Relazione progetto

UnOriginal.RPG

Corso: Programmazione a Oggetti

AA: 2015/16

Maltoni Niccolò

Naldini Federico

Rasi Stefano

Semprini Luca

Sommario

Questo documento rappresenta la relazione finale per il software “UnOriginal.RPG”, sviluppato per il progetto del corso Programmazione a Oggetti nell’anno accademico 2015/2016.
Questa relazione è inserita all’interno di un repository BitBucket che contiene, oltre alla relazione stessa, l’applicazione salvata in estensione Jar e il codice sorgente dell’applicativo.

Link al repository: https://www.bitbucket.org/NiccoMlt/oop15-maltoni-niccol-naldini-federico-semprini-luca-rasi

Indice

1 Analisi

1.1 Requisiti……4
1.2 Problema……4

2 Design

2.1 Architettura……6
2.2 Design dettagliato……7

3 Sviluppo

3.1 Testing Automatizzato……27
3.2 Divisione dei compiti e metodologia di lavoro……27
3.3 Note di sviluppo……27

4 Commenti Finali

4.1 Conclusioni e lavori futuri……28
4.2 Difficoltà incontrate e commenti per i docenti……29
4.3 Appendice Guida Utente……29

Capitolo 1

Analisi

1.1 Requisiti

Il software “UnOriginal.RPG” si pone come obbiettivo di realizzare un gioco di genere Classic-RPG (Role Play Game).
Il genere Classic-Role-Play game ha come scopo quello di proporre al giocatore un’esperienza molto dinamica e malleabile: la struttura del gioco viene infatti disposta in modo che il giocatore possa avere molta libertà decisionale all’interno del mondo di gioco.
Per fornire al giocatore questa libertà, il genere GDR pone alcune regole non scritte per la realizzazione delle sue strutture di gioco:

- Un mondo di gioco totalmente aperto e interconnesso, dove il giocatore sia libero di muoversi senza eccesivi vincoli.
- Un sistema di differenziazione dei personaggi basato su un insieme di statistiche, divise tra difese e attacchi, che possa rendere il singolo personaggio più o meno adatto a certe situazioni.
- Possibilità di creazione e personalizzazione di un numero limitato di personaggi che diventeranno il team controllato dal giocatore.
- Un sistema di battaglia strutturato a turni, dove l’insieme dei personaggi controllato dal giocatore si trova ad affrontare creature ostili presenti nel mondo di gioco.
- Difficoltà crescente all’interno del gioco, che rende le creature più forti man mano che si procede nel gioco.
- Presenza di armi, armature e aumenti di livello in modo da potenziare i personaggi e permettere al giocatore di affrontare sfide sempre più ardue.
- Sistema di salvataggio che permetta di interrompere una partita e riprenderla in un secondo momento.

1.2 Analisi e modello del dominio

UnOriginal.RPG dovrà essere in grado di soddisfare tutti i requisiti di un gioco di ruolo, dovrà cioè permettere al giocatore la creazione dei membri della sua squadra, o Party che dir si voglia, e il loro movimento all’interno delle mappe di gioco; dovrà poi realizzare una mappa di gioco che possa essere liberamente esplorata dal giocatore, facendo attenzione a dividere le aree sicure da quelle in cui appaiono mostri.

Dovrà inoltre essere implementato un sistema di battaglia a turni che permetta al giocatore di poter indirizzare i propri attacchi verso specifiche creature ostili, a ogni vittoria sarà attribuito un punteggio in esperienza, che porterà i personaggi a aumentare di livello e potenziare le loro caratteristiche.
Vanno infine modellati equipaggiamenti e oggetti, e il loro utilizzo tramite appositi menù di gioco.
In figura 1.1 è possibile vedere lo schema scheletro del modello.

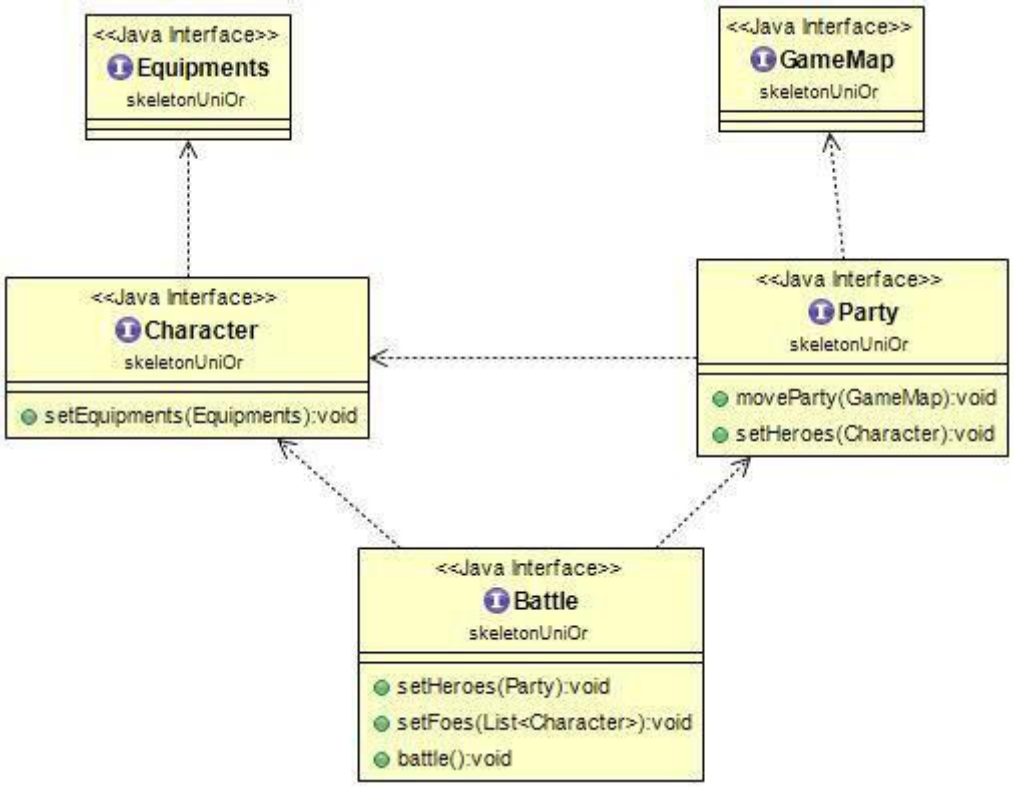
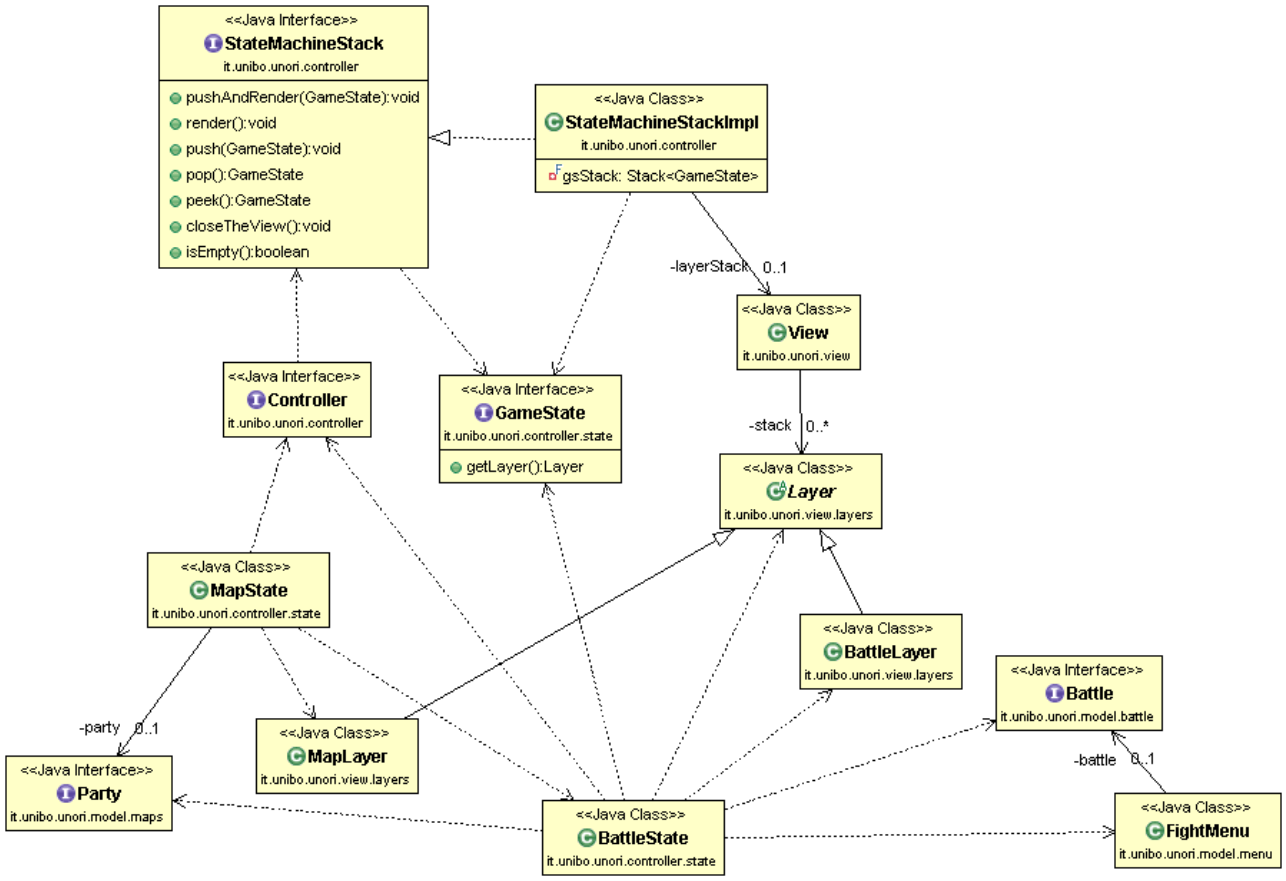


Figura 1.1: Schema Uml dell'analisi del problema

Capitolo 2

Design

2.1 - Architettura



Il core che costituisce l'architettura del gioco ruota intorno alla battaglia e alla mappa, i due stati principali in cui si svolge il gioco. La mappa è modellata dal *MapState* che utilizza l'oggetto *Party* del model per tutta la componente logica della gestione della mappa attuale, dello spostamento e dell'interazione con le celle adiacenti, e l'oggetto *MapLayer* per la componente grafica. Sarà il controller a gestire il passaggio al *BattleState* per via dell'incontro casuale di un gruppo di nemici o di una sfida con un boss.

Il secondo stato più importante è dunque la battaglia, che coinvolge attraverso *BattleState* diverse classi di modello e di visualizzazione per fornire personaggi e oggetti, logiche di battaglia e attacchi e sprite grafici per una corretta visualizzazione.
Abbiamo implementato questo tipo di architettura attraverso il pattern MVC, e anche se dobbiamo riconoscere che a seguito della conclusione dell'implementazione possiamo dire che poteva essere implementata una suddivisione leggermente migliore, in quanto la modifica o la completa sostituzione dell'implementazione di alcune classi di Model o View comporterebbe modifiche anche negli altri componenti. Nonostante ciò, vista la dimensione del progetto, la difficoltà di gestione della suddivisione del lavoro tra 4 sviluppatori per 3 componenti e il limite di 100 ore a testa, ci riteniamo sufficientemente soddisfatti della struttura del nostro progetto.

2.2 – Design dettagliato

Model parte 1: Mappe, personaggi e equipaggiamenti.

