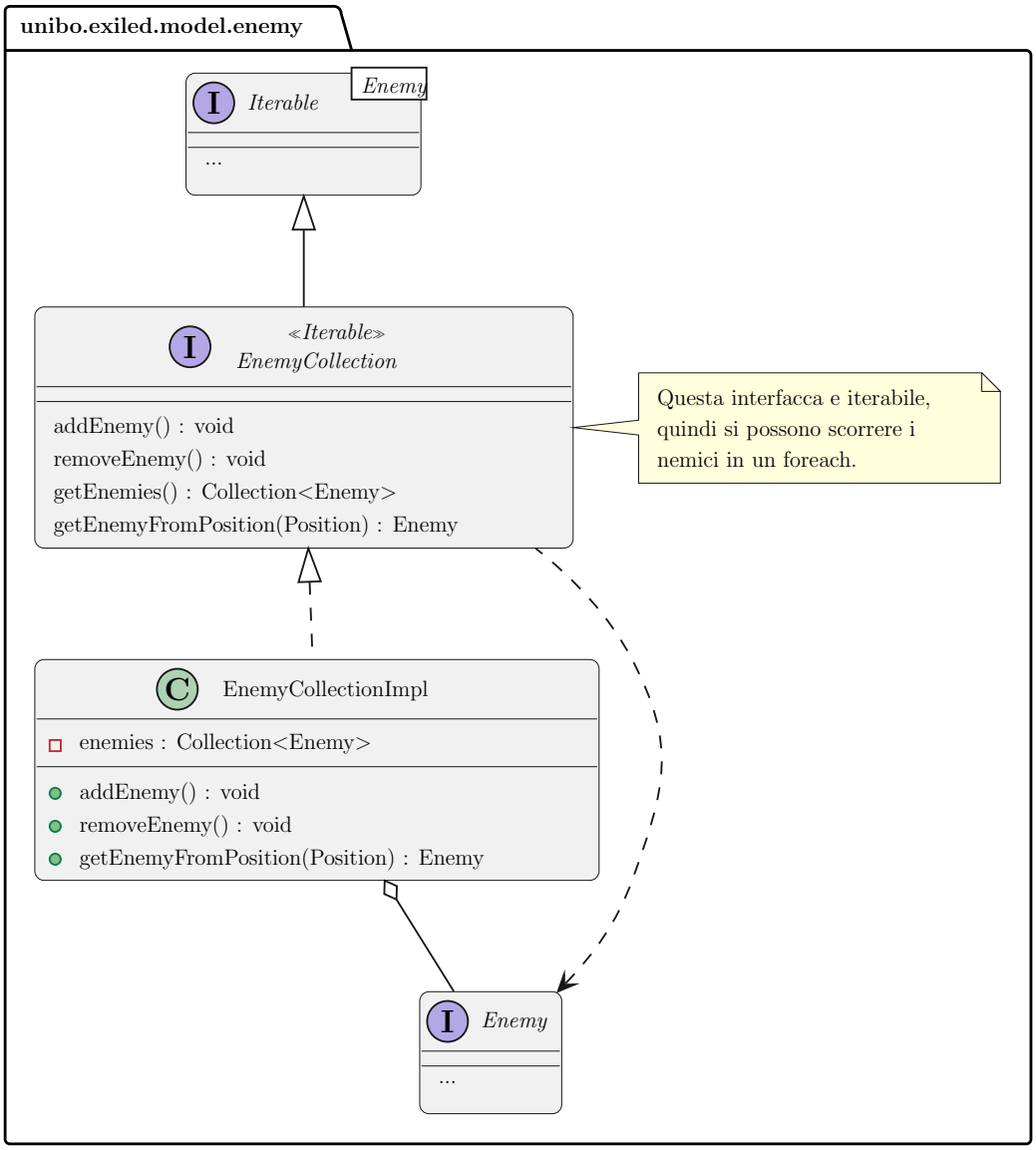

Figura 2.5: Schema UML della collezione di nemici per gestire la mappa.

Attributi di un Character (Nemico o player)

Problema: Bisogna fare una distinzione tra attributi che aggiungono una quantità ad una caratteristica del *GameCharacter*, quelli che invece conten-

gono una costante moltiplicativa per ridurre od aumentare una caratteristica che non riguarda una quantità specifica, ma va applicata al momento della battaglia, e di quelli invece che fanno entrambe queste due cose.
Soluzione: Gli attributi vengono gestiti mediante un'interfaccia Attribute che contiene due metodi che consentono di stabilire di che tipo di attributo si tratti. Questi due metodi vengono implementati nelle classi che implementano le sotto interfacce *AdditiveAttribute*, *MultiplierAttribute* e *CombinedAttribute*.
Qui presento uno schema UML che rappresenta come sono gestite le distinzioni tra attributi moltiplicativi e additivi o combinati.

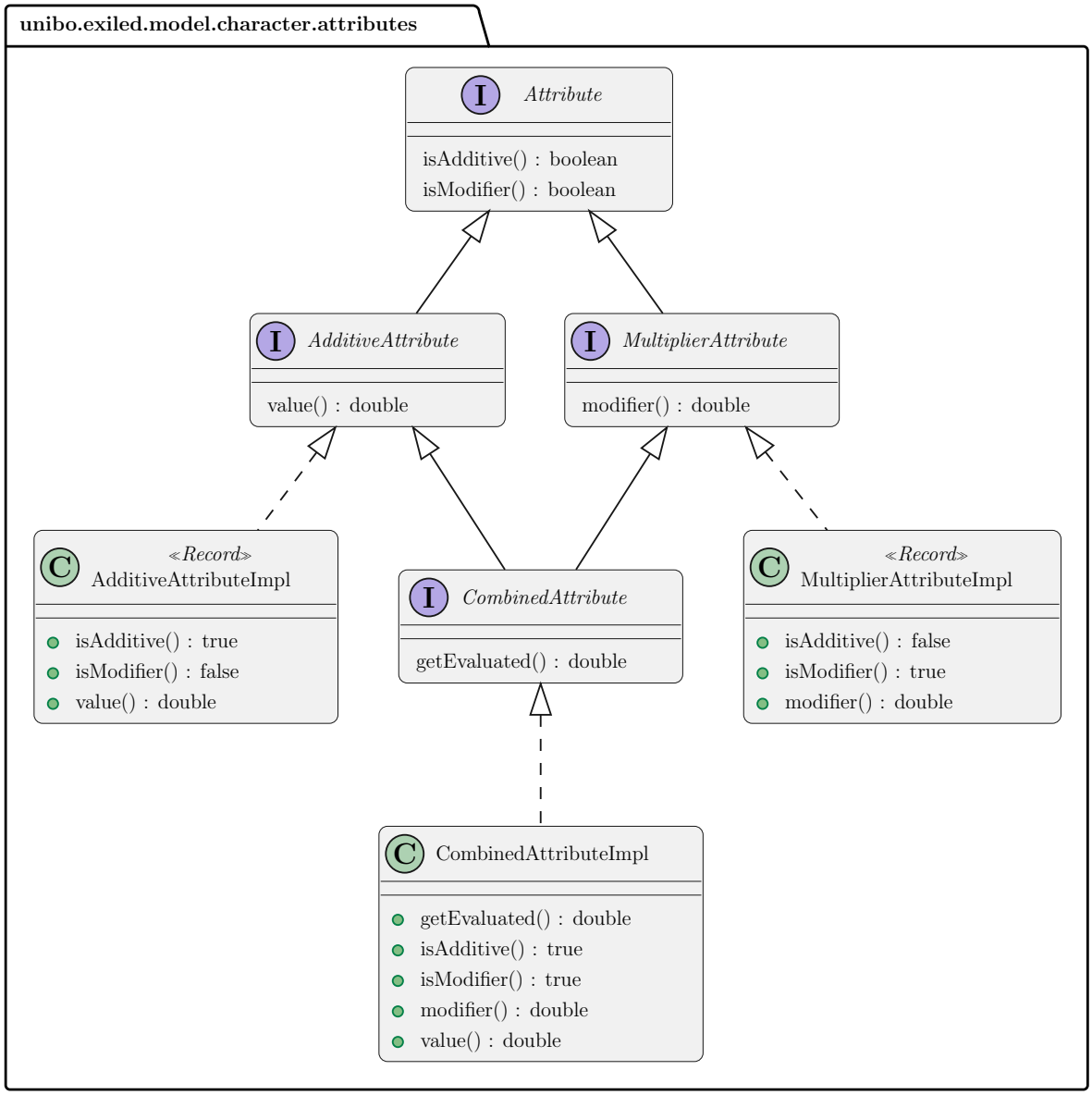

Figura 2.6: Schema UML della divisione degli attributi

Problema: Come gestire tutte le caratteristiche di gioco del giocatore o di un nemico in modo tale da non avere una quantità eccessiva di campi all'interno di una classe e in modo tale da gestire la scalabilità? E come identificare ciascun attributo esistente?
Soluzione: È stato introdotto un enumeratore con tutti gli identificatori di attributi necessari al gioco, questo si chiama *AttributeIdentifier* e contiene

l'associazione di ciascun elemento dell'enumeratore al corrispettivo nome in stringa così da poter facilmente essere interpretato dal controller e successivamente la view. Per la generazione degli attributi di un *GameCharacter* invece è stato utilizzato il pattern *Factory Method* attraverso l'interfaccia denominata *AttributeFactory* e la sua implementazione *AttributeFactoryImpl* che creano una mappatura di tutti gli attributi necessari per ciascun *GameCharacter*, così da poter diversificare gli attributi di ognuno senza aggiungere campi eccessivi alle classi.
In tal modo un *GameCharacter* contiene al suo interno una mappatura tra *AttributeIdentifier* e relativo *Attribute* in modo tale da poter definire attraverso la factory gli attributi di ciascuno.
Segue uno schema UML della factory.

