Relazione progetto **PokeJavaMon**

---

by

Christian Serra, Daniel Veronesi,
Davide Bondi, Michael Camporesi

26 Maggio 2016

Sommario……...**1**

1 Analisi……..**2**

1.1 Requisiti……..**2**

1.2 Analisi e Modello del Dominio……..**3**

2 Design……..**5**

2.1 Architettura…….**5**

2.2 Design Dettagliato……..**6**

Model + Map(Christian Serra).....**6**

Fight (Davide Bondi).....**14**

View (Daniel Veronesi).....**17**

Controller (Michael Camporesi).....**22**

3 Sviluppo…….....**29**

3.1 Testing Automatizzato……....**29**

3.2 Metodologia di lavoro……**30**

3.3 Note di Sviluppo……**31**

4 Commenti Finali…….……**33**

4.1 Autovalutazione e Lavori Futuri……...**33**

5 Guida Utente……...**36**

5.1 Guida al Gioco……...**36**

5.2 Guida alla Creazione della Mappa…….**38**

1 Analisi

1.1 Requisiti

Il nostro progetto ha come obiettivo primario la realizzazione di una versione personalizzata del famoso videogioco Pokémon.

Pokémon è un gioco ambientato in una mappa liberamente esplorabile dal protagonista dell'avventura, controllato dall'utente tramite opportuni comandi. Nella mappa è possibile parlare con vari personaggi, sfidare allenatori avversari e, in opportune zone, incontrare pokémon selvatici.

Le difficoltà principali nella gestione della partita risultano essere la gestione del movimento del personaggio, la collisione di esso con vari elementi dell'ambiente, la gestione dell'interazione con gli elementi della mappa e i vari NPC, e l'incontro dei vari pokémon selvatici.

- Il giocatore è in grado di muoversi liberamente e fluidamente nella mappa di gioco;
- La gestione delle collisioni con i vari elementi dell'ambiente (quali edifici, cartelli, allenatori, etc.) deve essere gestita attentamente;
- Deve essere gestito in maniera opportuna il combattimento contro allenatori e pokémon selvatici (dei quali deve inoltre essere gestito l'incontro nelle zone stabilite);
- Viene data al giocatore la possibilità di usare diversi strumenti (pozioni per curare pokémon feriti, strumenti per catturare quelli selvatici, etc.), l'utilizzo dei quali deve essere attentamente gestito;
- Diventa così possibile per ciascun giocatore scegliere un percorso e un approccio di gioco sempre diversi, così da permettere a ciascuno la scelta dei pokémon da allenare e dell'approccio di gioco da seguire;
- E' possibile salvare la partita in qualsiasi momento, riprendere una vecchia partita o iniziarne una nuova;
- Il software offre un'ampia varietà di pokémon selvatici, ciascuno con il proprio set di mosse personale da poter usare in combattimento;
- Alcuni pokémon hanno la possibilità di evolversi, cambiando così il loro aspetto e le loro statistiche, che variano dinamicamente a seconda del pokémon e del livello dello stesso;

- Sebbene non viene dato al giocatore un obiettivo specifico, questo può decidere di catturare un pokémon per ogni specie presente, battere tutti gli allenatori avversari, battere tutti i capi palestra (allenatori speciali che, una volta battuti, consegnano al giocatore una medaglia), catturare tutti i pokémon leggendari (speciali pokémon che si incontrano in zone segrete), o tentare di riuscire a compiere tutte queste imprese. Questo lascia al giocatore la scelta totale riguardo a come impostare la sua partita;
- Come già accennato, esistono allenatori speciali, i capi palestra, che conferiscono medaglie che permettono al giocatore di accedere a nuove aree di gioco inizialmente inaccessibili;

- Sono inoltre presenti pokémon speciali, che non è possibile incontrare casualmente come i pokémon selvatici, ma con i quali si può interagire direttamente. Questi sono più forti e più difficili da catturare degli atri.