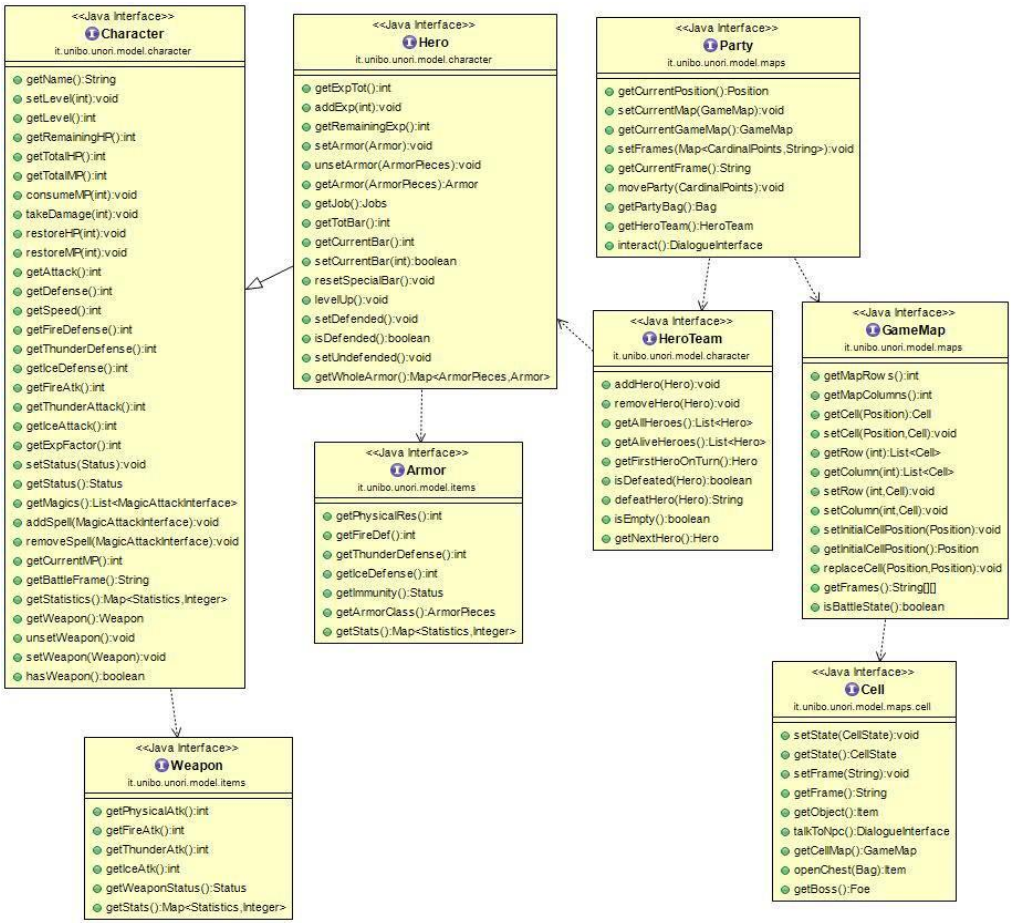


Figura 2.2.M1.1 Schema UML generale della sotto parte 1 di model.

Lo scopo della sotto parte 1 del Model era quello di implementare le mappe di gioco, la modellazione di generici personaggi di gioco e infine la realizzazione di un equipaggiamento sia offensivo che difensivo.
Il nodo centrale di questa parte di model è rappresentato dall'interfaccia *party*, che racchiude in sé sia la componente di movimento su una mappa, sia la gestione, come lista di eroi, dei personaggi del giocatore.

Model parte 1: Mappe.

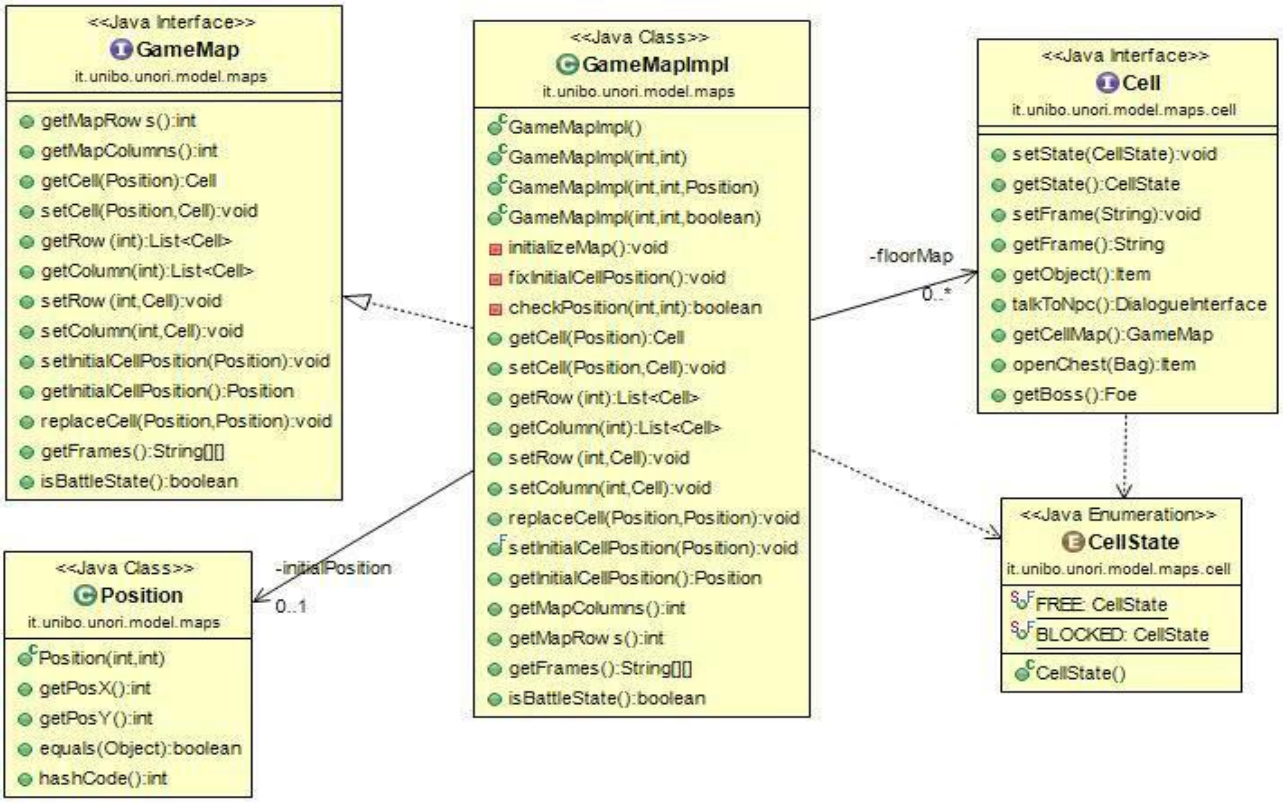


Figura 2.2.M1.2 Schema UML delle Mappe

L'intera modellazione delle mappe è racchiusa in un package specifico chiamato "*model.maps*", la cui classe fondamentale risulta essere *GameMapImpl*.
GameMapImpl si basa sulla classe Position per descrivere le posizioni in termini di coordinate all'interno della mappa, e sull' interfaccia Cell per designare le singole porzioni di mappa, descrivendone caratteristiche come l'accessibilità da parte del giocatore o la presenza di eventuali oggetti, dialoghi, nemici.
Conseguentemente *GameMapImpl* ha due importanti campi privati: una matrice di oggetti Cella che rappresentano la scacchiera su cui il giocatore sarà libero di muoversi e un oggetto di tipo posizione, che designa la posizione della cella di partenza della mappa, *GameMap* ha poi alcuni metodi per gestire sia le singole celle che intere righe o colonne della matrice, in modo da rendere più semplice il settaggio delle singole celle, dispone poi di alcuni metodi privati per settare

automaticamente la cella iniziale, qualora un cambiamento rendesse non idonea la cella attuale, e per il controllo dei valori di input.

Model parte 1: Personaggi.

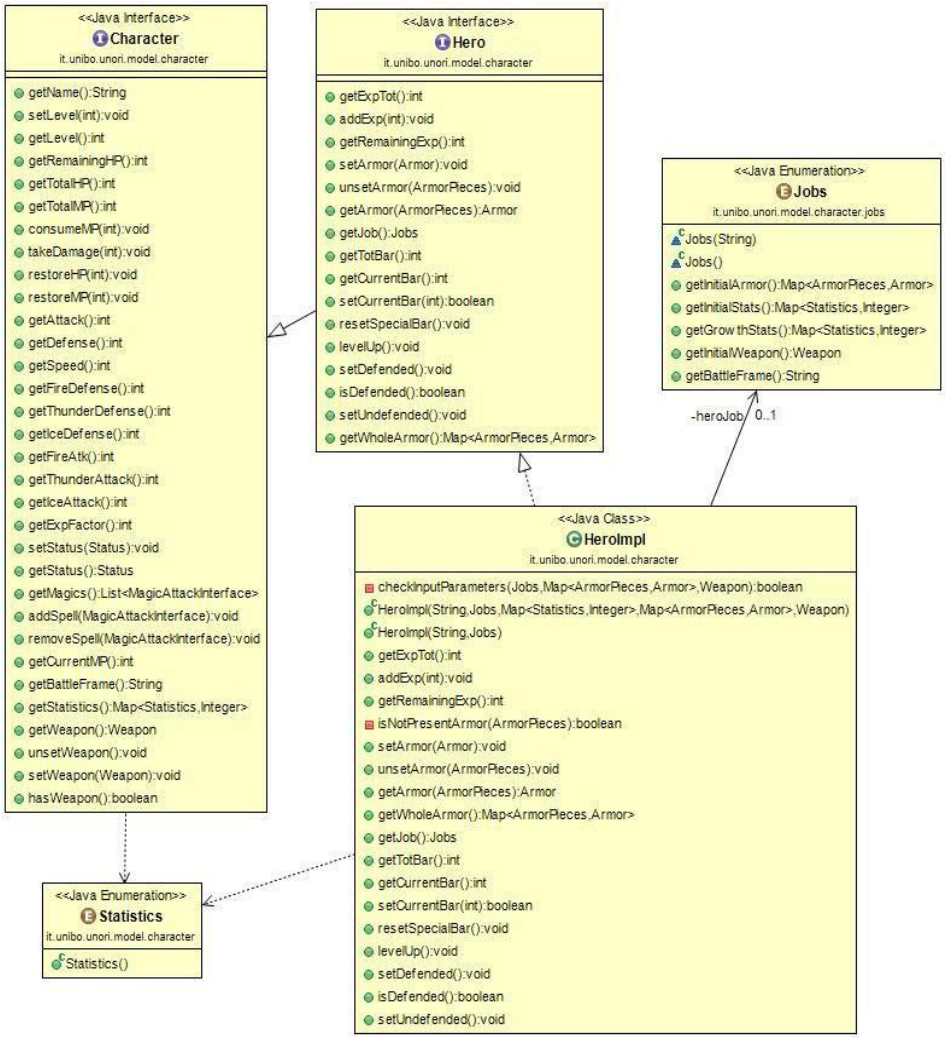


Figura 2.2.M1.3: Schema UML dei personaggi

L'interfaccia *Character* si occupa della modellazione delle caratteristiche base di un qualsiasi personaggio di gioco, controllato dal giocatore o nemico che sia, appoggiandosi all'Enumerazione *Statistics*, dove sono salvati i vari tipi di caratteristica; qualunque implementazione di *Character* si basa su una mappa dove per ogni statistica è salvato il corrispondente valore.
Character dispone di una serie di metodi per accedere alle statistiche e sottrarre o aggiungere punti a statistiche variabili come i punti ferita o i punti magia.

Per la modellazione dei personaggi del giocatore, è stata creata l'interfaccia *Hero* che estende l'interfaccia *Character* aggiungendo metodi per la gestione dell'armatura e della barra dell'esperienza, inoltre è presente un campo di tipo *Job*, che può essere scelto tra i valori dell'Enumerazione Jobs, che determina automaticamente statistiche inziali, statistiche di crescita, armatura e arma iniziali e infine lo sprite da utilizzare in battaglia.

Model parte 1: Party.

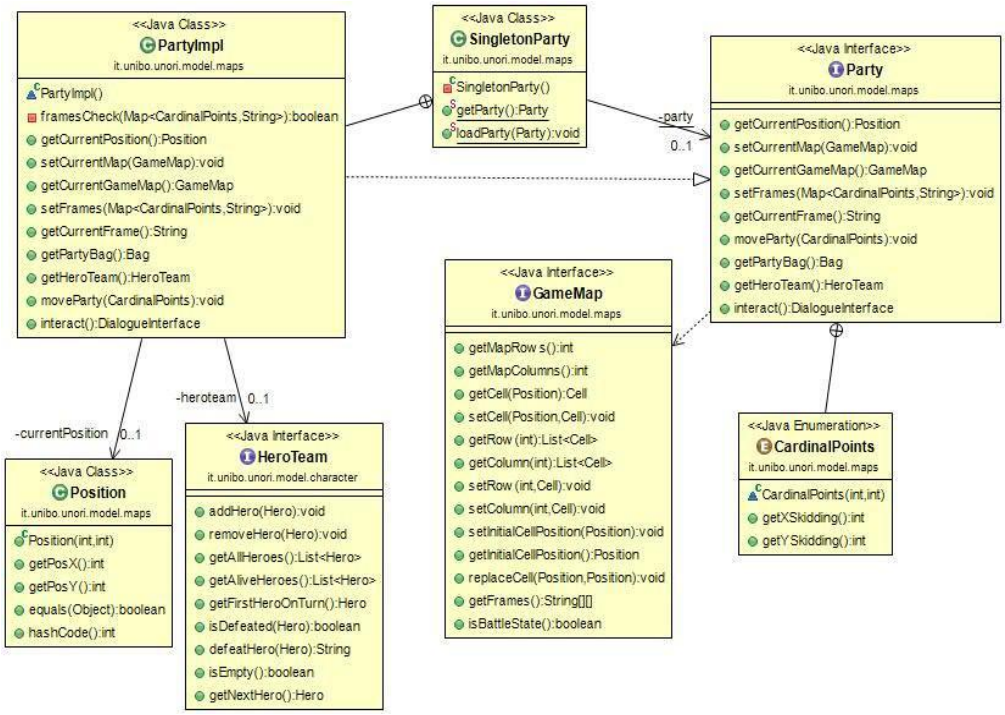


Figura 2.2.M1.4 Schema UML del Party

L'interfaccia party e la sua implementazione sono la parte più importante di questa parte di model, in quanto tutti gli elementi descritti nella parte di Mappe e Personaggi si fondono qui. Innanzitutto è stato scelto di implementare la strategy Singleton per l'implementazione dell'interfaccia party, poiché l'oggetto party sarebbe stato unico per ogni partita di gioco. *PartyImpl* contiene al suo interno un campo *GameMap*, un campo *HeroTeam* (Una struttura per gestire la lista dei personaggi del giocatore, approfondito in Model2) e un campo *Bag* (funge da inventario per gli oggetti di gioco, approfondito in Model2)
Fondamentali sono il metodo *move()* e *interact()* : *move()*, prendendo in input un oggetto dell'enumerazione *CardinalPoints*, permette di girare il personaggio nella direzione desiderata, eventualmente avanzare e cambiare anche mappa; *interact()* chiama in ordine una serie di metodi della cella davanti al personaggio, che lanciano eccezioni se la cella non è istanza di una specifica classe, questi metodi permettono di visualizzare il dialogo con un personaggio sulla mappa, aggiungere un oggetto all'inventario, o iniziare una battaglia con un nemico.