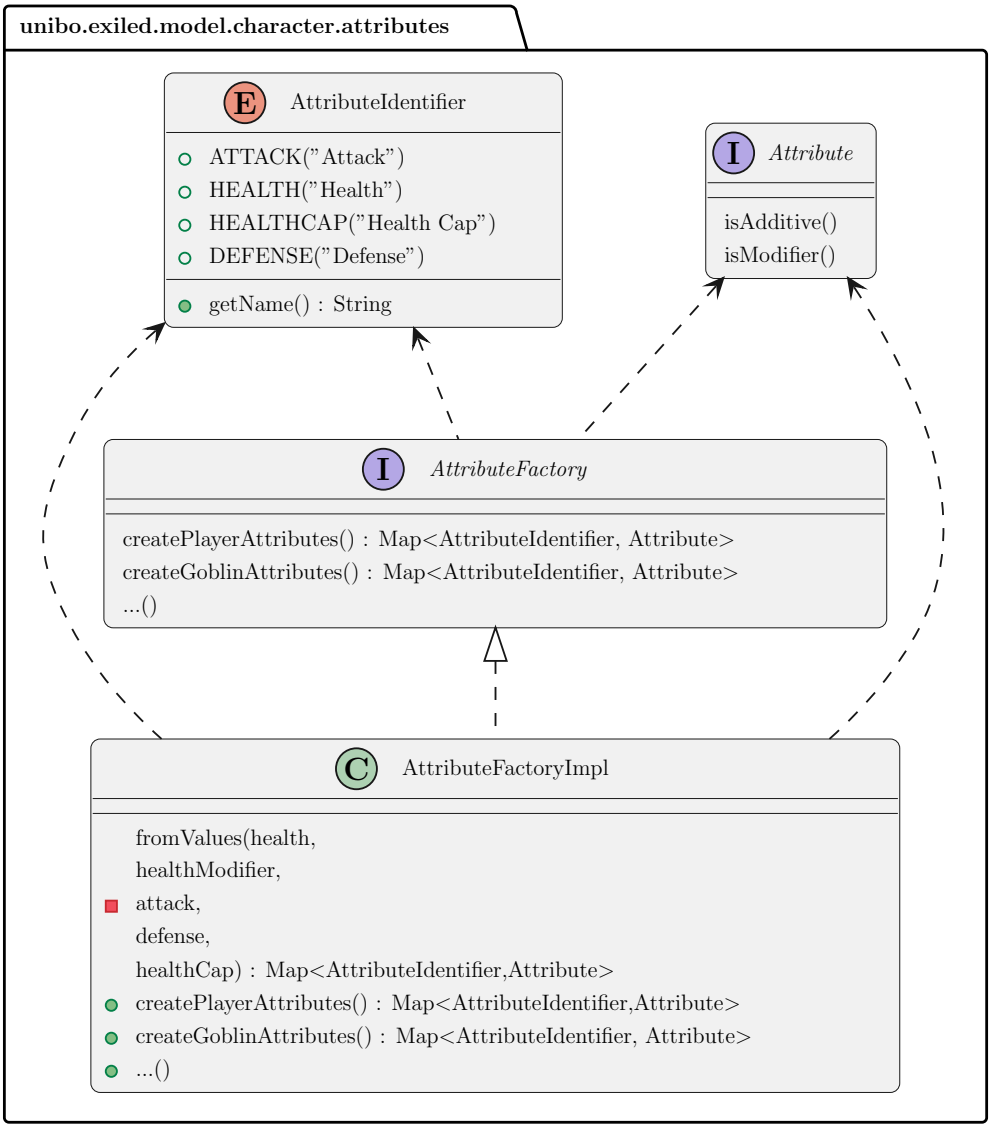

Figura 2.7: Schema UML con la factory degli attributi.

Piazzamento dei nemici sulla mappa : *CharacterModel*

Problema: Mettere in relazione la mappa e i nemici attraverso un'unico concetto nel model.
Soluzione: Effettuare lo scattering dei nemici durante la loro inizializzazione, al momento della creazione del model. Una delle parti da me gestite è stata quella dell'inizializzazione dei nemici utilizzando le classi sopradescritte, e il loro posizionamento sulla mappa attraverso alcuni metodi all'interno dell'implementazione dell'interfaccia *CharacterModel*.

In particolare, nella classe *CharacterModelImpl* è contenuta la *EnemyCollection* che servirà al controller per scorrere su ogni nemico presente, vi è anche un metodo di inizializzazione in cui si prende il numero di nemici da configurazione, e si generano tutti attraverso la *EnemyFactory* e vengono man mano posizionati in celle (e aggiunti alla collezione) a patto che:

- Non vengano posizionati in uno dei 4 vertici della mappa. (Lì ci sono i boss)
- Non vengano posizionati in una cella il cui tipo non corrisponde a quello del nemico.
- Non vengano posizionati in una cella in cui sia già presente un nemico o il giocatore (cella piena)

Una volta generati tutti i nemici generici, vengono creati i 4 boss sempre attraverso la factory e vengono aggiunti anch'essi alla *EnemyCollection* perché si tratta sempre di nemici e vengono spostati manualmente ai quattro vertici della mappa, attraverso un apposito metodo che dato il tipo di cella ne ricava l'angolo della mappa (Spiegato nella prossima sezione relativa alla mappa).

Mappa di gioco nel *Model*

Problema: Come gestire la mappa di gioco a livello di divisione delle celle? Come dare un certo tipo ad ogni cella?
Soluzione: Ho utilizzato un enumeratore per elencare tutti i possibili tipi di cella, e a seconda della loro quantità questi vengono distribuiti in maniera omogenea sulla mappa attraverso i metodi di inizializzazione all'interno di *GameMap* e la sua implementazione *GameMapImpl*.
Quest'ultima inoltre si occupa di definire quali sono i 4 angoli della mappa dove posizionare i boss in base al loro relativo tipo. La mappa ha ovviamente una dimensione stabilita dalla classe di configurazione dove sono presenti tutte le costanti. Le macroaree della mappa vengono posizionate in maniera casuale, questo significa che se la parte con tipo elementale "Acqua" si trovava in alto a destra in una partita, nella successiva la stessa potrebbe trovarsi in quella in alto a sinistra o in qualsiasi altra partizione della mappa a griglia.
Segue uno schema UML che descrive il modello della mappa.

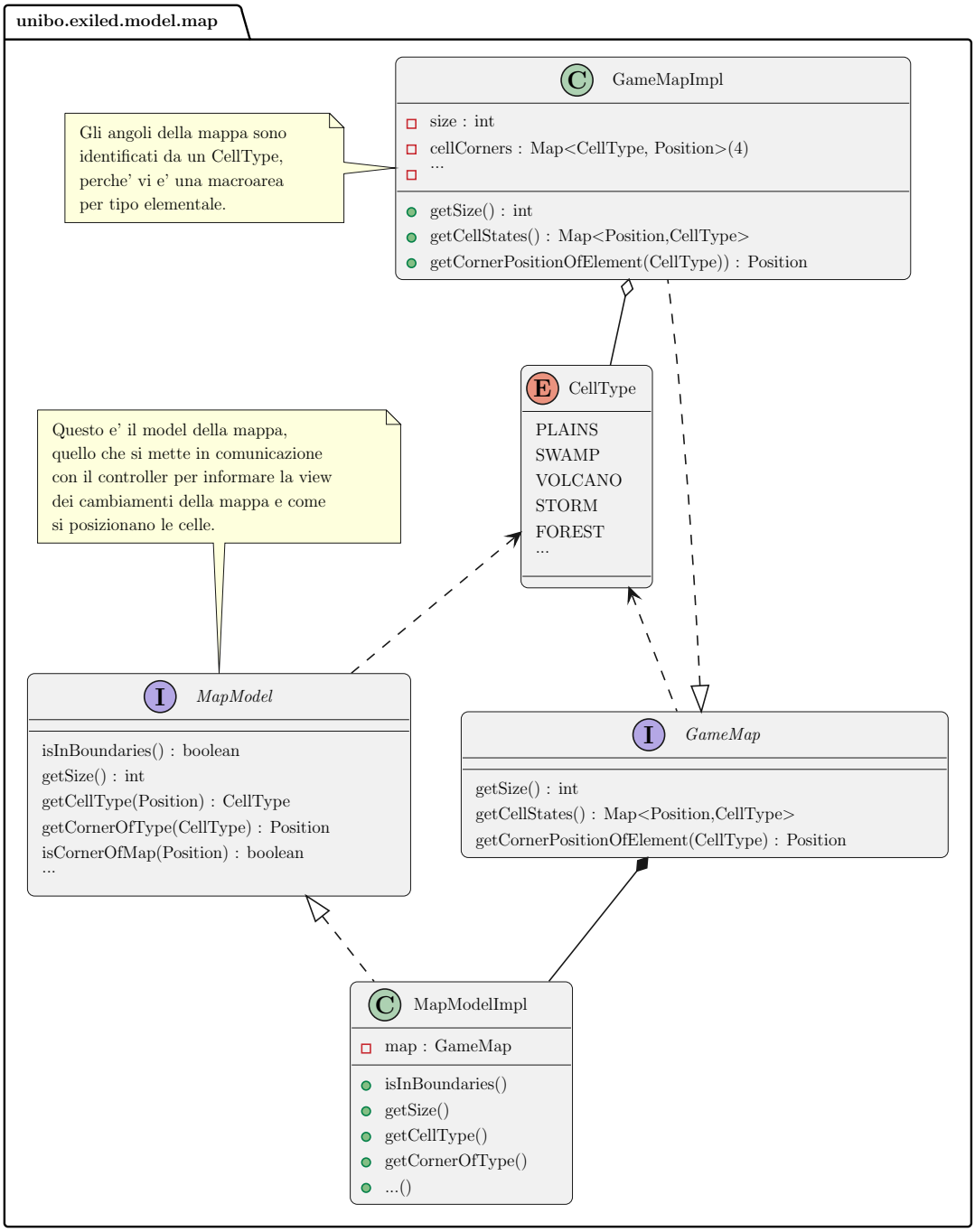

Figura 2.8: Schema UML della mappa e parte di model.