1.2 Analisi del Dominio e Modello

Il programma deve essere in grado di gestire la mappa di gioco e tutti gli elementi che la popolano.

La mappa è composta da diversi tile, sui quali si trovano diverse entità.

Esistono vari tipi di entità:

- **NPC(Non-Player-Character)**: Un personaggio con il quale è possibile parlare;

- **Trainer:** Un allenatore nemico che è possibile sfidare, ha una ***Squad*** con i ***Pokemon*** pronti per combattere;

- **PokeMap:** La mappa divisa in ***Tile*** che contiene tutte le entità di mappa, tra cui anche ***Trainer, NPC***… Viene inizializzata grazie all'inner class ***ImportMap***;

- **Player:** Il player giocabile, possiede una ***Squad*** per contenere i ***Pokemon*** da usare in ***Fight***, un ***Inventory*** nel quale sono presenti gli ***Item*** utilizzabili ed anche un ***Box*** per depositare i pokémon non utilizzati.

Altro elemento fondamentale è il combattimento, che richiede di gestire opportunamente le scelte del giocatore, il quale può decidere, durante il suo turno, di far attaccare il proprio pokémon usando una delle mosse a sua disposizione, cambiare pokémon, usare un oggetto e, ove possibile, tentare di fuggire dal combattimento.

Deve essere inoltre pensata un'intelligenza artificiale in grado di rispondere ottimamente alle mosse del giocatore e, a seconda della situazione, contrattaccare con la migliore mossa possibile.

Devono essere gestite con attenzione le collisioni tra il giocatore e gli elementi presenti nella mappa, cercando di garantire al giocatore un'animazione fluida ed adattabile.