Model parte 1: Equipaggiamenti

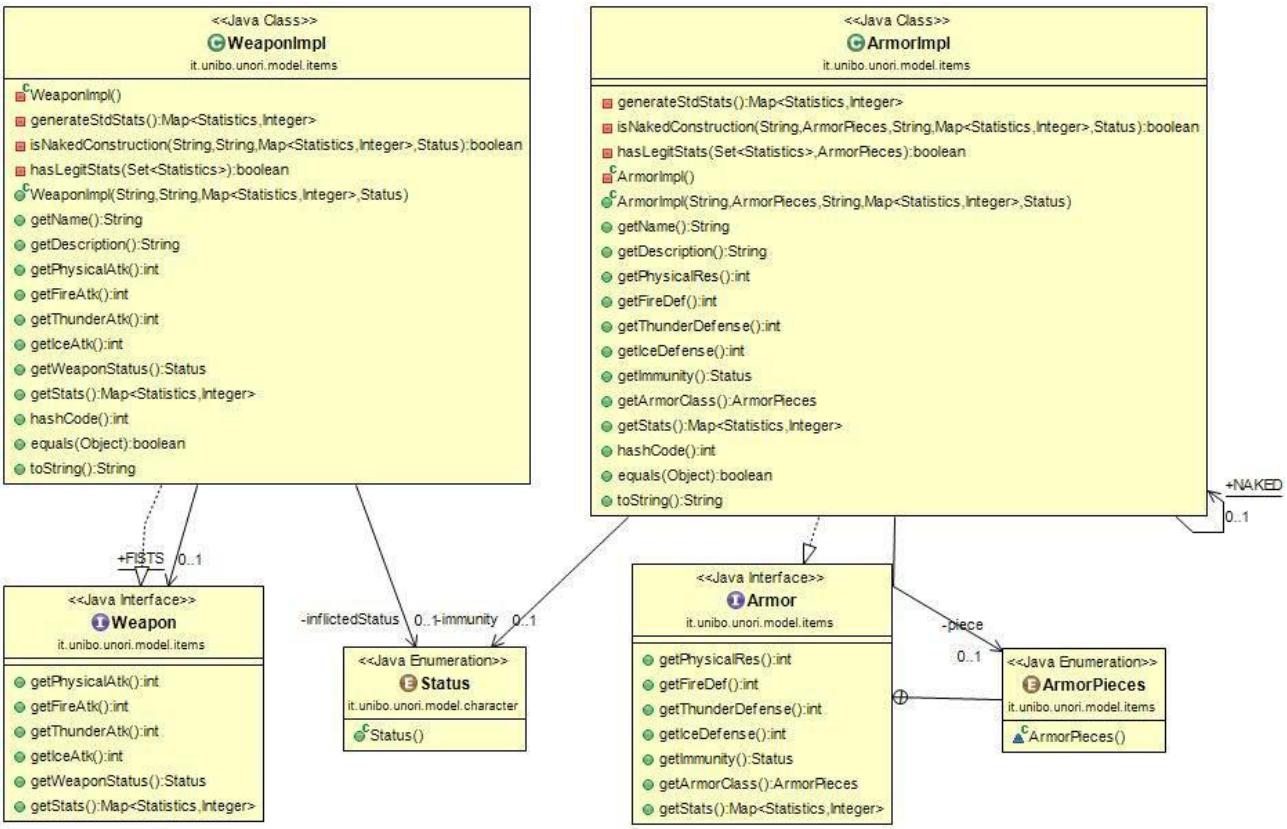


Figura 2.2.M1.5: Equipaggiamenti

L'ultima parte della prima parte di Model copre la realizzazione degli equipaggiamenti di attacco e difesa, la cui implementazione risulta molto simile.
Entrambe le classi dispongono di una mappa di statistiche e corrispettivi valori, che vengono controllati e settati al momento della costruzione, inoltre possiedono una loro implementazione statica realizzata con costruttore vuoto che rappresenta l'assenza di arma o armatura, inoltre possiedono un campo dove può essere settato uno status che l'oggetto infligge/difende.
Le armature hanno una parte aggiuntiva rispetto alle armi: siccome è stato deciso di rendere possibile l'equipaggiamento di più pezzi d'armatura in contemporanea, ogni oggetto di *ArmorImpl* contiene il riferimento a un valore dell' enumerazione *ArmorPieces*, che indentifica la tipologia di pezzo dell'armatura, diverse tipologie possono essere equipaggiate contemporaneamente, ma per ogni tipologia solo un pezzo alla volta può essere equipaggiato.

Model Parte 2: Battaglia, Menu, Personaggi e Pozioni.

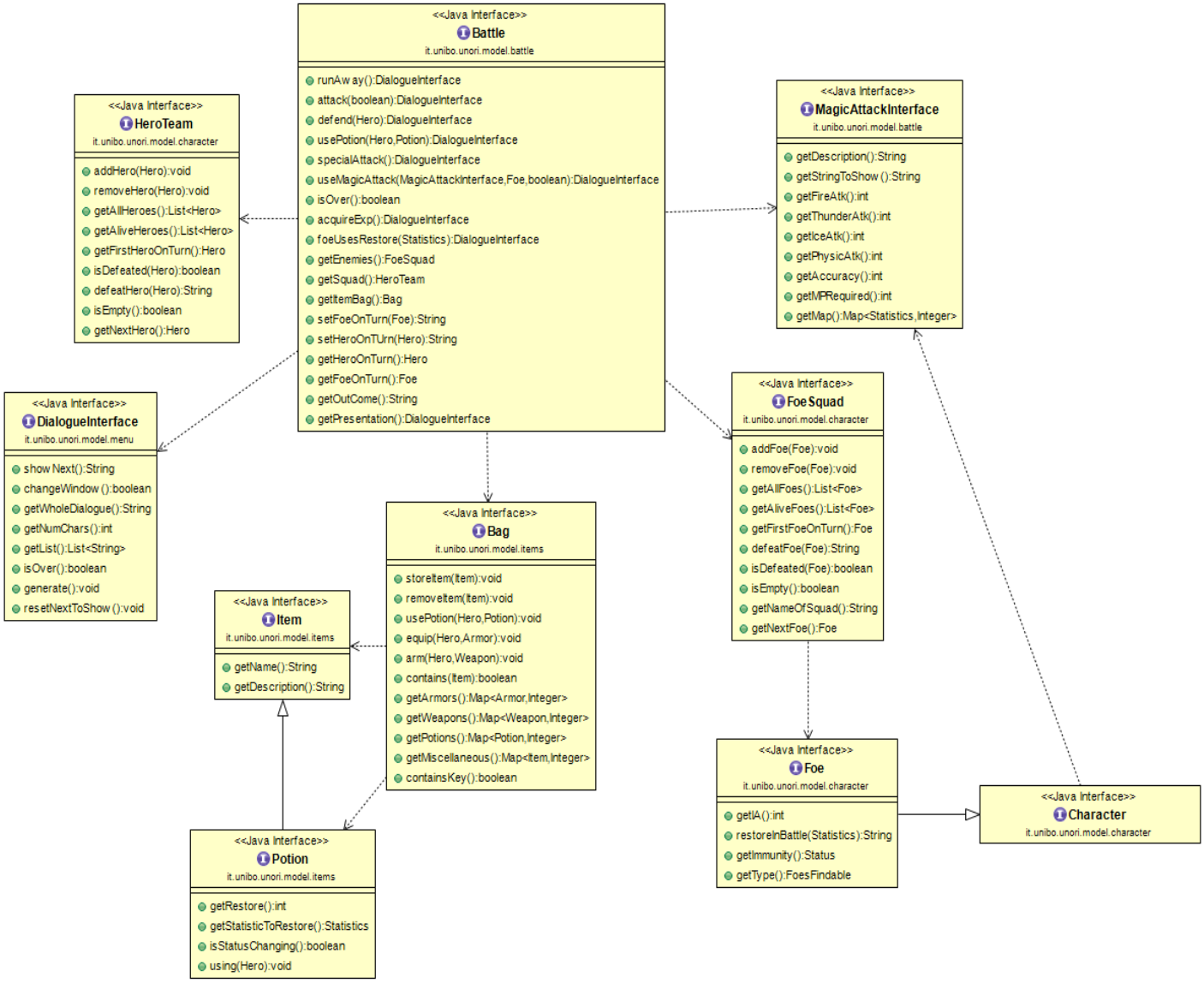


Figura 2.2.M2.1 – Schema UML generale della sotto parte 2 di model.

Le problematiche di progettazione affrontate nella sotto parte 2 del Model riguardavano in gran parte le logiche da adottare nella Modalità Battaglia, oltre che la realizzazione di un'astrazione che permettesse di utilizzare Nemici ed Eroi all'interno della stessa. Un altro obiettivo era quello di concentrarsi su Strumenti rigeneranti (Pozioni), attacchi magici e sulla realizzazione di uno scheletro per i Menu, che potesse essere utile al Controller.

Come si può facilmente intuire da questo schema il punto focale di questa parte di Model risiede nell'interfaccia Battle, che consiste in un raccordo tra la modellazione di logiche statistiche, strumenti utilizzabili, magie scagliabili e menu di gioco.

Model Parte 2: Battaglia.

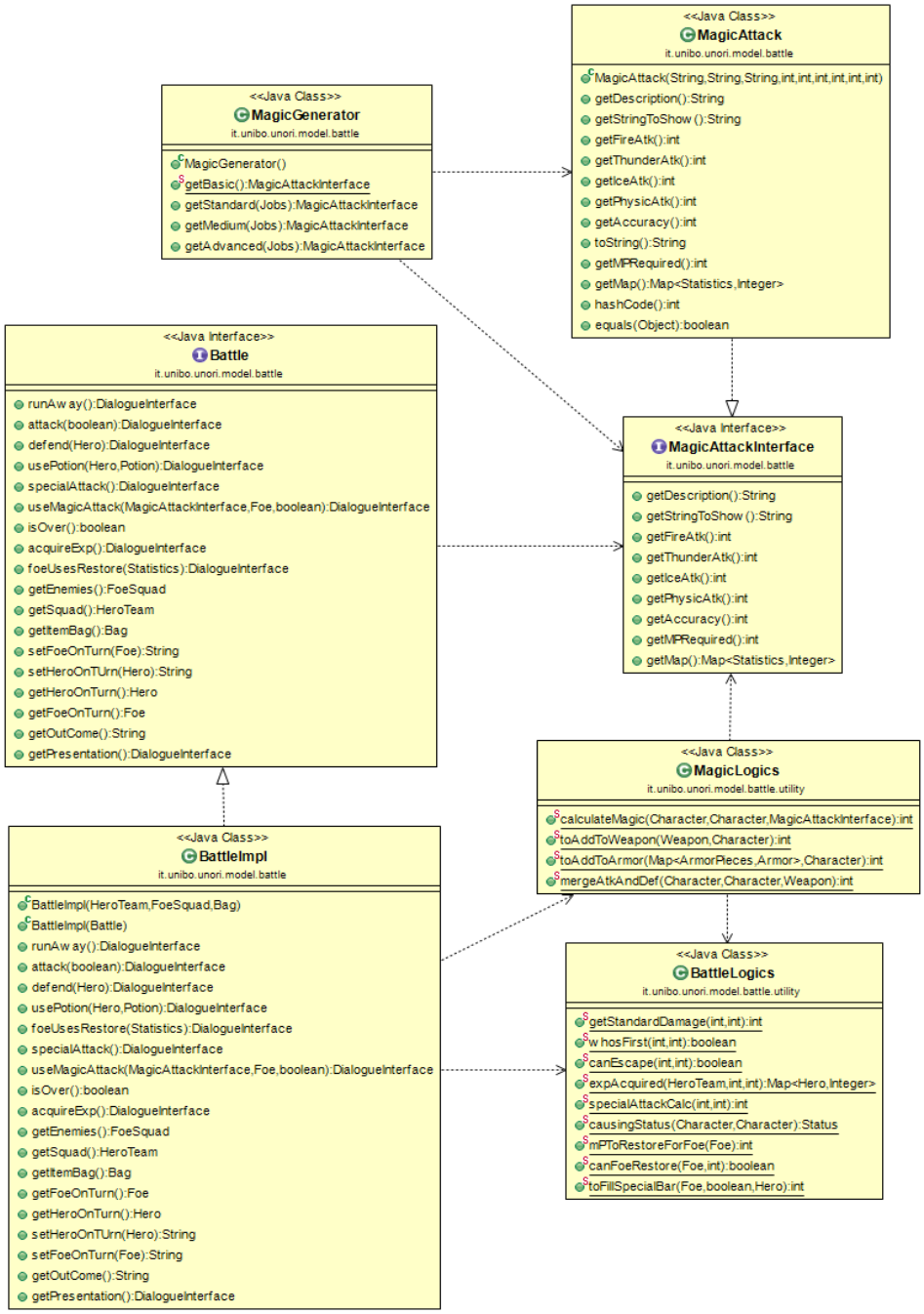


Figura 2.2.M2.2 – Schema UML relativo alla Battaglia.