La parte che mette in interazione sia i nemici che il player e il controller è parte condivisa ed è gestita all'interno della classe *CharacterModelImpl*, men-

tre la sopradescritta *MapModelImpl* è una parte di model specificatamente pensata per la mappa.

2.2.2 Francesco Pazzaglia

La mia parte di progetto si concentra sull'implementazione dettagliata del giocatore, comprendendo la definizione delle sue caratteristiche quali attributi(la vita, l'esperienza, la relativa classe...) e le sue meccaniche in gioco. Mi sono anche occupato della gestione del movimento del giocatore e delle altre entità (i nemici) all'interno della mappa di gioco. Infine, ho implementato la gestione delle condizioni di chiusura del gioco(vittoria e game over), che includevano la definizione delle regole e dei possibili eventi che portavano al termine del gioco, nonché la gestione delle rispettive schermate.

Player

Nella parte inerente al giocatore, il problema principale consisteva nell'identificare e gestire in modo efficiente le caratteristiche comuni con i nemici, al fine di evitare ripetizioni e mantenere il codice conciso. Pertanto, in collaborazione con *Luca Casadei*, responsabile della parte relativa ai nemici, abbiamo deciso di introdurre nel Model un'interfaccia denominata *GameCharacter*. Questa interfaccia contiene i metodi principali che definiscono le caratteristiche fondamentali comuni a tutte le entità nel gioco, come la vita, la posizione, il movimento e gli attributi. In aggiunta all'interfaccia, è stata implementata la classe astratta *GameCharacterImpl*, che concretizza l'interfaccia *GameCharacter*. Questa classe è stata progettata in modo che il giocatore possa estenderla e acquisire le caratteristiche comuni menzionate in precedenza. Questa architettura ci ha consentito di gestire in modo efficiente le proprietà condivise tra il giocatore e i nemici, garantendo una maggiore coerenza e facilità di estensione del codice senza ripetizioni.

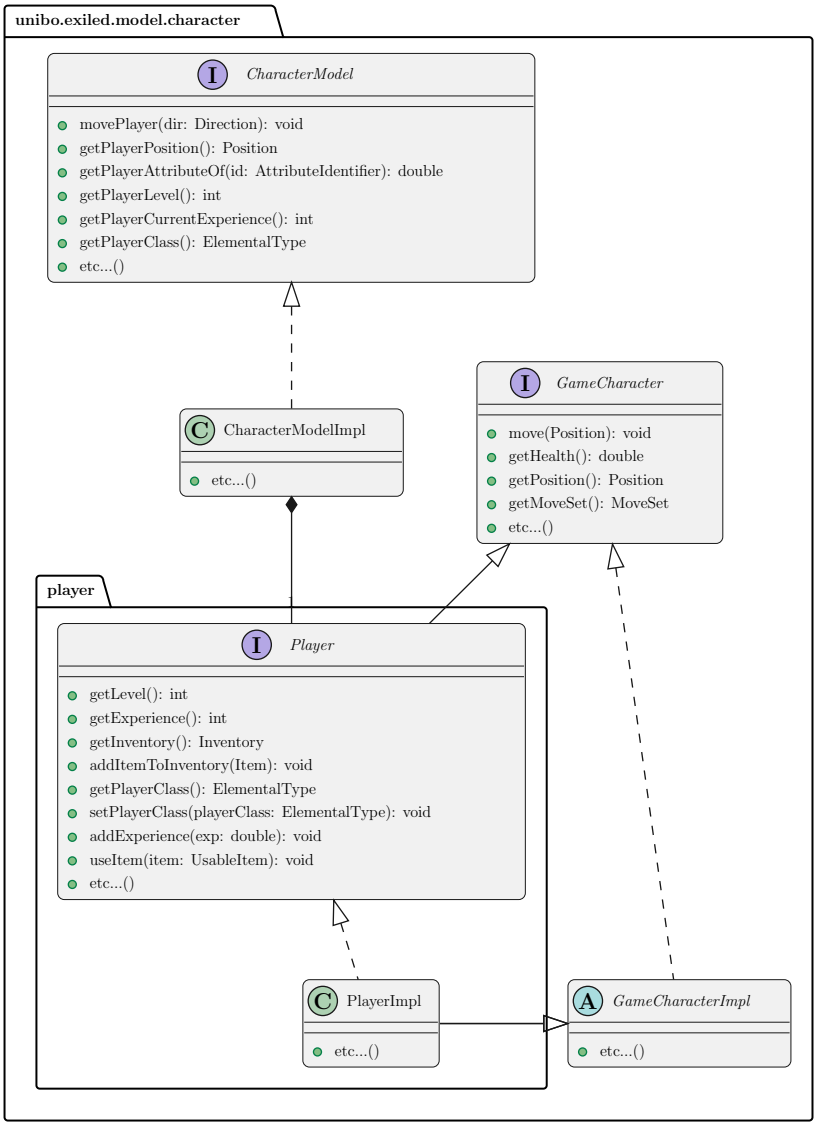

Figura 2.9: Schema UML del Model del Player e del GameCharacter

In maniera più approfondita, per la gestione del giocatore, ho introdotto un'interfaccia denominata *Player*, che estende l'interfaccia preesistente *GameCharacter*. L'interfaccia *Player* viene implementata da una classe chiamata *PlayerImpl* che ne definisce ulteriori metodi specifici del giocatore per il livello, l'esperienza, l'inventario ecc.

Per quanto riguarda la parte inerente alla View sia il giocatore che i nemici sono stati gestiti attraverso una classe chiamata *CharacterView*. Questa classe è stata progettata per utilizzare immagini al fine di rappresentare graficamente le entità del gioco. In particolare, sono state gestite delle ani-

mazioni mediante l'utilizzo di due immagini per ciascuna direzione (NORD, SUD, EST e OVEST). Oltretutto sia per il *Player* che per i nemici, viene scelta un'immagine colorata in base alla tipologia di appartenenza. La *CharacterView* ha permesso una gestione separata della scalabilità delle entità all'interno della griglia di gioco in caso di ridimensionamento della finestra e ci ha permesso inoltre di effettuare il cambio di immagine per le animazioni.

Per garantire la comunicazione tra il Model e la View del *Player* è stato introdotto un Controller, denominato *CharacterController* dove, al suo interno sono stati definiti i metodi per il controllo del *Player*, tra cui quelli relativi al movimento, all'assegnazione della tipologia, all'uso di modificatori di esperienza e di livello, nonché i relativi metodi getter necessari per l'interfaccia grafica. Oltre a questi, sono stati definiti metodi generali applicabili a qualsiasi entità presente nel gioco. Per tale motivo, è stata introdotta un'ulteriore interfaccia denominata *CharacterModel* e la relativa classe *CharacterModelImpl* che contiene le entità del gioco, ovvero il Player e una collezione di nemici. Oltre a ciò sono stati inseriti i metodi della parte di Model per la gestione di entrambe le entità consentendo al Controller di avere una organizzazione uniforme e coerente.

Il design pattern utilizzato per gestire la parte del *Player* è Template Method. Questo pattern consente di definire uno scheletro di comportamento comune per il *Player*, infatti la classe astratta *GameCharacterImpl* costituisce la base per tutte le entità di gioco, ad esempio definisce il metodo *move(Position)*, ossia la struttura generica per il movimento di un'entità.

Gestione del movimento

Nella gestione del movimento, il problema principale è stato comprendere come distinguere e concretizzare il movimento del giocatore da quello dei nemici sulla mappa di gioco. Inoltre, anche la scelta di utilizzare l'input da tastiera o tramite mouse ha generato incertezze riguardo a come realizzare la griglia relativa alla mappa.

In merito a quanto detto, per l'implementazione del movimento delle entità all'interno della mappa di gioco, ho adottato due distinzioni principali. Per il movimento del giocatore ho progettato un sistema basato sull'input dell'utente tramite tastiera. D'altra parte, per i nemici, ho sviluppato un meccanismo autonomo di movimento. In pratica, i nemici si spostano in maniera pseudo-casuale all'interno di un'area definita dalla loro tipologia(Fire, Bolt, Water e Grass) e qualora il giocatore si dovesse avvicinare, entro un certo raggio, ad un nemico, quest'ultimo inizierà a inseguirlo per combattere. Ho scelto questo approccio perché rende sicuramente il gioco più interessan-

te e stimolante per il giocatore, dato che per poterlo completare bisogna sconfiggere i quattro Boss che sono situati agli estremi della mappa di gioco.

Considerando la decisione di non utilizzare i Thread nel nostro progetto, i nemici si sposteranno esclusivamente in risposta all'input dell'utente. Di conseguenza, i nemici si muoveranno solamente quando l'utente eseguirà uno spostamento nella mappa.

In particolare, ho utilizzato il *CharacterModel* per definire il meccanismo di movimento sia per il personaggio che per i nemici. Successivamente, all'interno del *CharacterController*, ho implementato un metodo di movimento che ha richiamato le funzioni precedentemente definite nel Model.

E' stata realizzata una classe *GameKeyListener* della View, dove ho gestito l'input dell'utente associando i movimenti alle direzioni corrispondenti ai tasti premuti (ad esempio, W per su, S per giù, A per sinistra e D per destra). La *GameView* rappresenta la schermata principale dov'è inserita la mappa e l'HUD che visualizza gli attributi principali del personaggio. Quando l'utente preme un tasto specifico sulla tastiera tra quelli elencati prima, all'interno della *GameView*, il *CharacterView* associato al giocatore si sposterà in una cella corrispondente. Di conseguenza, come precedentemente descritto, tutti i nemici, a meno che non siano vicini al giocatore, si sposteranno anch'essi, in una direzione scelta in modo pseudo-casuale.