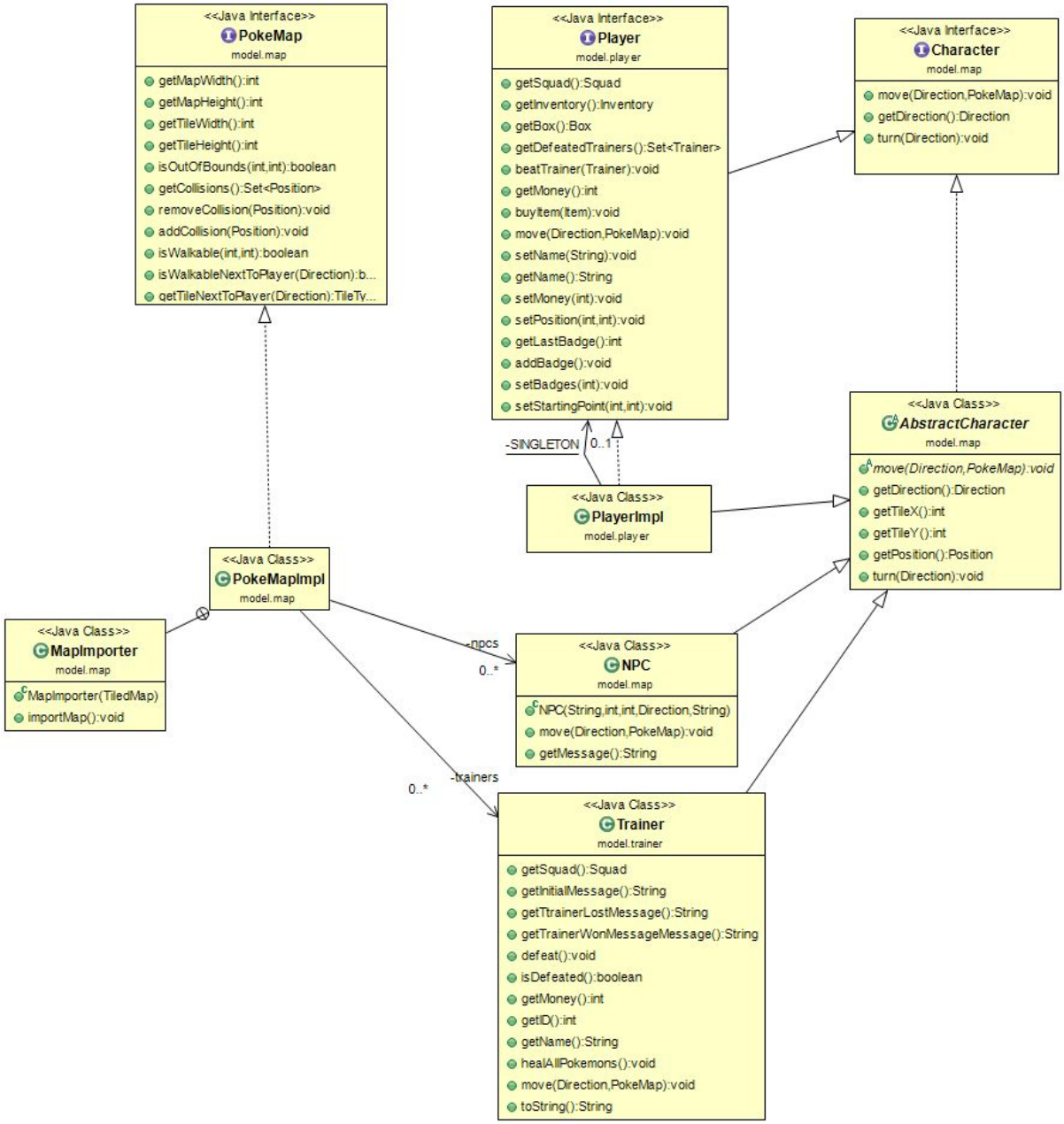
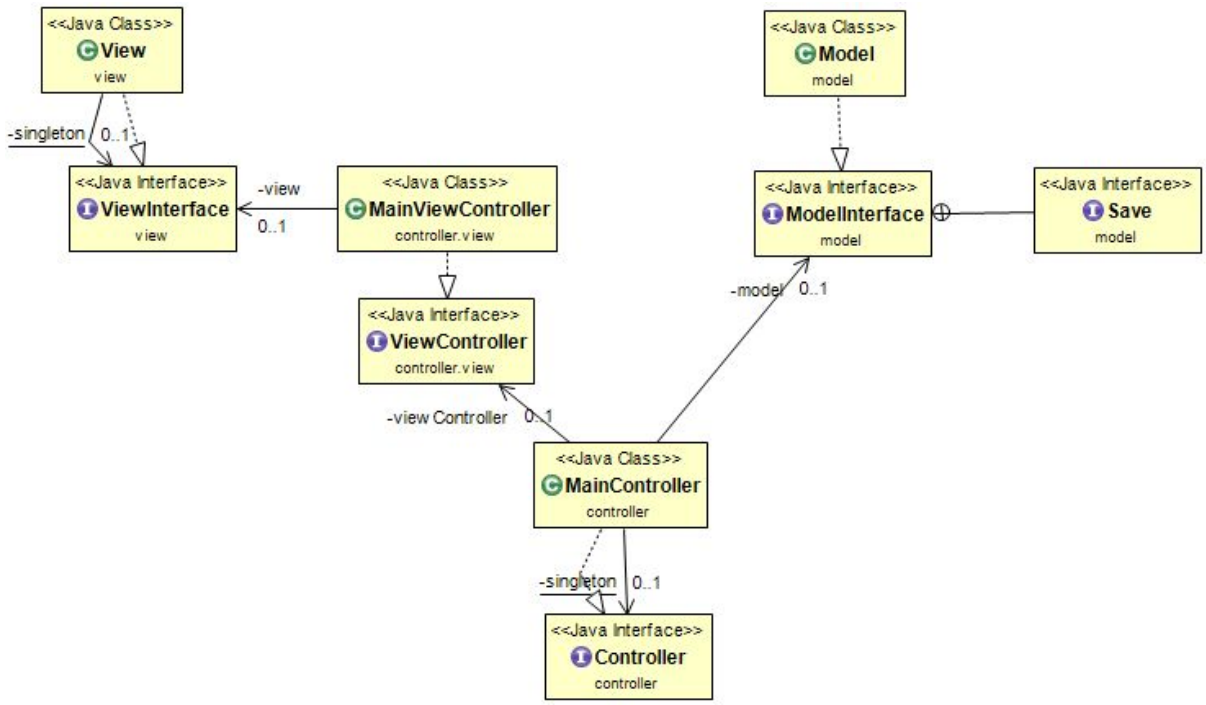