Lo schema in figura rappresenta il contenuto del package ***it.unibo.unori.model.battle***, in cui troviamo la classe *BattleImpl*, implementazione dell'interfaccia Battle, che consiste nel punto centrale di questa parte di model (come si può anche evincere dalla figura 2.2.M2.2). La classe *BattleImpl* delega le più complesse operazioni e decisioni a due *classi di utility*:

- ***BattleLogics***: che ha il compito di calcolare danni standard o speciali, di stabilire chi possa muovere per primo nel turno di battaglia, di calcolare il quantitativo di punti Esperienza acquisiti al termine della Battaglia dai vari Eroi della squadra, e molto altro.
- ***MagicLogics*****:** che permette di sfruttare al meglio la differenziazione delle Statistiche (proprie di attacchi magici, armi, armature o Personaggi stessi) nei tre tipi Fuoco, Fulmine e Ghiaccio, oltre che operare un *merge* tra le statistiche di difesa e di attacco per calcolare al meglio e in modo "fair" il danno da infliggere in battaglia.

La classe *MagicAttack* implementa *MagicAttackInterface*, e modella un attacco magico, definito da un nome, una descrizione e la quantità di punti per le quattro statistiche di attacco. Questi attacchi sono generati dalla classe di *factory* ***MagicGenerator***, che, a seconda della tipologia (*Job*) dell'Eroe fornisce magie differenti, e di potenza incrementale.

Model Parte 2: Personaggi.

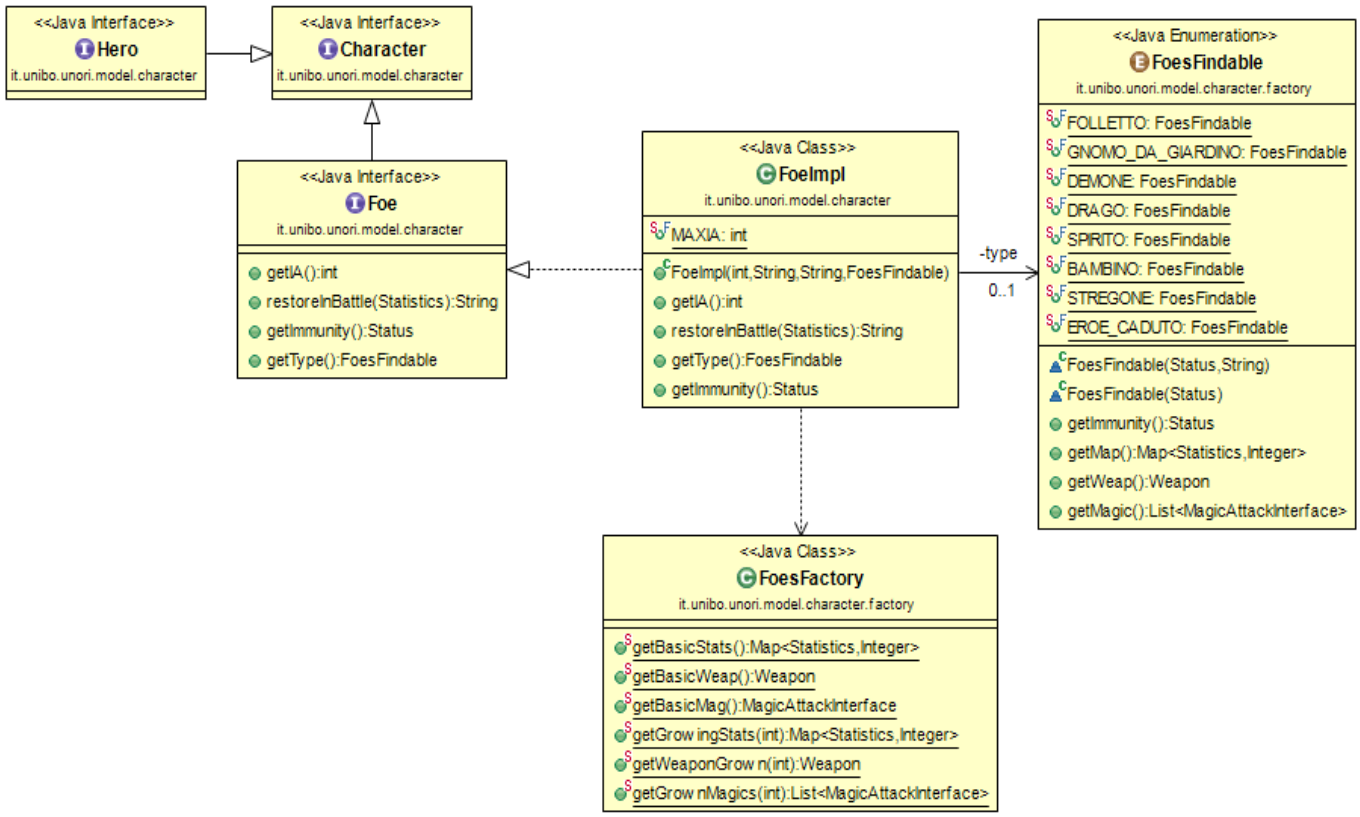


Figura 2.2.M2.3 – Schema UML relativo ai Personaggi, focalizzato sui Nemici.

L'interfaccia *Character* (già presentata nella parte 1 di Model) è estesa fondamentalmente da due interfacce "figlie": una è Hero, anche questa già vista nella parte 1, e l'altra è *Foe*.
L'interfaccia *Foe* rappresenta una astrazione per un generico Nemico, trovabile all'interno del gioco, in particolare nel Dungeon. Un Nemico può brandire un'arma e scagliare attacchi magici, come un Hero, ma non è in grado di difendersi con un'Armatura né, ovviamente, di accrescere il proprio livello acquisendo punti esperienza. Ogni *Foe* è caratterizzato da un *tipo*, descritto nell'Enumerazione FoesFindable e da **un'immunità**, ovvero uno Status al quale risulta essere immune, in modo tale da renderlo più competitivo in Battaglia e compensare l'assenza di un'Armatura. Altra caratteristica propria di un *Foe* è la sua IA (Intelligenza Artificiale), ovvero un valore che può oscillare da 1 a 10 (range implementato in modo che possa essere espandibile in caso di evoluzioni future del gioco), e che lo rende via via meno battibile: a seconda di questo valore, infatti, il Nemico può rigenerare i suoi Health Points o Mana Points più o meno spesso all'interno della battaglia, grazie al metodo *restoreInBattle(Statistics)*.
Le statistiche, le magie e le armi che un *Foe* può possedere sono determinate nella classe *FoesFactory*, che costruisce sia le statistiche iniziali, sia quelle al variare dell'IA, quindi per nemici più forti, permettendo loro di brandire armi e scagliare attacchi magici sempre più potenti.

Model Parte 2: Personaggi in Battaglia e Borsa.

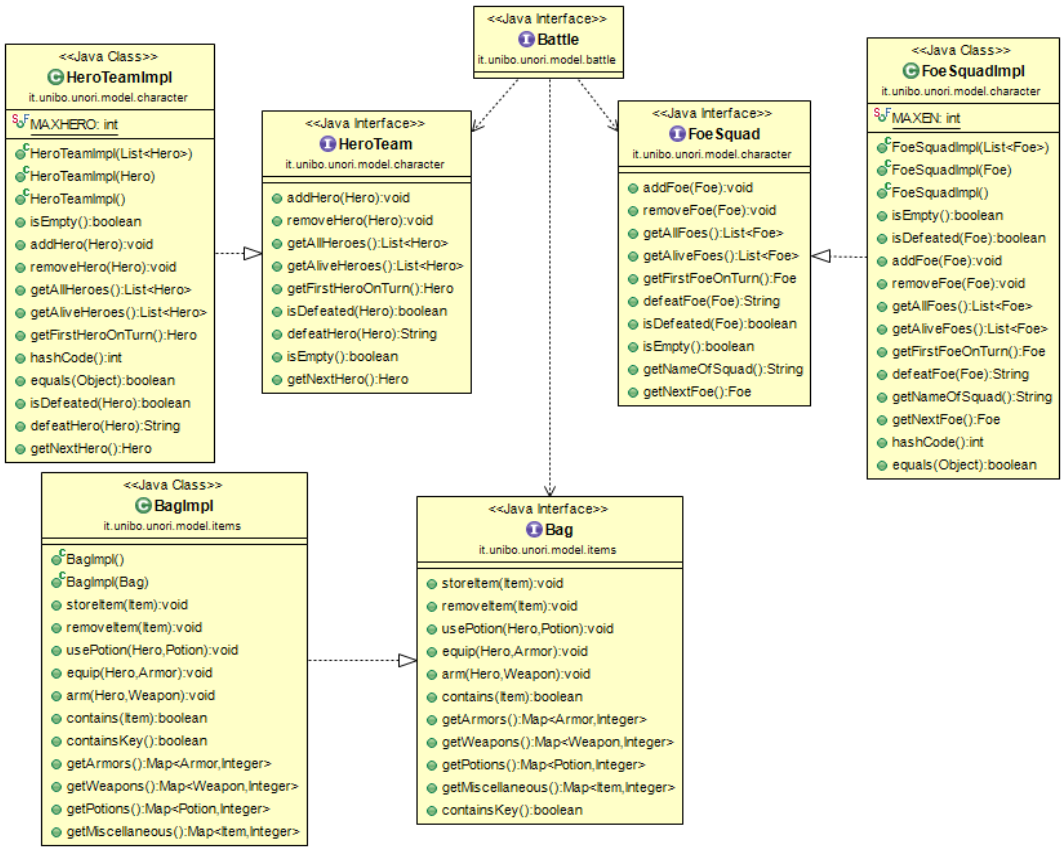


Figura 2.2.M2.4 – Schema UML relativo ai Personaggi in Battaglia e alla Borsa.

Nella modalità Battaglia, i Personaggi, sono memorizzati all'interno della classe *BattleImpl* grazie a due astrazioni: *HeroTeam* e *FoeSquad*. Queste interfacce permettono di memorizzare un numero definito di Personaggi, da coinvolgere proprio all'interno della battaglia; questo numero è definito (rispettivamente per le due classi) dal campo *MAXEN* e *MAXHERO* e può essere cambiato facilmente, per esempio in vista di espansioni del gioco, aumentando i personaggi coinvolti in battaglia. Le implementazioni di queste due interfacce hanno molte altre funzionalità, tra cui rimuovere dalla squadra un personaggio sconfitto in battaglia, restituire la lista dei personaggi ancora in vita o restituire il prossimo personaggio da schierare in caso di sconfitta del personaggio di turno.
Molto importante è, inoltre, l'interfaccia *Bag*, che rappresenta la modellazione di una Borsa di Strumenti, che può contenere Pozioni, Armi e Armature. Essa contiene i metodi necessari per usare questi oggetti, oltre che per aggiungerli o rimuoverli dalla Borsa stessa.

Model Parte 2: Strumenti rigeneranti - Pozioni.