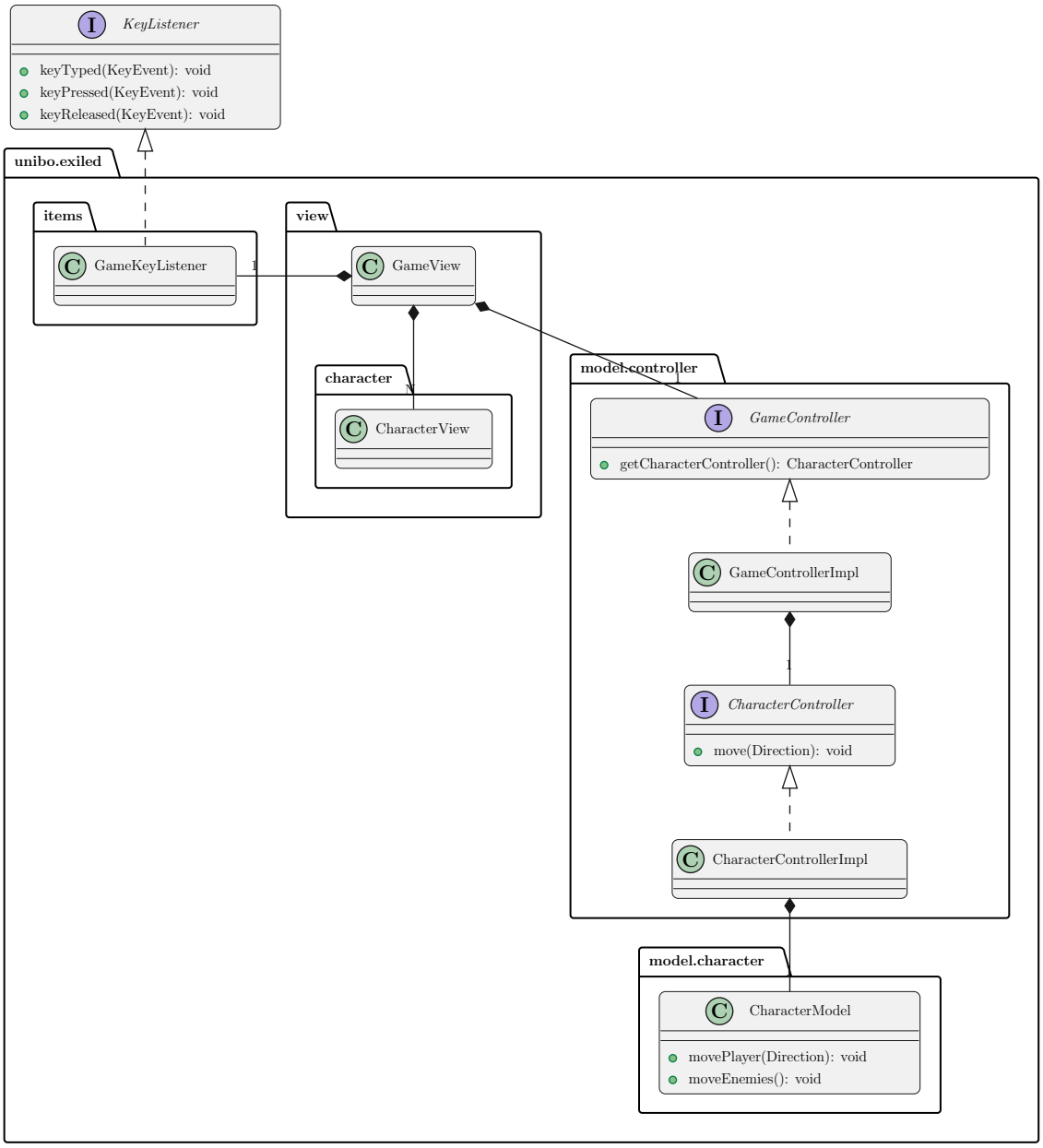

Figura 2.10: Schema UML inerente al meccanismo di movimento

Oltre ai *KeyListener* di base per il movimento del giocatore, ho integrato la funzionalità che permette di utilizzare il tasto E per aprire l'inventario quando si è in gioco e il tasto ESC per aprire/chiudere il menu di gioco. Questa scelta è stata fatta per migliorare l'esperienza di gioco e rendere

l'interfaccia più intuitiva e user-friendly, consentendo ai giocatori di navigare facilmente tra le diverse funzionalità senza forzare l'utilizzo del mouse.

Inoltre, ho implementato la gestione della visibilità della mappa in base allo spostamento del giocatore. È stato definito un range di visibilità, in modo che l'utente possa vedere solo le celle adiacenti al giocatore all'interno di tale intervallo.

Avendo separato la gestione del movimento e dell'ascolto dei *KeyListener* in una classe dedicata, è stato adottato il design pattern Observer. Questo pattern ha consentito di definire un meccanismo di notifica personalizzato attraverso la classe *GameKeyListener*, che funge da Observer. *GameKeyListener* implementa l'interfaccia *KeyListener* e si occupa di ricevere notifiche sugli eventi di pressione dei tasti. In questo modo, sono stato in grado di gestire dinamicamente il movimento delle entità di gioco in risposta all'input dell'utente.

Meccanismo di fine gioco

Il problema relativo al meccanismo di fine gioco riguardava l'interfacciamento e la visualizzazione delle relative schermate, a tal proposito mi sono occupato di gestire la conclusione del gioco attraverso due scenari distinti: la vittoria e la sconfitta).

In entrambi gli esiti, ho realizzato la gestione di fine gioco attraverso il *GameController*.

Utilizzando il metodo *isOver()*, viene deciso quando il gioco è terminato e di conseguenza quando deve essere visualizzata l'immagine inerente al game over. Avendo un unico metodo per determinare la sconfitta del giocatore, ho potuto facilmente definire il suo criterio. Essendo un gioco centrato su un singolo giocatore principale, è stato naturale impostare il game over nel caso in cui la sua vita dovesse azzerarsi o scendere sotto lo zero, a seguito di un combattimento con un nemico.

Lo stesso principio si applica alla vittoria, la quale rappresenta la schermata che compare quando il gioco è completato con successo. Questa viene visualizzata quando il metodo *isWon()* restituisce un valore che indica se il giocatore ha vinto il gioco. Per poter vincere il gioco, il personaggio deve riuscire a sconfiggere i Boss finali e quindi ad ottenere i quattro cristalli della redenzione.

All'interno della *GameView*, la principale vista definita in precedenza, è presente il metodo *draw()*, dove viene controllato lo stato del gioco. In caso di fine gioco, viene istanziata una vista chiamata *EndGameView*, che rappresenta in modo generico la schermata di fine gioco. Se il gioco è stato

vinto, la *EndGameView* mostrerà l'immagine di vittoria, mentre in caso di sconfitta verrà visualizzata l'immagine relativa al game over. Questa gestione delle immagini avviene attraverso l'utilizzo dell'enumeratore *EndState*, che permette di identificare lo stato di fine gioco.

All'interno della *GameView* è anche contenuto il *GameController*, che facilita il collegamento tra il Model e la View stessa. Grazie a questo Controller, è possibile determinare quando la schermata principale deve essere sostituita dalla quella di vincita o perdita del gioco.

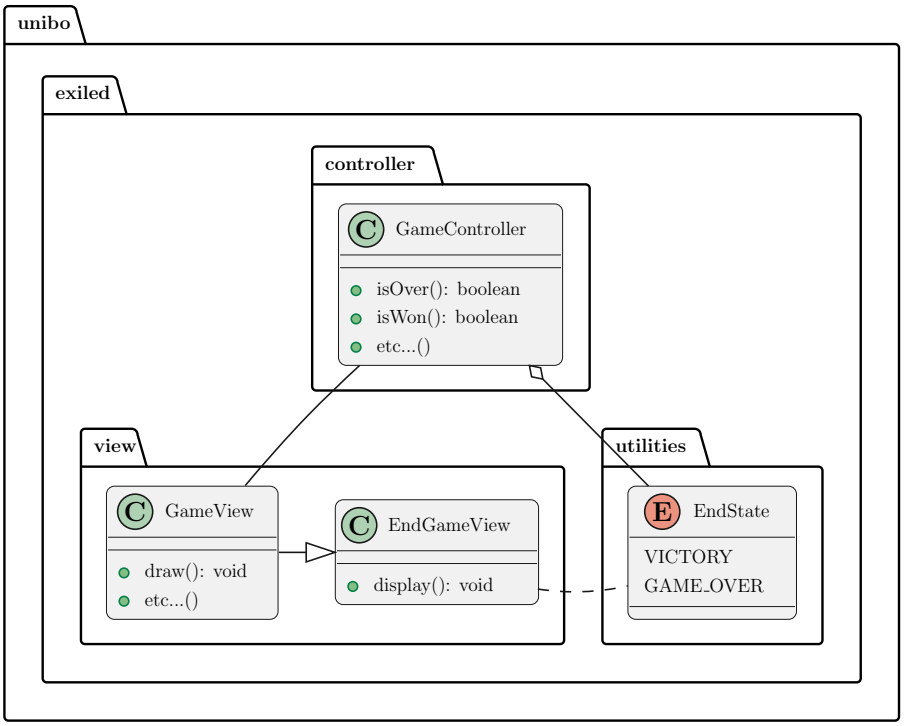

Figura 2.11: Schema UML dell'End Game con relativa vista sulle schermate

2.2.3 Marco Magnani

La parti del progetto in cui ho impiegato la maggior parte del tempo sono l'inventario del player, gli oggetti da cui esso è composto, la hud, la logica per l'aumento di livello del player e le mosse magiche.

Item e Inventory

Per implementare la gestione delle entità degli oggetti, ho adottato un approccio di astrazione mirato a massimizzare la riutilizzabilità del codice e a eliminare duplicazioni non necessarie. Nel contesto del gioco, gli oggetti possono essere categorizzati in tre tipologie distinte: curativi, di potenziamento e risorse (ad esempio, cristalli). L'obiettivo era creare un sistema in cui l'inventario potesse accogliere tutti questi tipi di oggetti diversi, trattandoli però uniformemente come se appartenessero tutti alla stessa categoria.