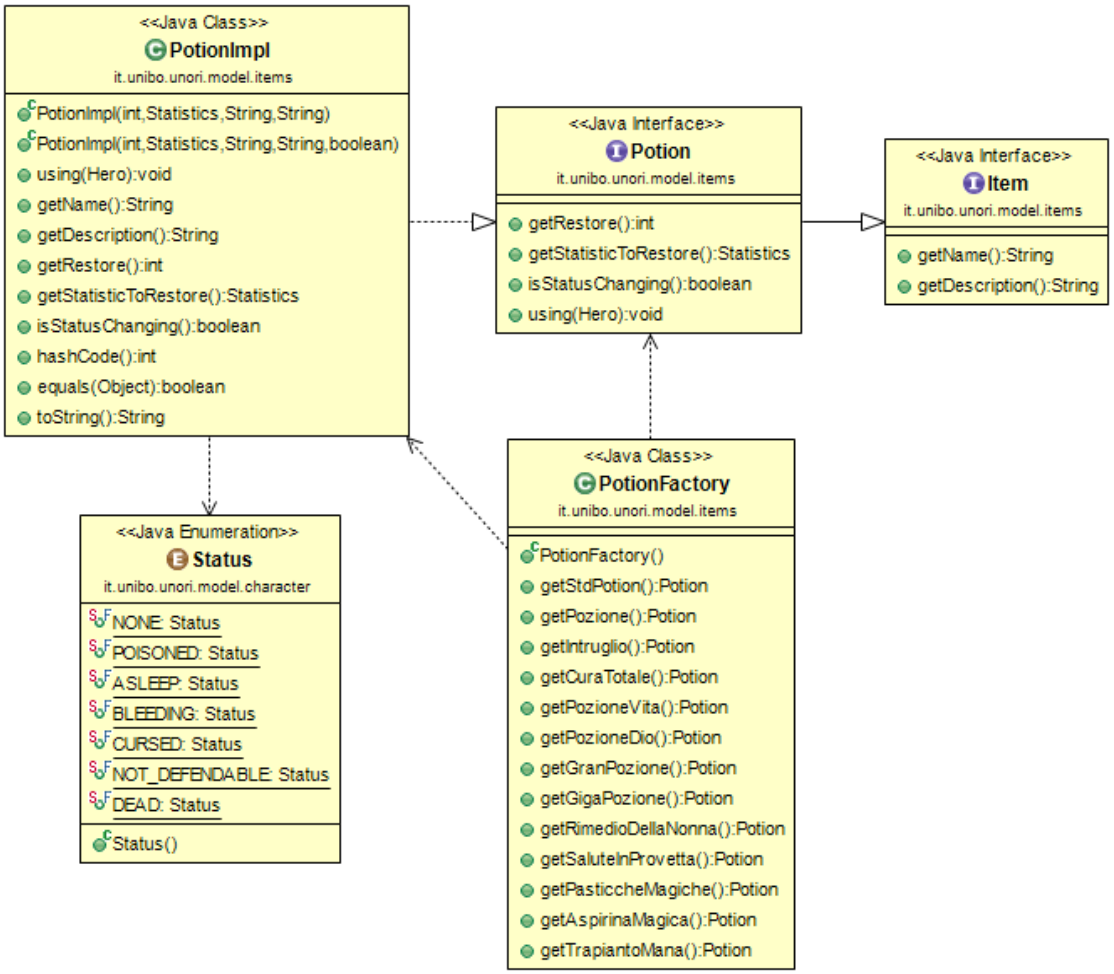


Figura 2.2.M2.5 – Schema UML relativo alle Pozioni.

Le Pozioni sono Strumenti (l'interfaccia *Potion* estende Item, come si può vedere in figura) in grado di rigenerare due tipi di Statistiche: Health Points e Mana Points. Alcune possono anche risolvere lo Status di un personaggio riportandolo allo *Status NONE*; quest'ultima particolare funzione è determinata da un parametro booleano all'interno della classe *PotionImpl* e può essere controllata grazie al metodo *isStatusChanging()*. Una Pozione può essere usata da un *Hero* anche in modalità Battaglia, grazie al metodo intrinseco di *PotionImpl using(Hero)*.
Le Pozioni sono generate dalla classe factory *PotionFactory*, che, grazie a dei *getter methods* restituisce la gamma di Pozioni (già costruite) incontrabili all'interno del gioco.

Model Parte 2: Menu

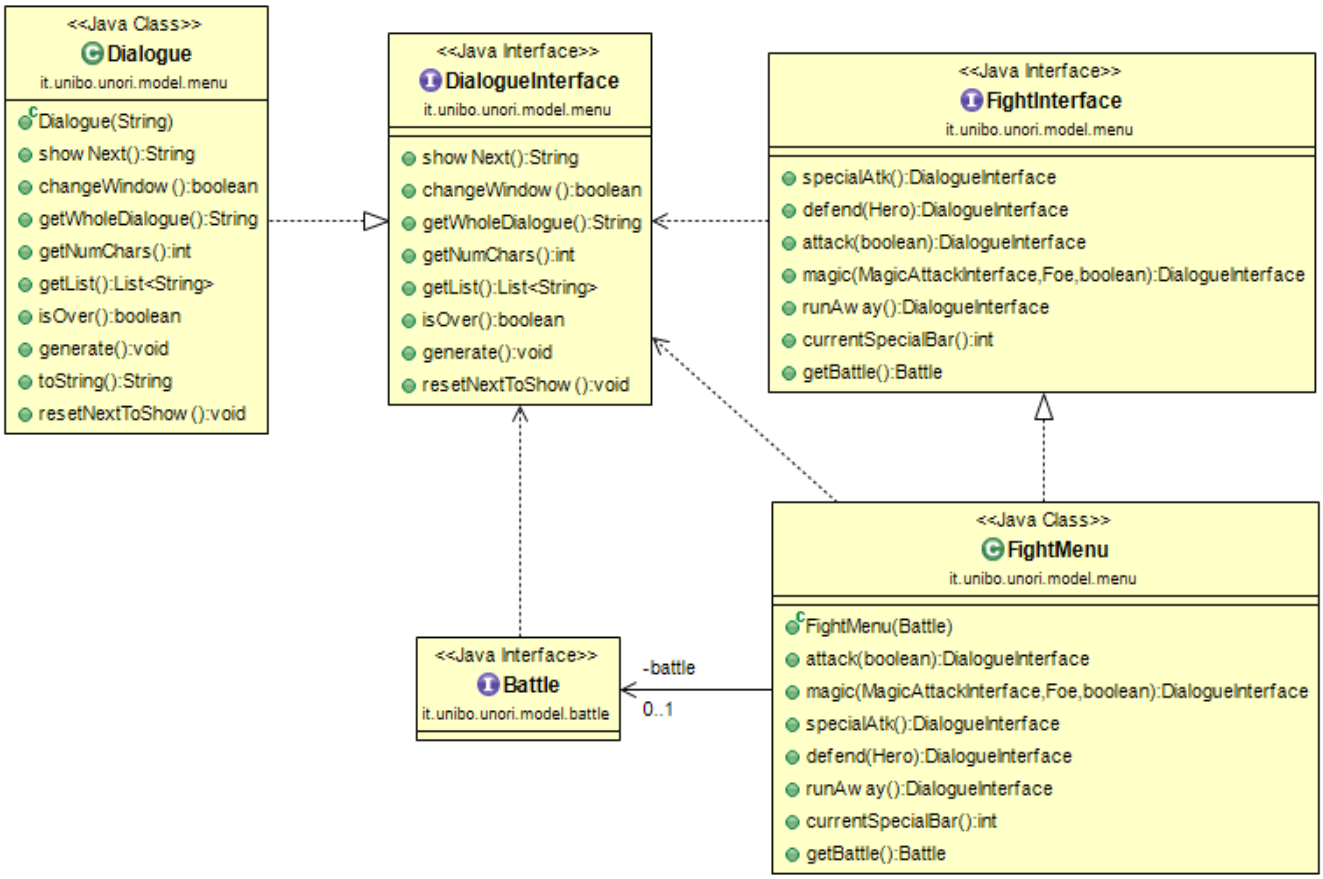
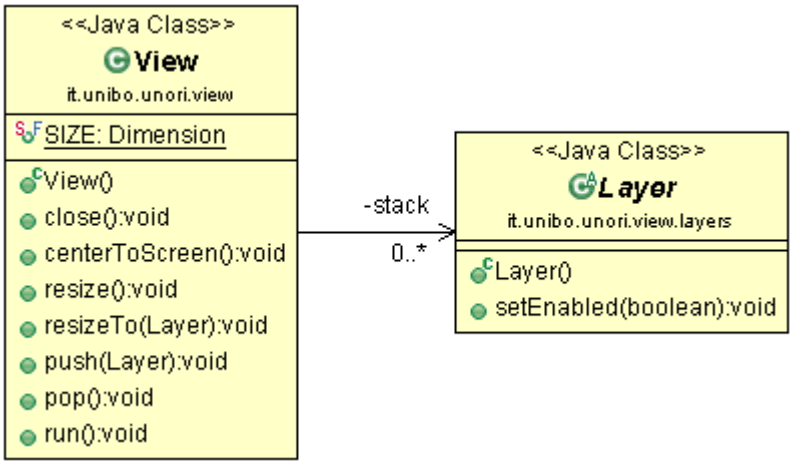


Figura 2.2.M2.6 – Schema UML relativo alla modellazione dei Menu.

La modellazione dei menu è incentrata sull'interfaccia *DialogueInterface* e sulla sua implementazione *Dialogue*. Quest'ultima consiste in una modalità di rappresentazione per Stringhe particolarmente lunghe da mostrare all'interno del gioco: questa classe suddivide una Stringa in varie sotto stringhe di lunghezza massima stabilita (70 caratteri, ma estendibile facilmente) che andranno a rappresentare le righe del dialogo, da inserire nella finestra. Grazie ai metodi *showNext()* e *changeWindow()* è possibile scorrere la lista di righe salvata all'interno della classe e pulire la

finestra di dialogo una volta riempita (ogni finestra può contenere due righe di testo; numero, però, espandibile).
L'interfaccia *FightInterface* rappresenta un'astrazione per l'interfaccia Battle, utile da passare al Controller. Essa (insieme alla sua implementazione *FightMenu*) contiene tutti i metodi fondamentali da includere nella modalità Battaglia.

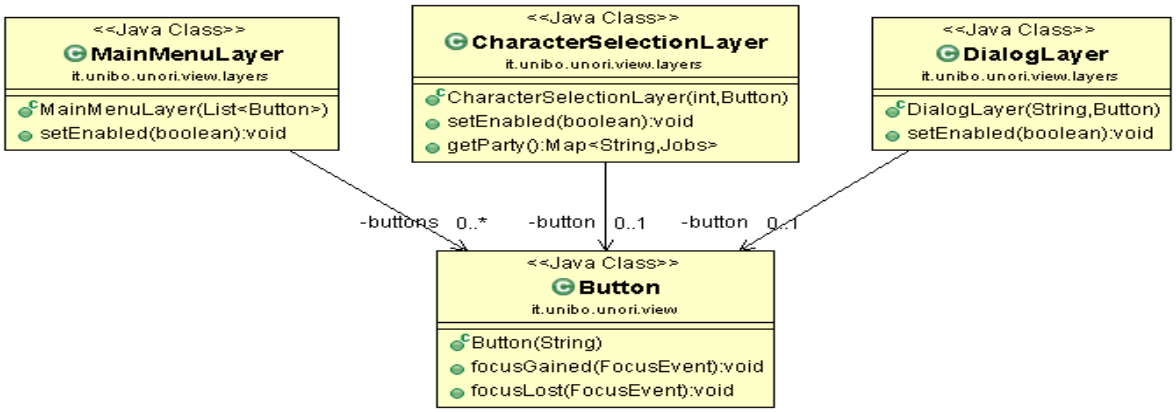
View: Struttura della View



La classe `View` è la *classe* principale per la visualizzazione di questo gioco, è la **finestra** di gioco, nella quale vengono sovrapposti i vari menu, interfacce e visuali.
Funziona concettualmente come uno ***stack*** che aggiunge in trasparenza vari livelli di grafica. Ha i metodi di un qualunque stack: *push()* e *pop()* **.**
I metodi aggiuntivi servono a posizionare la finestra di gioco nello schermo (*centerToScreen()*) e ridimensionare la visuale di gioco (*resize()*, *resizeTo(Layer)*).

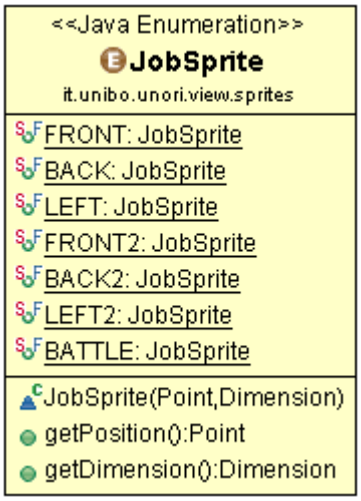
La classe *Layer* rappresenta un **livello** di grafica da inserire nella *View*. Ha la sola funzione essenziale di potere essere disabilitato (*setEnabled(boolean)*), e non ricevere quindi più nessun tipo di input.

View: Le interfacce iniziali



Le interfacce di gioco iniziali, *MainMenuLayer*, *CharacterSelectionLayer* e *DialogLayer* sono implementati come *Layer* da inserire nella *View* di gioco. Queste visuali vengono inizializzate con tutte le informazioni necessarie.
L'unico *Layer* che ha una funzione aggiuntiva è il *CharacterSelectionLayer* che deve restituire al **Controller** l'insieme dei personaggi creati, con *getParyt()*.
Tutte queste interfacce di gioco condividono un elemento grafico, il bottone (*Button*), che è bottone standard modificato per una resa grafica più gradevole. L'uso di questo *button* permette anche la comunicazione con il **Controller**, infatti può attivare una azione che viene inserire al suo interno e quindi passare il controllo.

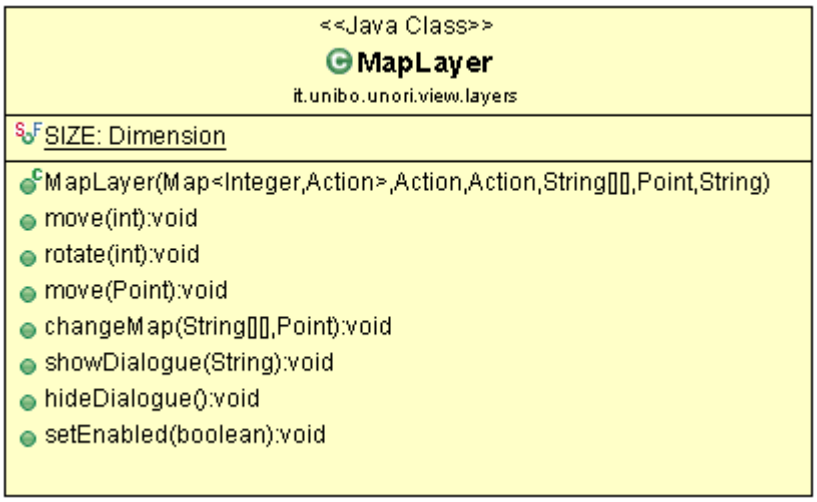
View: Gestione immagini di gioco



Per la gestione di tutte le **immagini** di gioco ho usato *classe* fornita dalla libreria grafica usata a lezione (AWT/Swing), chiamata *BufferedImage*. Combinata a questa classe ho dovuto creare un *enumeration*, *JobSprites*, per la gestione di più immagini contenute in un solo file, necessaria per l'**animazione** dei personaggi nella mappa.

In questa *enumeration* ho inserito le vari visuali del personaggio (*FRONT*, *BACK*, *LEFT*), due per ognuna di queste, per permettere di simulare un movimento. Per creare la visuale di destra (*RIGHT*), non presente nell'immagine, ho creato un metodo nelle classi che la utilizzano per invertire l'immagine di sinistra (*LEFT*).

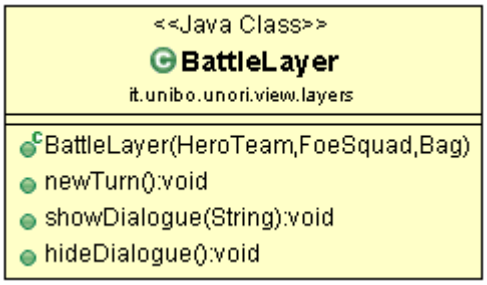
View: La mappa



Per la visualizzazione della **mappa** ho creato un *Layer* che viene inizializzato con una matrice di *stringhe* contenenti il percorso delle immagini che compongono la mappa (*String[][]*), e una serie

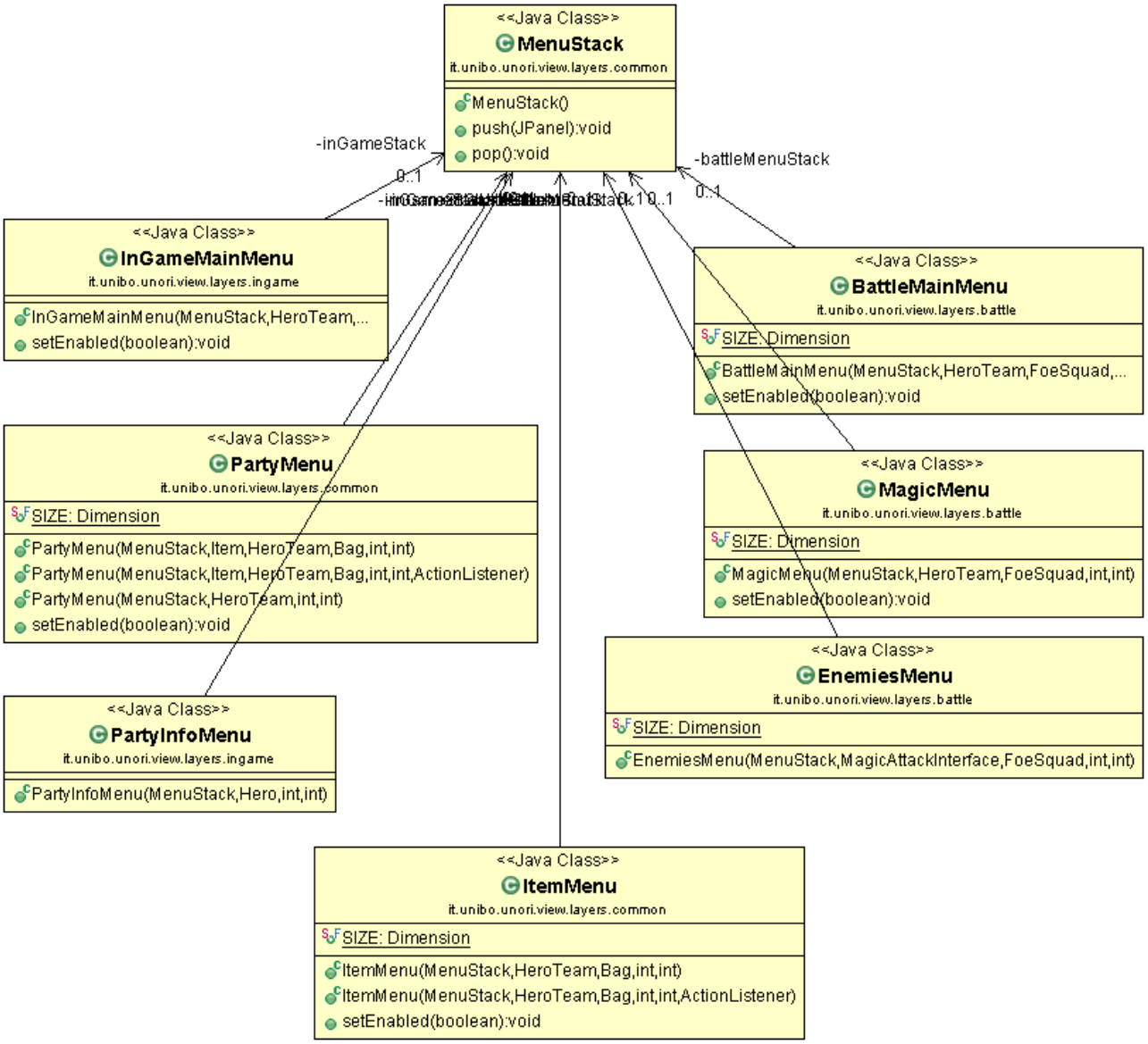
di azioni (*Action*), che vengono chiamate alla pressione dei tasti, che portano il controllo del gioco al **Controller**.
Le funzioni di questa classe, *show*/*hideDialogue* per la visualizzazione di un dialogo, *rotate*/*move* per la rotazione il movimento del personaggio, *changeMap* per l'aggiornamento della mappa di gioco, permettono al **Controller** di gestire lo stato della mappa e dirigere il movimento e l'interazione del personaggio.

View: La battaglia



La visualizzazione della **battaglia** consiste nel mostrare i personaggi che interagiscono tra di loro e le loro informazioni. Per fare questo la *BattleLayer* viene inizializzata contenente il team degli eroi e quello dei nemici, più la *Bag* che ha gli oggetti da potere usare. Come nella mappa il dialogo è gestito da *show*/*hideDialogue()*. Ogni turno è scandito da *newTurn()*.

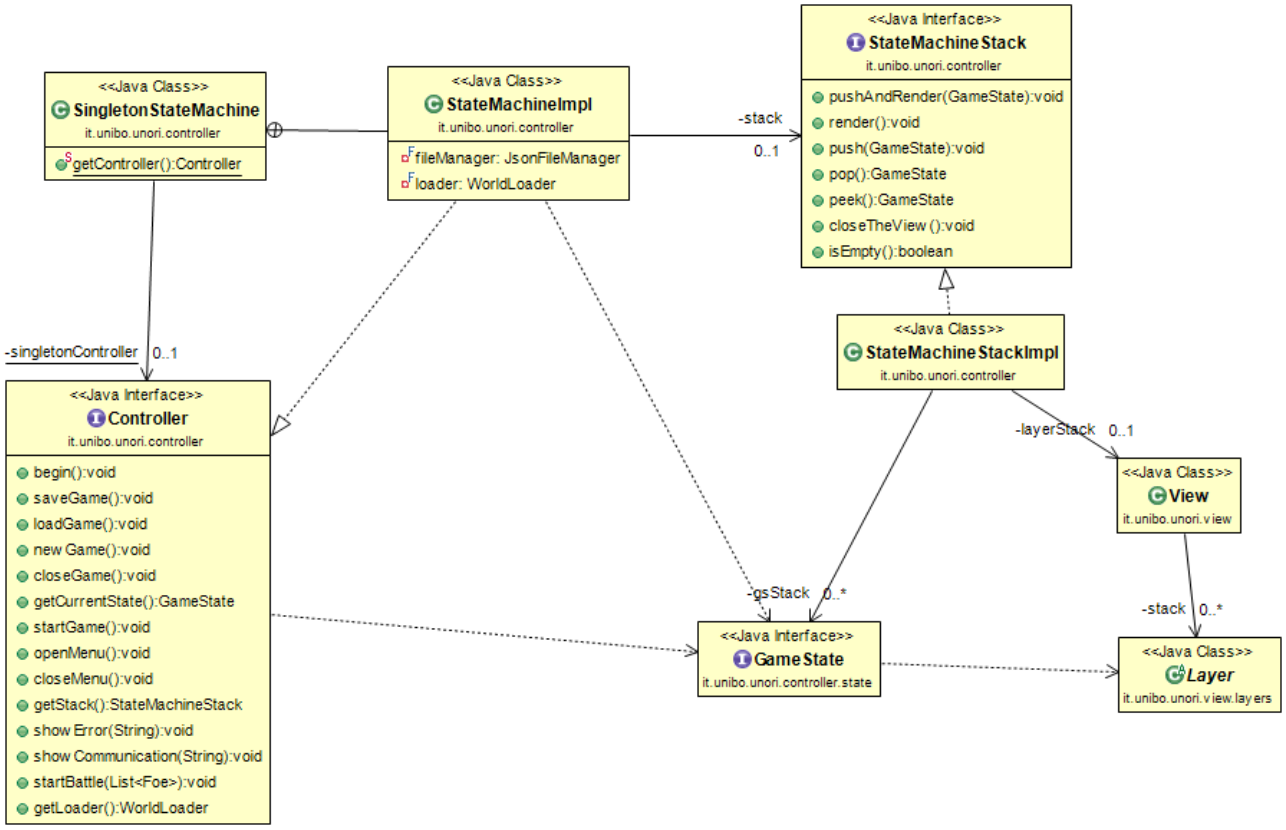
View: I menu



Per creare i menu all'interno della mappa e della battaglia ho riutilizzato il concetto di ***stack*** della View. Ho inserito un *MenuStack* sia *nell'InGameMenuLayer*, il menu in-game che nel *BattleLayer*. Questo *stack* di menu mi permette di aprire **menu** che a loro volta aprono altri menu, prendendo informazioni dal menu precedente.