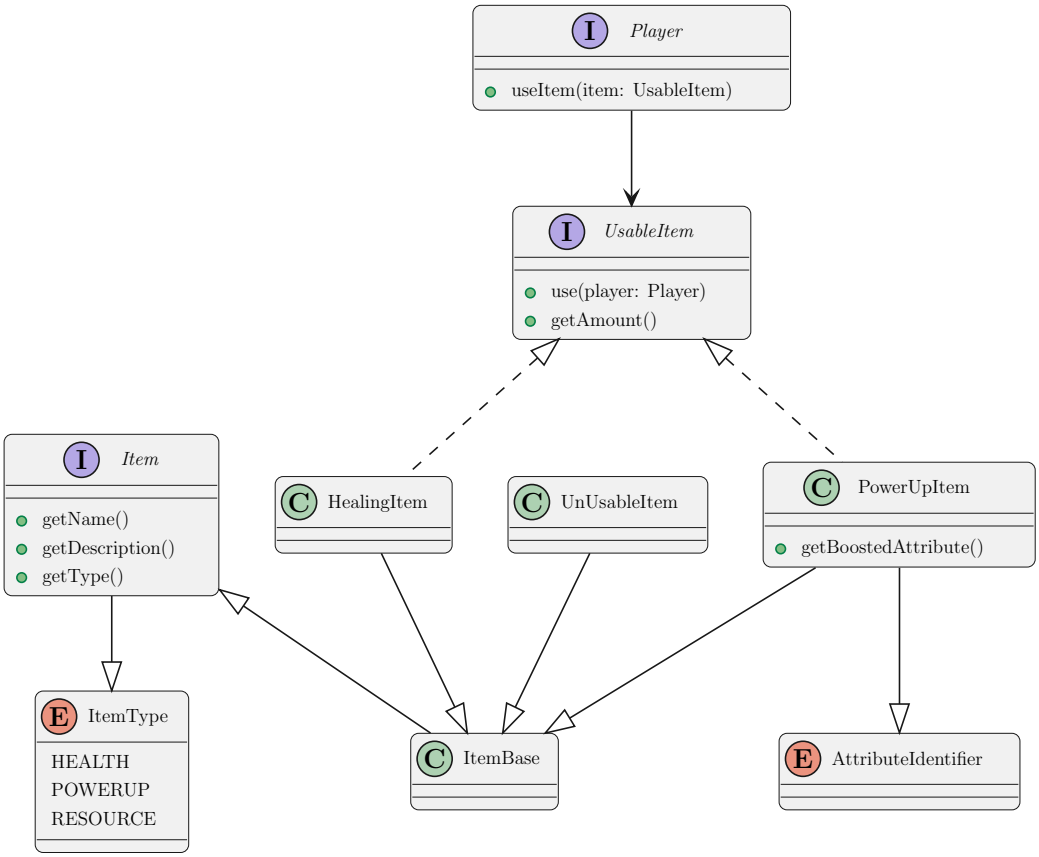

Figura 2.12: Schema UML del Model degli Item e dell'Inventory

La struttura che ho concepito prevede che tutti gli oggetti estendano un'interfaccia astratta, garantendo uniformità nei campi essenziali. Un ostacolo che ho affrontato riguardava la suddivisione tra oggetti curativi e di potenziamento, poiché entrambi possono essere utilizzati nel gioco, ma la logica di implementazione per i due utilizzi è differente.

Per superare questa sfida, ho introdotto l'interfaccia *UsableItem*, seguendo il design pattern *Template Method*. Questa interfaccia consente ad entrambe le categorie di implementare la propria logica di utilizzo, differenziandosi attraverso il metodo *"use()"*.

L'astrazione fornita da UsableItem aiuta anche per la visualizzazione dell'inventario, in quanto consente di rendere cliccabili solo gli oggetti effettivamente utilizzabili.

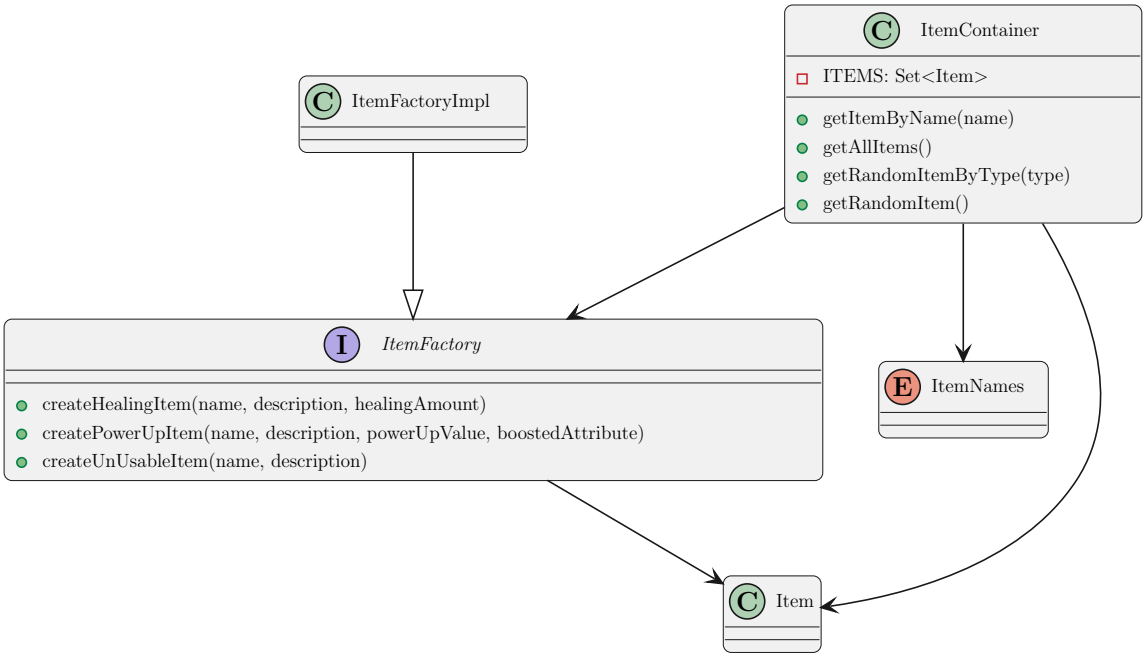

Figura 2.13: Schema UML della ItemFactory e dell'ItemContainer

Per risolvere un altro problema legato alla creazione degli oggetti, ho voluto garantire che qualsiasi classe che necessiti di creare un oggetto non dovesse conoscere la logica specifica di creazione. A questo scopo, ho implementato una Factory utilizzando il design pattern *Factory* per la creazione degli item. Questa Factory viene utilizzata dall'Item Container per gestire la creazione di tutti gli oggetti di gioco. L'enum *ItemNames* è stato introdotto per contenere i nomi di tutti gli oggetti del gioco, svolgendo il ruolo di "chiavi" per identificare gli item.

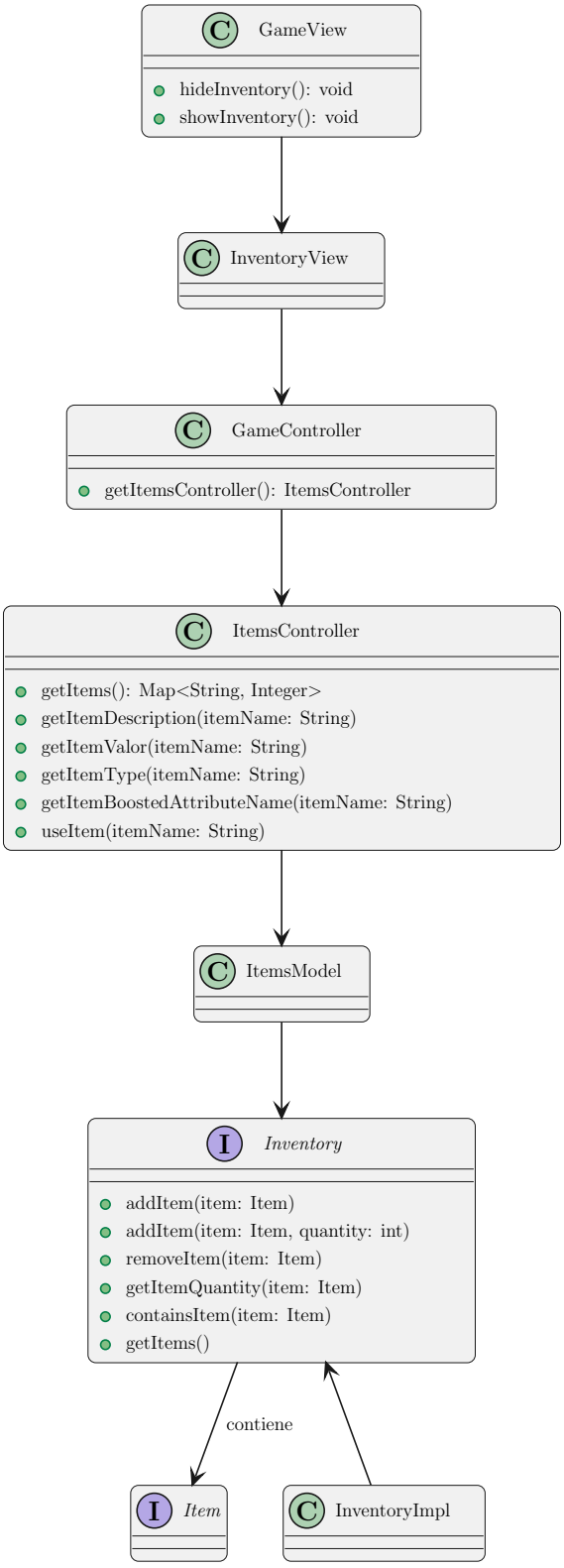

Figura 2.14: Schema UML dell'inventario

Gli oggetti sono tutti generalizzati attraverso l'interfaccia Item, semplificando notevolmente la gestione, specialmente all'interno dell'inventario. Per la View dell'inventario, ho adottato il pattern *Strategy*, in particolare mediante l'interfaccia *ItemController* e la sua implementazione, *ItemControllerImpl*. Quest'ultima è responsabile dell'effettiva logica di gestione degli oggetti nel gioco e interagisce con il modello degli item per gestire l'inventario.

L'utilizzo di questo pattern è stato pensato per rendere il progetto aperto a future estensioni. Se si dovesse aggiungere un'altra visualizzazione degli oggetti, sarebbe possibile creare un'altra implementazione dell'interfaccia ItemController, utilizzando una logica specifica per la gestione degli oggetti in quella particolare vista.

Hud

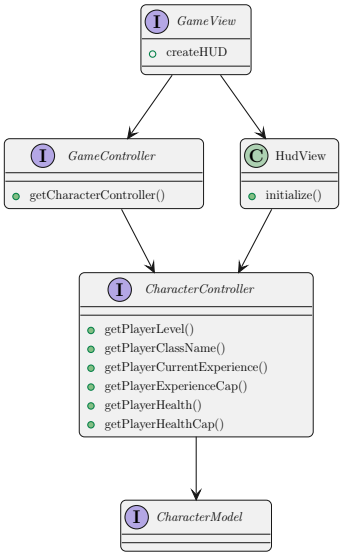

Figura 2.15: Schema UML della HUD

Passando all'HUD (head-up display), quest'ultimo mostra le informazioni di base del giocatore sia sulla pagina principale che durante gli scontri con i nemici. Queste informazioni includono la vita, il livello, la classe e l'esperienza del giocatore. Ho gestito l'HUD come una classe di visualizzazione separata e indipendente, la quale può essere resa visibile e aggiornata quando necessario.

Dal momento che l'HUD è sempre visibile, anche durante i combattimenti, una problematica è stata quella di mantenere costantemente aggiornate le statistiche del giocatore in base ai suoi cambiamenti di stato, come la diminuzione della vita o l'aumento di livello. Per risolvere questo problema,

ho optato per un'implementazione simile ad un *Observer*. Ogni volta che il modello del giocatore subisce una modifica, ad esempio quando la vita diminuisce, l'HUD viene notificato attraverso il metodo initialize dei cambiamenti, consentendogli di aggiornare la visualizzazione dell'HUD di conseguenza.

Levelling

Infine, ho gestito il processo di aumento di livello del giocatore, che coinvolge la logica di incremento del livello raggiungendo una certa quantità di esperienza. Ho implementato anche il calcolo dell'esperienza necessaria per il prossimo livello, l'aumento delle statistiche ad ogni livello e la logica secondo cui il giocatore apprende nuove mosse ad intervalli specifici. Per quanto riguarda l'aumento delle statistiche ad ogni incremento di livello, ho scelto di utilizzare un approccio randomizzato al fine di rendere il gioco meno monotono.

Per la logica delle nuove mosse apprese, ho adottato uno schema specifico: il giocatore di base apprende mosse della sua classe (ad esempio, mosse di tipo acqua se la classe è acqua). Se il giocatore conosce già tutte le mosse di quel tipo, imparerà casualmente una mossa di qualsiasi tipo. Questa scelta è stata fatta per rendere la gestione delle mosse più scalabile, in considerazione della possibilità di creare nuovi tipi elementali. Ad esempio, se venisse introdotto un tipo con una singola mossa, il giocatore comunque apprenderebbe nuove mosse.

2.2.4 Manuel Baldoni

All'interno del progetto mi sono occupato principalmente della realizzazione del combattimento e del menu di gioco. Ho contribuito alla realizzazione di alcuni metodi del Character e delle mosse e della gestione del toggle dei pannelli all'interno della view del gioco. Ho implementato la gestione del cambio della mossa del player al level up in collaborazione con Fracesco Pazzaglia.

Combat

Nella gestione del combattimento ci sono state principalmente due sfide: oggettificazione del concetto di combattimento, gestione del timing di attacco. Per quanto riguarda la prima sfida, la difficoltà stava nel fatto che il combattimento di fatto non è un oggetto simile agli altri. Il combattimento al suo interno ha il *Player*, che è sempre lo stesso e un *Entità "Nemico"*, che

varia in ogni combattimento. Proprio perché un combattimento è un concetto astratto va inizializzato ogni volta che il player incontra un nemico, tramite il metodo initializeCombat nel CombatController. Per quanto riguarda la seconda sfida, l'obiettivo era quello di avere un'interfaccia e un comportamento simile a quella del gioco *Pokemon*. Nel quale una volta che il player ha effettuato la sua mossa il nemico aspetta qualche istante prima di effettuare la sua e vi è inoltre, una descrizione di quello che avviene durante la battaglia. Per avere questo comportamento ho utilizzato il design pattern State in combinazione con l'oggetto Timer di Java Swing. Quando il giocatore effettua una mossa viene cambiato immediatamente lo stato del combattimento, viene effettuata la routine di attacco e vengono modificate le proprietà necessarie. Tuttavia l'aggiornamento della view viene effettuato in un secondo momento. Come anche il conseguente attacco del nemico, che viene appunto *triggerato* in un secondo momento. Nella view del combattimento c'è una sezione apposita alla descrizione di cosa sta succedendo.
Stati del combattimento:

- *IDLE*: Lo stato iniziale del combattimento. In questo stato all'inizio viene mostrato nella sezione di descrizione una stringa di questo tipo "_____" successivamente viene mostrata l'ultima mossa effettuata dal nemico;
- *ATTACKING*: In questo stato viene mostrata la stringa (nome attaccante) attacking...;
- *DEFEATING*: Questo stato consente di fa visualizzare al player se è stato sconfitto o se ha sconfitto il nemico;
- *DEFEATED*: Nello stato defeated viene effettuata la routine di sconfitta del player o del nemico.

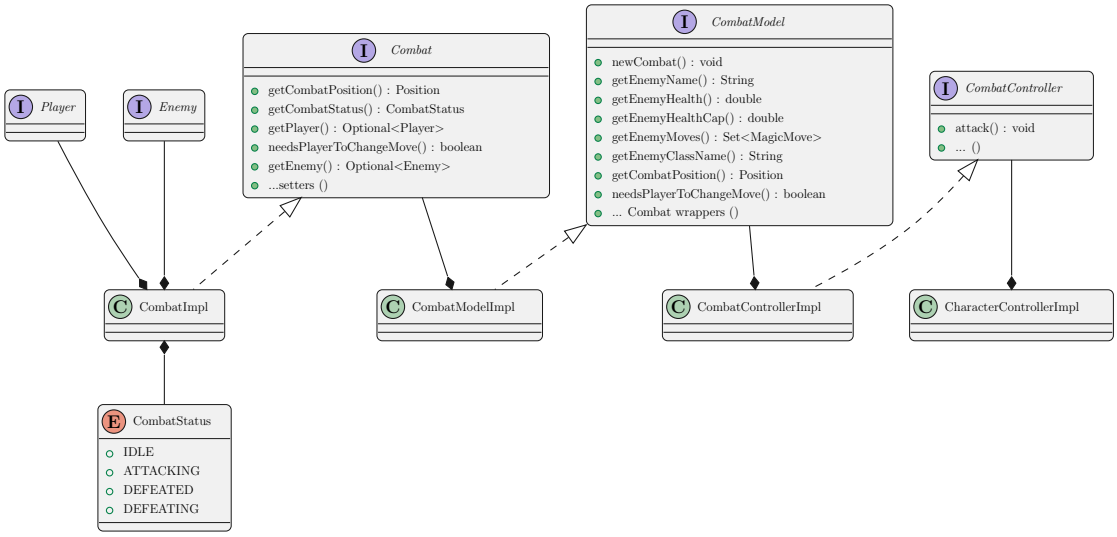

Figura 2.16: Schema UML del Combat

Menu

La difficoltà maggiore nell'implementazione del menu è stata la gestione di diversi menu. Infatti all'interno del gioco sono presenti più menu, ognuno con i suoi pulsanti. I pulsanti posso variare per ogni menu. Per risolvere questo problema ho utilizzato il design pattern Command. Quando viene creato un nuovo menu esso può avere dei bottoni con dei comandi prestabiliti e finiti. Questi comandi vengono poi gestiti dal listener. Nel gioco sono implementati due menu principali: NewGameMenu e InGameMenu.

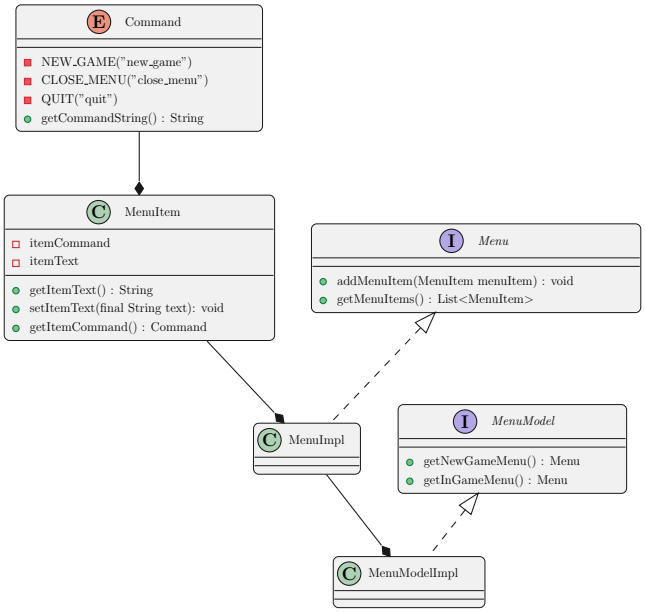

Figura 2.17: Schema UML del Menu

ChangeMove

Il giocatore al *Levelling*, impara una mossa se il livello è un multiplo di 5. Nel caso in cui il numero di mosse del player sia maggiore di 4, l'utente può scegliere se imparare una nuova mossa (sostituendola con una del suo move set) oppure no. Ho creato un campo flag nel combat che mi permette di verificare se il player deve imparare una nuova mossa oppure no. Quando vengono soddisfatte le condizione il flag si attiva e viene mostrata la *GameChangeMoveView*. Alla decisione del player il flag viene disattivato. Il cambio della mossa e la generazione della mossa sono metodi del player.