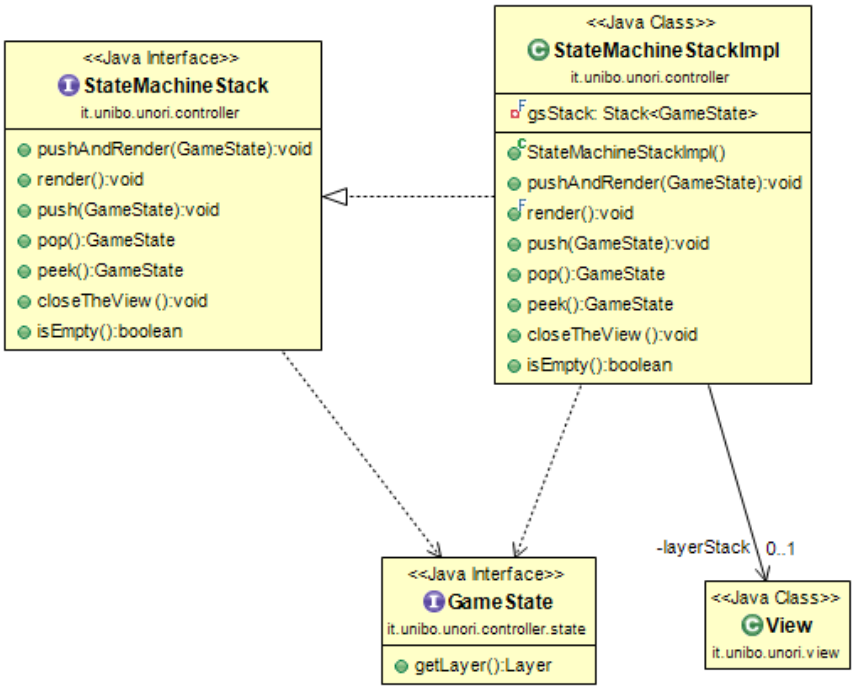
Controller: Struttura

Il controller ha avuto come obiettivo una gestione quanto più trasparente possibile dell'intero programma, permettendo il corretto funzionamento del gioco fornendo funzionalità di interfaccia tra model, view e sistema esterno (salvataggio e caricamento attraverso file).



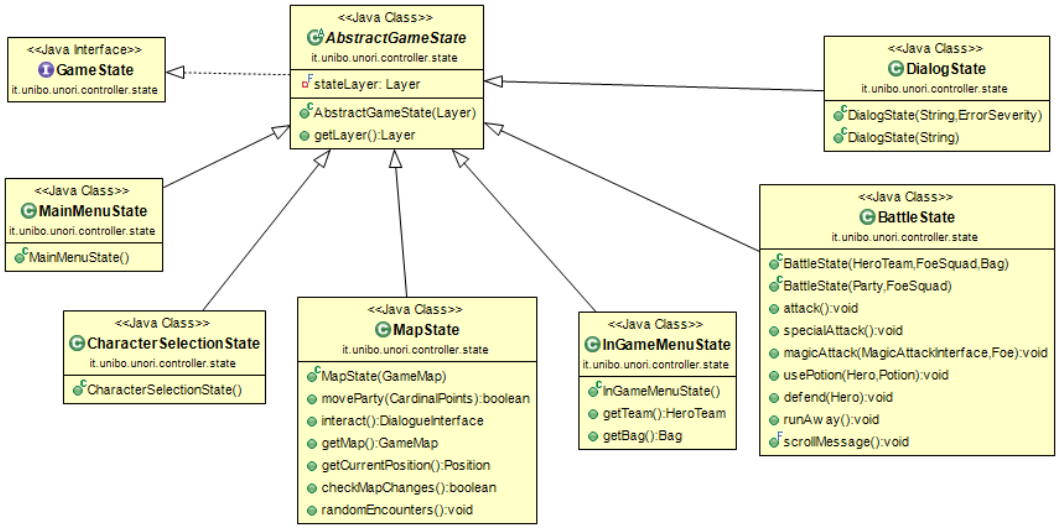
L'unità principale del controller si basa sul contratto fornito dall'interfaccia *Controller*, implementata sfruttando un pattern Singleton, *SingletonStateMachine* implementa una gestione del gioco come macchina a stati, dove ogni stato rappresenta un effettivo stato nel gioco (vedi figura). Attraverso l'unica istanza a runtime del controller è possibile accedere alle funzioni principali del gioco, quali il caricamento di una nuova partita o di una preesistente, il salvataggio su file, la visualizzazione di errori e comunicazioni, in generale tutte le funzioni che comportano un cambiamento di stato sullo stack. Queste funzionalità sono garantite dalla struttura a stack della macchina a stati.

Controller: Stack



L'interfaccia *StateMachineStack* è implementata come un doppio stack che contiene parallelamente uno *Stack<GameState>*, implementato sfruttando la classe Stack di libreria e uno stack di Layer, implementato invece nel nostro progetto attraverso la classe View.
Lo StateMachineStack fornisce i metodi base di uno stack, quali *push()*, *peek()*, *pop()* e *isEmpty()*, attraverso i quali eseguire i corrispettivi metodi nei due campi stack interni. Questa è la classe che gestisce più da vicino la grafica, occupandosi di estrarre da ogni *GameState* il corrispettivo *Layer* grafico da porre in cima allo stack grafico; essendo la classe più vicina alla componente grafica principale, è anche quella che si occupa della chiusura del gioco quando richiesto. La chiusura del gioco non comporta salvataggio, che deve essere fatto separatamente.

Controller: Stati di gioco

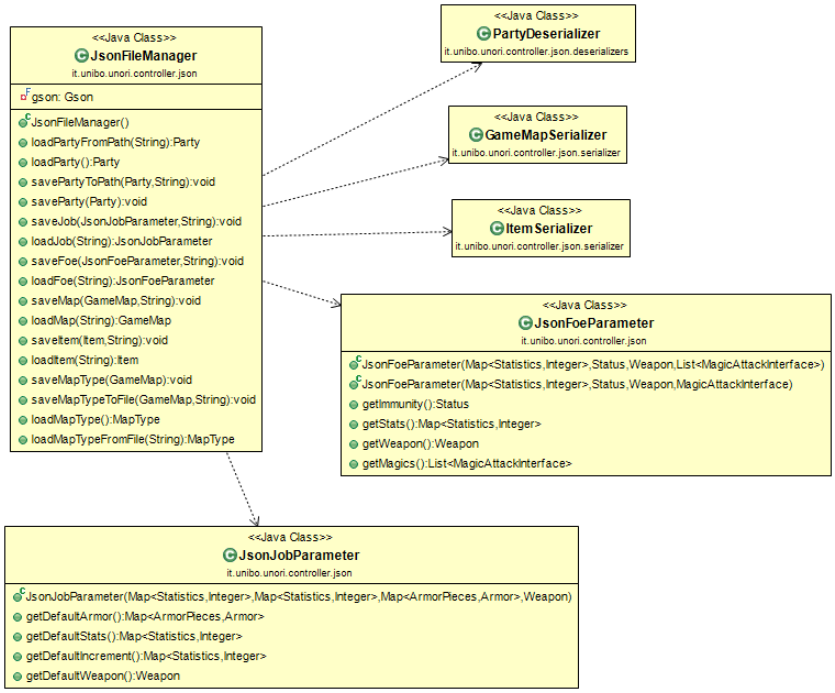


Implementando il contratto di GameState ed estendendo (per migliore incapsulamento del codice) la classe astratta AbstractGameState, vengono implementati tutti i possibili stati di gioco:

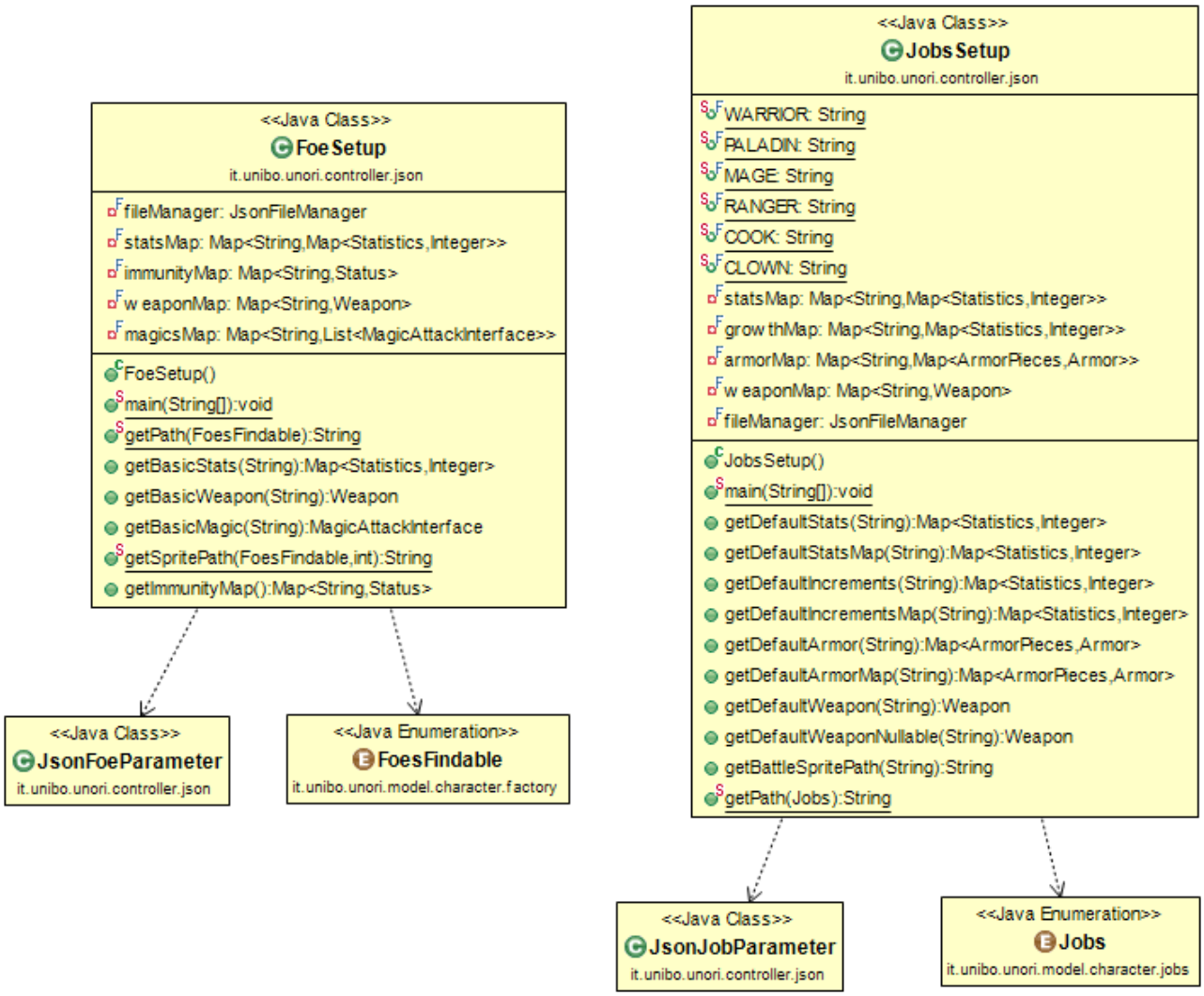
- **Menu principale**, implementato da *MainMenuState*, dal quale vengono richieste tutte le maggiori operazioni di caricamento da file (mappe, salvataggio); permette di uscire dal gioco, creare una nuova partita o caricare una partita precedentemente salvata su file.
- **Selezione del personaggio**, implementata attraverso *CharacterSelectionState*, che si occupa di costruire una interfaccia grafica per permettere al giocatore di scegliere gli eroi con cui iniziare una nuova partita.
- **Mappa**, implementata attraverso *MapState*, che permette, appoggiandosi ad un oggetto party e a un layer apposito per disegnare la mappa, di gestire il movimento impartito dall'utente sia a livello logico che a livello grafico. Gestisce anche l'inizio delle battaglie
- **Menu di gioco**, rappresentato da *InGameMenuState*, che richiama un layer grafico apposito in trasparenza sullo stato mappa precedente.
- **Battaglia**, rappresentato con *BattleState*, si occupa di gestire la battaglia sfruttando le rispettive classi di model e sfruttando un metodo per far visualizzare all'interfaccia grafica tutte le informazioni che il model ritorna ad ogni azione; esso si occupa inoltre di gestire l'alternanza dei turni e il comportamento dei nemici.
- **Errori e comunicazioni**, mostrate attraverso lo stato di gioco "anomalo" *DialogState*, che si occupa di far visualizzare all'utente attraverso una finestra di dialogo in overlay allo stato precedente una comunicazione di azione non valida o di un errore/eccezione, dando la possibilità allo sviluppatore di chiude il gioco o di ritornare semplicemente alla partita.

Tutti questi stati di gioco vanno di pari passo coi corrispettivi layer grafici, permettendo un corretto funzionamento dello stack precedentemente descritto.
L'interazione con gli elementi grafici all'interno di questi stati è possibile attraverso classi che estendo Action e ActionListener, che permettono al controller di ricevere input dalla view.