2.2.5 Elementi di design condivisi

In questa sezione verranno elencate parti di codice create con il contributo di tutti i membri del gruppo.

Mosse dei character

Una mossa è presente all'interno di un'enumerazione denominata *MagicMove*, quest'ultima contiene tutte le informazioni necessarie per rappresentare una mossa (nome, descrizione, potenza, tipo elementale etc. . .). Per utilizzare queste mosse però è necessario immagazzinarle in una qualche collezione

per poter essere facilmente ottenibile dal giocatore o da un nemico, per questo è stato introdotto il concetto di interfaccia *MoveSet* che ha tutti i metodi necessari per aggiungere, rimuovere e rimpiazzare le mosse di una collezione, che a sua volta viene implementata nella realizzazione della suddetta interfaccia, *MoveSetImpl*.
I *MoveSet* dei diversi *GameCharacter* sono generati attraverso l'uso del pattern Factory Method nell'interfaccia *MoveSetFactory* implementata dalla classe *MoveSetFactoryImpl* che contiene i metodi di creazione di MoveSet basilari di default o specifici (come quelli dei nemici o dei boss).

Segue uno schema UML che rappresenta come vengono gestite le mosse.

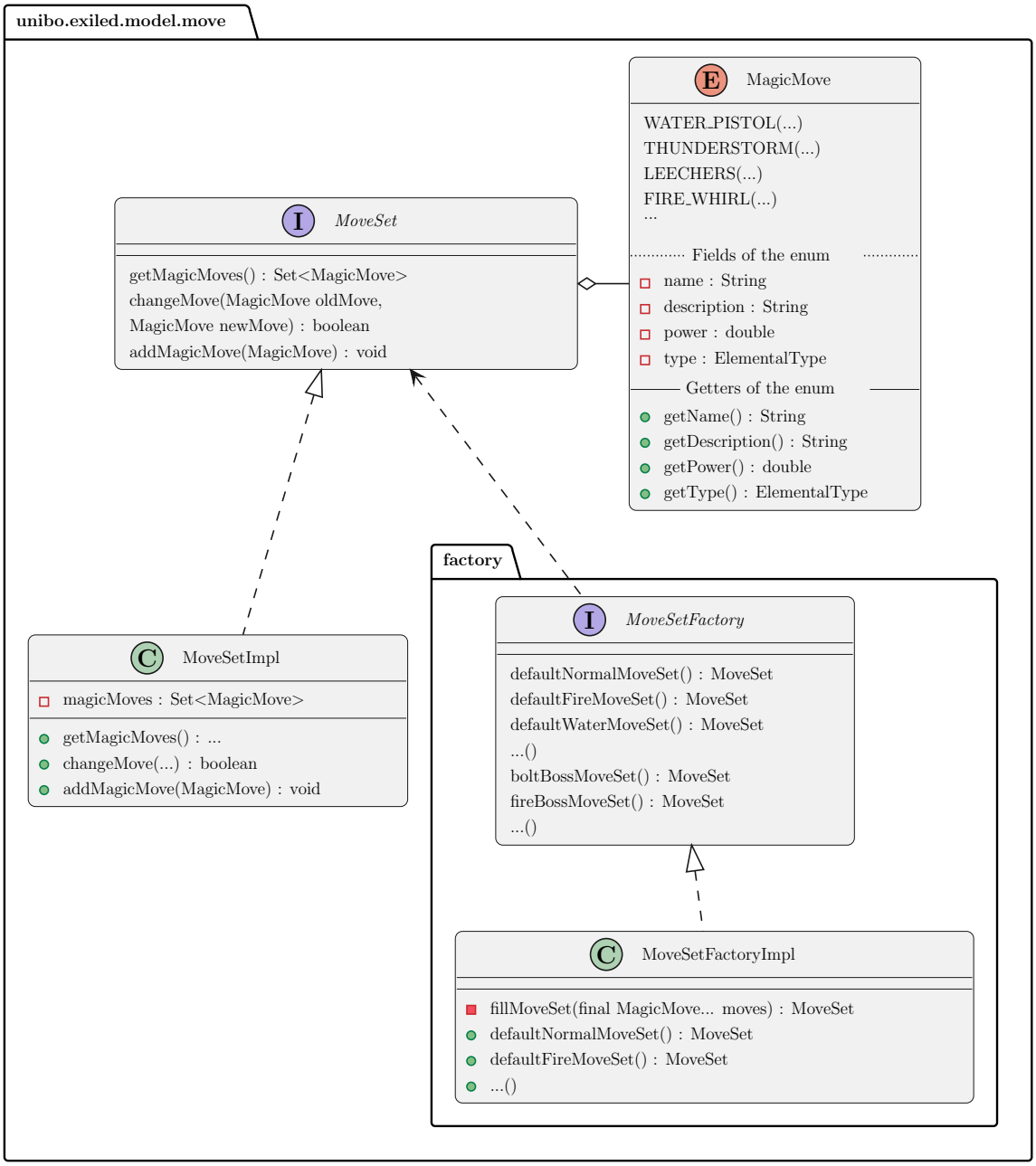

Figura 2.18: Schema UML di come sono gestite le mosse.

Elementi della View

Alcune parti di come vengono mostrati i componenti grafici dell'applicativo sono stati realizzati insieme, per esempio:

- La classe *GameView* spiegata sopra.
- La classe *PlayerClassView* che mostra la finestra di selezione del tipo elementale del giocatore all'inizio del gioco.
- La classe *GameGridView* che mostra la griglia.
- Elementi grafici della View come i bottoni, delle label particolari che compongono l'interfaccia.

Tutta la realizzazione delle sprites dei nemici, del carattere giocante e degli oggetti resa possibile da applicazioni di terze parti come *Gimp*.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per realizzare test automatici è stata utilizzata la libreria JUnit 5 con l'ausilio della libreria di Mocking Mockito. Per la copertura del codice dai test abbiamo fatto uso dello strumento di verifica JaCoCo, le classi che quindi sono state sottoposte a test automatizzati sono:

- *ItemFactoryImpl* $\longrightarrow$ 100% della copertura.
- *MoveSetFactoryImpl* $\longrightarrow$ 100% della copertura.
- *GameLauncher* che è il launcher del gioco: $\longrightarrow$ 100% della copertura.
- *CellType, GameMapImpl, MapModelImpl* $\longrightarrow$ 95% della copertura.
- *EnemyFactoryImpl* $\longrightarrow$ 95% della copertura.
- *Command, MenuImpl, MenuModelImpl, MenuItem* $\longrightarrow$ 94% della copertura.
- ...

Qui verrà allegato il report di JaCoCo con le parti testate per brevità.

the-exiled

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
unibo.exiled.model.character.attributes		100%		n/a	0	27	0	36	0	27	0	5
unibo.exiled.model.move.factory		100%		100%	0	14	0	16	0	13	0	1
unibo.exiled.model.item.factory		100%		n/a	0	4	0	4	0	4	0	1
unibo.exiled		100%		n/a	0	1	0	2	0	1	0	1
unibo.exiled.model.map		97%		73%	6	30	1	62	0	17	0	3
unibo.exiled.model.menu		96%		n/a	1	13	2	32	1	13	0	4
unibo.exiled.model.character.enemy.factory		96%		87%	3	27	3	53	2	20	0	3
unibo.exiled.model.combat		95%		n/a	1	24	1	48	1	24	0	3
unibo.exiled.model.game		93%		n/a	1	6	1	12	1	6	0	1
unibo.exiled.model.character.enemy		92%		n/a	1	8	1	13	1	8	0	2
unibo.exiled.model.item		82%		50%	11	43	14	83	1	31	0	6
unibo.exiled.controller		77%		58%	30	115	54	247	14	91	0	7
unibo.exiled.model.item.utilities		75%		35%	13	23	24	67	7	16	0	3
unibo.exiled.utilities		73%		46%	12	31	22	70	4	20	1	9
unibo.exiled.model.character.player		73%		25%	7	23	12	60	2	17	0	1
unibo.exiled.model.character.enemy.boss		71%		n/a	5	10	5	15	5	10	0	5
unibo.exiled.view		65%		36%	70	110	204	574	38	69	3	10
unibo.exiled.model.character		65%		46%	34	81	54	158	14	49	0	2
unibo.exiled.model.move		62%		0%	19	29	24	60	11	21	0	3
unibo.exiled.view.character		42%		18%	8	11	24	41	1	4	0	1
unibo.exiled.view.items		29%		1%	38	48	86	131	10	20	0	5
Total	2.298 of 8.006	71%	225 of 369	39%	260	678	532	1.784	113	481	4	76

3.2 Note di sviluppo

3.2.1 Luca Casadei

Elencherò solamente un'occasione in cui si presentano queste parti di codice più avanzato, questo non impedisce che siano state usate anche in altre parti del progetto.

- **Espressioni Lambda** e **Optional**: Utilizzati nella classe *CharacterModel* nel metodo *isCellEmpty* per verificare se una cella della mappa contiene qualcuno.
  `https://github.com/luca-casadei/the-exiled/blob/9a97a74beb0dd351cf5b1f72e6d`
  `src/main/java/unibo/exiled/model/character/CharacterModelImpl.java#L99-L100`

- **Record**: Gli attributi sono gestiti come dei *Java Records*, in particolare *AdditiveAttributeImpl* e *MultiplierAttributeImpl*.
  `https://github.com/luca-casadei/the-exiled/blob/9a97a74beb0dd351cf5b1f72e6d`
  `src/main/java/unibo/exiled/model/character/attributes/AdditiveAttributeImpl.java#L8`

- **Libreria Mockito**: Usata per alcuni tests di JUnit, in particolare il test della classe *CharacterControllerImpl*. Il permalink è relativo ad una piccola parte del codice dove viene usata questa libreria.

https://github.com/luca-casadei/the-exiled/blob/506d8a5ed7be7028e52f6b9613c
src/test/java/unibo/exiled/controller/TestCharacterControllerImpl.
java#L49

- ***AtomicReference*, *Optional* e Stream**: Ho utilizzato *AtomicReference⟨MagicMove⟩* per poter avere un riferimento modificabile in maniera atomica in un contesto in cui ho usato ifPresent() di optional per poter utilizzare gli stream. In questo caso mi è servito per poter ritornare il valore desiderato al metodo. https://github.com/luca-casadei/the-exiled/blob/4213b41b965a44db1f1a6ffffef8d3c11aab9434/src/main/java/unibo/exiled/model/move/Moves.java#L29-L34

3.2.2 Francesco Pazzaglia

- **Espressioni Lambda**: Utilizzate per le *ActionListener* dei JButton in *EndGameView*. https://github.com/luca-casadei/the-exiled/blob/a59eb1a55ef30e4a8236a7833d634999110a64a6/src/main/java/unibo/exiled/view/EndGameView.java#L62

- **Optional**: Utilizzati nel *Player*(metodo *getNewMove()*) per la generazione di una nuova mossa. https://github.com/luca-casadei/the-exiled/blob/b5d735949ba7e010ffae414b746d4d7f464148ba/src/main/java/unibo/exiled/model/character/player/PlayerImpl.java#L127

- **Stream**: Utilizzati nei Controller per controllare che il *Player* abbia vinto il gioco. https://github.com/luca-casadei/the-exiled/blob/b5d735949ba7e010ffae414b746d4d7f464148ba/src/main/java/unibo/exiled/controller/CharacterControllerImpl.java#L197

3.2.3 Marco Magnani

- **Stream e Optional**: Utilizzati sia nell' *ItemContainer* che nelle *Moves* per ricavare esempio un item in base al tipo. https://github.com/luca-casadei/the-exiled/blob/71356cacbbef08c33f96d68802a
src/main/java/unibo/exiled/model/item/utilities/ItemsContainer.
java#L76C7-L76C8

- **Stream**: In *CharacterModelImpl* utilizzato per evitare la compenetrazione di enemy all'interno della mappa, aggiunto nel commit con hash: f25b577fb319fbbcb38154b3f2b8a3233d5c561b https://github.com/luca-casadei/the-exiled/blob/71356cacbbef08c33f96d68802a

src/main/java/unibo/exiled/model/character/CharacterModelImpl.
java#L159C1-L160C1

- **Optional**: In *ElementalType* e *ItemNames* il metodo per prendere le relative immagini ritorna un Optional per gestire il caso in cui la mossa non sia presente nel progetto
https://github.com/luca-casadei/the-exiled/blob/71356cacbbef08c33f96d68802a3
src/main/java/unibo/exiled/model/item/utilities/ItemNames.java#
L83C1-L84C55

- **Stream**: Nella *ItemsModel* utilizzata per la conversione da *Map<Item,Integer>* a *Map<String,Integer>* ed anche per filtrare gli oggetti in base al nome, aggiunto nel GameModel commit con hash: 2e72b6911f473bfa3ee8685cd61f14ae8922f936
https://github.com/luca-casadei/the-exiled/blob/71356cacbbef08c33f96d68802a3
src/main/java/unibo/exiled/model/item/ItemsModelImpl.java#L28C1-L28C90

- **Stream**: In *InventoryView* quando vengono aggiornati i bottoni contenenti gli item li ordino in modo da migliorarne la visualizzazione.
https://github.com/luca-casadei/the-exiled/blob/b0f1e9a189dbf365a0413e72494
src/main/java/unibo/exiled/view/InventoryView.java#L106C1-L108C47

3.2.4 Manuel Baldoni

- **Stream**: Utilizzato nel metodo *getEnemyRandomMoveName*. Per avere il nome di una mossa randomica.
https://github.com/luca-casadei/OOP23-the-exiled/blob/2607859d3adef7b5c2309
src/main/java/unibo/exiled/controller/CombatControllerImpl.
java#L137C5-L148C6

- **Stream**: Utilizzato nel metodo *attack*. Per trovare la mossa effettuata dall'attaccante dal suo moveset.
https://github.com/luca-casadei/OOP23-the-exiled/blob/2607859d3adef7b5c2309
src/main/java/unibo/exiled/controller/CombatControllerImpl.
java#L156C2-L259C6

- **Optional ed Espressioni Lambda**: Utilizzate per le verificare la presenza dell'item droppato dal nemico e per aggiungerlo all'inventario. https://github.com/luca-casadei/OOP23-the-exiled/blob/
2607859d3adef7b5c23090199e0807cd852fc256/src/main/java/unibo/
exiled/controller/CombatControllerImpl.java#L203C9-L214C10

- **Espressione Lambda**: Utilizzata per gestire il listener del comando del menu. https://github.com/luca-casadei/OOP23-the-exiled/

blob/2607859d3adef7b5c23090199e0807cd852fc256/src/main/java/unibo/exiled/view/MenuView.java#L67C9-L91C12 https://github.com/luca-casadei/OOP23-the-exiled/blob/2607859d3adef7b5c23090199e0807cd852fc256/src/main/java/unibo/exiled/view/CombatView.java#L95C17-L99C20

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Luca Casadei

Personalmente ritengo che il mio contributo all'interno del team di sviluppo sia stato importante, sia per la parte realizzativa che per quella decisionale, ma non più importante di quella degli altri componenti del gruppo, con i quali è stato possibile lo confrontare le proprie idee di realizzazione, la corretta suddivisione dei compiti e uno sviluppo continuativo dell'applicazione anche se non necessariamente tutti presenti allo stesso momento o nello stesso luogo. Anche il caricamento del codice in parallelo con sistemi di controllo della versione è avvenuto in maniera secondo me pulita, perché tutti sono stati in grado di adottare questi strumenti in modo efficace senza creare troppi intoppi. Siccome la progettazione finale del progetto secondo me risulta abbastanza scalabile, non è esclusa l'ipotesi di arricchire ulteriormente il contenuto del gioco e le sue componenti, anche prendendo in forte considerazione la valutazione dei docenti.

4.1.2 Francesco Pazzaglia

Nonostante le parti complesse del progetto e il fatto di sviluppare un progetto in team mi avesse inizialmente intimorito, sono soddisfatto di come siamo riusciti a completare e consegnare il gioco. Di solito sono abituato a lavorare da solo, ma ho trovato la collaborazione del team piuttosto soddisfacente. Lo scambio di idee e il confronto ci hanno aiutato a coordinarci efficacemente, individuando i metodi e le classi necessarie per varie parti del progetto. Lavorare in team è stato molto formativo e mi ha permesso di apprezzare come

si possa realizzare un progetto insieme, nonostante stili di programmazione talvolta diversi.

4.1.3 Marco Magnani

Il progetto è stato svolto con colleghi con cui ho avuto modo di collaborare già in precedenza alle superiori, quindi sapevo già prima cosa aspettarmi. La centralità della collaborazione e della discussione di idee è stata una costante durante lo sviluppo di questo progetto. Nel complesso, ritengo che il nostro lavoro sia stato soddisfacente, e in particolare, credo di aver gestito una buona parte del progetto sfruttando le competenze acquisite durante le lezioni.

4.1.4 Manuel Baldoni

Credo che tutti i componenti del gruppo siano individualmente molto preparati a livello di competenze di sviluppo e credo che siamo riusciti a collaborare in maniera efficace. Personalmente credo di aver svolto la mia parte di progetto in maniera efficace ed efficiente. Il mio contributo nelle altri parti del progetto sicuramente non è stato nel volume degli altri componenti del gruppo, tuttavia penso che alcuni miei interventi siano stati fondamentali per la realizzazione del progetto.

4.2 Difficoltà incontrate e commenti per i docenti

Nessun commento o difficoltà da segnalare.

Appendice A

Guida utente

Per la maggior parte delle azioni compibili nel gioco sono presenti bottoni sull'interfaccia, non è quindi necessario conoscere nessun particolare comando. Ne sono stati aggiunti alcuni aggiuntivi nell'inventario per l'apertura e la chiusura, come sottodescritto.

A.1 Movimento

Per muoversi sulla mappa a celle si usano i comandi:

- *W* per andare su.
- *S* per andare giù.
- *A* per andare a sinistra.
- *D* per andare a destra.

A.2 Inventario

Si può aprire l'inventario con *E* e richiuderlo con *ESC*

A.3 Menu

Si può aprire il menu di gioco utilizzando *ESC* quando si visualizza la mappa, e richiuderlo in maniera analoga.

Appendice B

Esercitazioni di laboratorio

Elenco dei Permalink delle consegne nel forum studenti del modulo di laboratorio.

Consegne

B.0.1 luca.casadei27@studio.unibo.it

- Laboratorio 06: `https://virtuale.unibo.it/mod/forum/discuss.php?d=146511#p208401`
- Laboratorio 07: `https://virtuale.unibo.it/mod/forum/discuss.php?d=147598#p209323`
- Laboratorio 08: `https://virtuale.unibo.it/mod/forum/discuss.php?d=148025#p209782`
- Laboratorio 09: `https://virtuale.unibo.it/mod/forum/discuss.php?d=149231#p211336`
- Laboratorio 10: `https://virtuale.unibo.it/mod/forum/discuss.php?d=150252#p212475`