Controller: Serializzazione, Google Gson e clean boundary

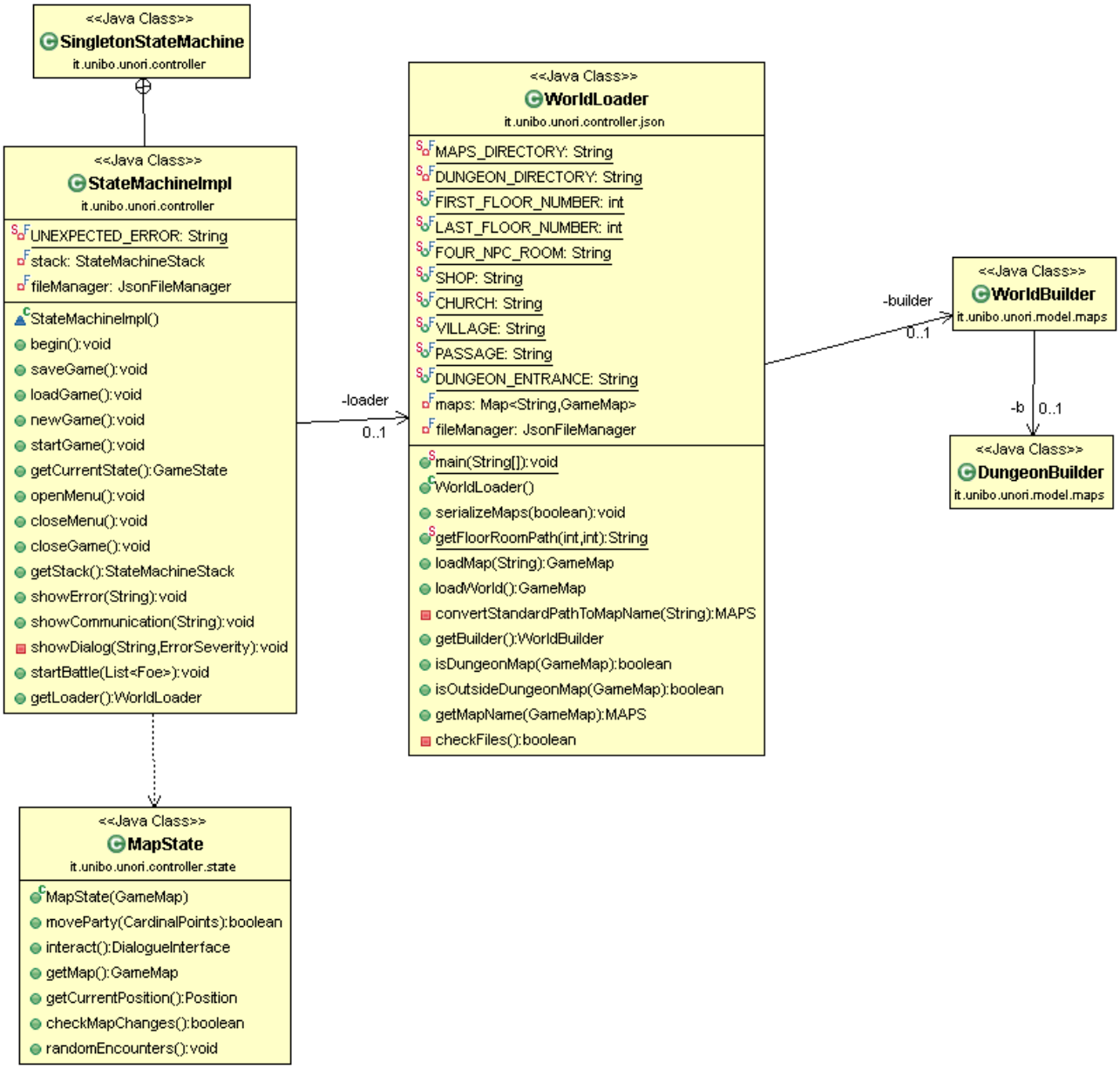


Poiché dovevamo gestire un salvataggio, abbiamo deciso di implementare la serializzazione su file human-readable, e la nostra scelta è ricaduta sul formato JSON nella sua implementazione fornita dalla libreria Google Gson. Purtroppo dai test iniziali è risultato che la libreria presentava problemi nella gestione delle nostre classi; ho così deciso di implementare un clean boundary tra le necessità del progetto e la libreria esterna: JsonFileManager. Esso fornisce, partendo da un oggetto Gson e delle classi specifiche che implementano *JsonSerializer* e/o *JsonDeserializer* (di alcune delle quali se ne ha la rappresentazione nell'UML in figura qui sopra) tutti i metodi per salvare su file e caricare da file tutti gli oggetti necessari al corretto funzionamento del programma.
La libreria è tornata utile non solo per quanto riguarda il salvataggio dei progressi di gioco, ma anche per le mappe e i parametri standard di eroi e mostri; in particolare per questi ultimi, è stato necessario modellare due classi, *JsonJobParameter* e *JsonFoeParameter*, che permettessero un salvataggio semplice solo dei parametri necessari, in modo che le enumerazioni del model potessero caricare senza problemi i parametri necessari.



Per fare ciò, sono state implementate le factory in figura, che sfruttano *JsonFileManager* e le due classi *JsonJobParameter* e *JsonFoeParameter* per fornire alle enumerazioni *FoesFindable* e *Jobs* del model tutti i parametri necessari. Nel caso fossero necessari gli stessi parametri più volte, vengono tenute delle copie di quanto deserializzato in attributi interni per migliorare le performance generali.

Controller: Mappa e caricamento



Il caricamento della mappa è anch'esso fatto da file attraverso un controllo della presenza delle mappe serializzate ed eventualmente genandone di nuove a partire dai parametri forniti dalle classi di model *WorldBuilder* e *DungeonBuilder*. Viene tenuto in memoria all'interno di attributi all'interno dell'istanza di *WorldBuilder* che è attributo della classe *WorldLoader*.

Sviluppo

3.1 Testing automatizzato

Nelle parti di model e controller sono stati sviluppati diversi test durante l'intera realizzazione del progetto; i test citati sono stati sviluppati sfruttando JUnit, un framework di Unit testing, secondo la metodologia proposta a lezione, si è cercato di inserire almeno un test per package; all'interno dei test si è cercato di evitare, per quanto possibile, di testare ogni metodo di ogni classe, ma di focalizzarsi solo su alcuni metodi il cui funzionamento era importante e necessitava di essere testato a fondo.
Per quanto riguarda la parte grafica invece, si è scelto di effettuare test manuali, anche inserendo alcune classi aventi il solo scopo di eseguire determinati blocchi di codice.

3.2 Metodologia di lavoro

Il team di lavoro era composto da quattro persone, e il pattern di progettazione scelto è stato l'MVC.
Il gruppo, dopo essersi trovato, ha deciso per la seguente divisione dei compiti, scegliendo di dividere la parte di model in due sotto parti:

- Maltoni: **Controller** (Gestione movimento, IA in battaglia dei nemici, salvataggio su file).
- Rasi: **View** (Ricerca e implementazione di grafica 2D).
- Semprini: **Model** (Meccaniche di lotta, modellazione degli oggetti consumabili, gestione in battaglia dei personaggi).
- Naldini: **Model** (Studio e realizzazione delle mappe di gioco e del movimento su esse, modellazione della parte statistica dei personaggi e degli equipaggiamenti

La maggior parte del lavoro è stato svolto dai componenti in maniera singola, sfruttando un repository Mercurial su BitBucket, tuttavia non sono mancate le occasioni di incontro o discussione tra i vari membri su questioni particolarmente ostiche; il gruppo ha cercato al meglio che poteva di eliminare le dipendenze tra le parti, ma il risultato non è stato dei più soddisfacenti.

3.3 Note di sviluppo

La classe *Pair*, presente nel package "*model.menu.utility*" è stata presa dalle soluzioni degli appelli d'esame dell'anno 2015.
Per la serializzazione abbiamo sfruttato la libreria *Google Gson*, reperita in formato jar da *www.java2s.com*.

Il gruppo ha inoltre consultato StackOverflow e i maggiori motori di ricerca per risolvere alcuni errori segnalati dai plugin FindBugs, PMD e Checkstyle.

Commenti finali

4.1 Autovalutazione e lavori futuri

A lavoro concluso il team può affermare di aver sottostimato la dimensione del problema da affrontare: infatti è stato molto difficoltoso per il team sia comprendere come realizzare tutte le meccaniche poste come obbiettivi, sia applicare il pattern MVC in maniera corretta e senza sovrapposizioni, in particolare la divisione del model in due parti ha comportato una frequente sovrapposizione del lavoro dei due membri sulle stesse classi, inoltre anche la suddivisione di lavoro tra la View e il Controller è cambiata molte volte a causa della mancanza, nei software usati, di una divisione netta tra grafica e controllo.
Inoltre il gruppo ritiene di aver svolto una fase di progettazione iniziale troppo breve, per colpa dell'inesperienza che li ha portati a sottostimare il problema.

- Autovalutazione di Naldini: ritengo di aver lavorato in generale in maniera discreta, con una cura verso i dettagli calante man mano che si avvicinava la data della consegna, tuttavia ho cercato di mantenere un livello medio di cura per i dettagli, sfruttando i plugin FindBugs, PMD e Checkstyle.
  In generale non ho riscontrato eccesive difficoltà nello sviluppo della mia parte di progetto, poiché probabilmente i problemi da risolvere erano molto simili a problemi già incontrati a lezione o nelle simulazioni d'esame, sono riuscito inoltre a tenere una certa coerenza di lavoro nella prima parte di sviluppo (fino a 45-50 ore circa) per poi fermare il mio lavoro causa altri impegni e riprendere circa due mesi dopo.
- Autovalutazione di Semprini: Ritengo di aver svolto la mia parte di lavoro in maniera coerente con quanto appreso durante il corso di studi, certamente non senza difficoltà o distrazioni. La cura alle ottimizzazioni di codice è sicuramente calata nelle settimane antecedenti la data della consegna, ma sono riuscito a evitare errori grossolani grazie ai plugin suggeriti nella parte di laboratorio del corso: FindBugs, Checkstyle e PMD. Le difficoltà principali sono arrivate soprattutto nella realizzazione degli algoritmi da adottare nella progettazione delle logiche di Battaglia, in cui ho cercato di unire fantasia e ragionamento per dare il mio massimo apporto al progetto.
  Sono riuscito a portarmi discretamente avanti con il lavoro nei mesi di maggio e luglio (comprendendo inizio agosto) per non accumulare il lavoro negli ultimi giorni rimanenti prima della dead-line.
- Autovalutazione di Rasi: Credo di essere riuscito a concludere la mia parte in maniera soddisfacente, pur non avendo curato abbastanza la struttura del mio codice e alcuni aspetti concettuali del progetto.
  Sono riuscito ad affrontare diversi problemi legati alla grafica sfruttando appieno le librerie fornite dal linguaggio di programmazione, creando dei pattern e codice riutilizzabile. Ho

sottovalutato però la difficoltà di altri problemi. Questo mi ha portato a lavorare in maniera non sempre costante.

- Autovalutazione di Maltoni: Penso di essere riuscito a svolgere il mio lavoro secondo quanto ho imparato a lezione e secondo le possibilità di tempo che ho avuto. Purtroppo però gli altri esami e alcune distrazioni mi hanno impedito di prestare tutta la cura che una pianificazione paziente e ben distribuita nel tempo può fornire. In ogni caso ho cercato di evitare tutti gli errori più comuni utilizzando i suggerimenti di FindBugs, Checkstyle e PMD.
  Le maggiori difficoltà di sviluppo le ho avute con la libreria Google Gson e con la gestione dei file, ma penso alla fine di essere riuscito ad utilizzarli in maniera abbastanza corretta. È stata inoltre una sfida cercare di capire il codice degli altri del gruppo per poter far funzionare il tutto nel modo più pulito possibile.

4.2 Difficoltà incontrate e commenti per i docenti

Non abbiamo nulla da segnalare

4.3 Manuale d'appendice

Per utilizzare il gioco, è necessario lanciare l'eseguibile jar, che aprirà una finestra con tre pulsanti: per lanciare una nuova partita, sarà necessario premere nuova partita, scegliere 4 personaggi con il tasto invio e fornendone un nome (i 4 nomi devono essere diversi) e si aprirà la mappa. Il movimento sulla mappa è possibile attraverso i tasti direzionali, mentre è possibile aprire e chiudere il menu con il tasto ESC. L'interazione con il menu di battaglia e con il menu in game è possibile attraverso il mouse.
La creazione di una nuova partita e il salvataggio durante la stessa, portano alla creazione di file all'interno della cartella UnOriginalRPG nella home utente del file system corrente.