Diagramma UML del dominio

2 Design

2.1 Architettura



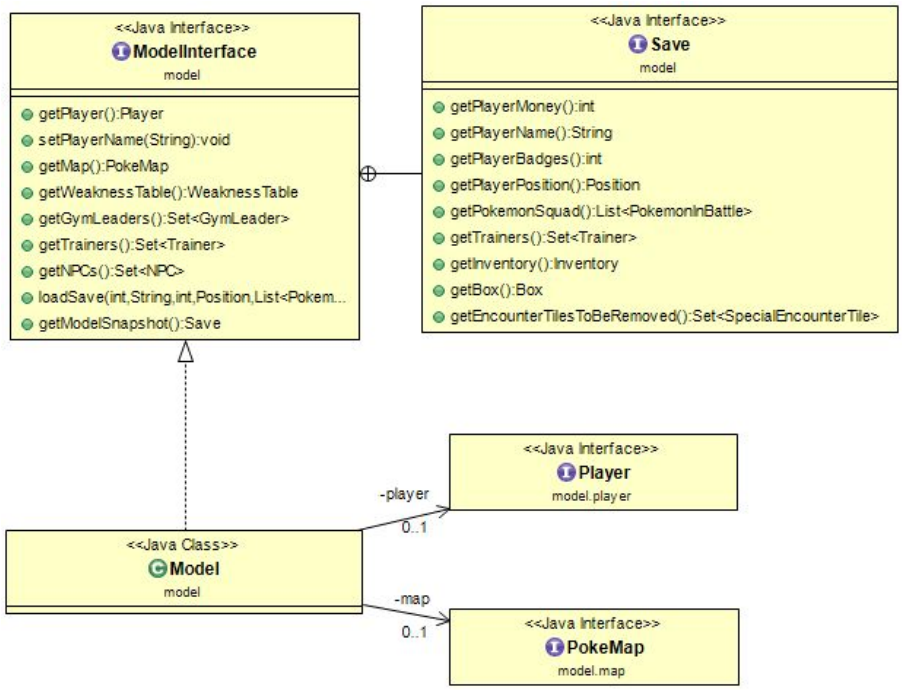
Durante la fase di analisi abbiamo deciso di adottare, per quanto concerne l'interazione tra le varie parti di cui è composto il nostro software, il pattern MVC (Model, View, Controller), che prevede la suddivisione del progetto in tre grandi parti distinte, per l'appunto Model, View e Controller.

All'inizio di una partita, oltre all'installazione delle risorse necessarie, viene inizializzato un Controller, che si occupa di inizializzare immediatamente la View.
Quando viene registrata la scelta dell'utente riguardo al tipo di partita che vuole intraprendere (una nuova partita o riprendere la vecchia), la View comunica al Controller la scelta, e questo inizializza il Model di conseguenza.

Particolare motivo di orgoglio è stato notare che, sopraggiunta la voglia di implementare nuovi allenatori, nuovi pokémon, nuove mosse, nuove zone nella mappa e nuove canzoni, è stato possibile fare questo senza alcuna modifica sostanziale in nessuna parte di codice, segno che l'espandibilità acquisita dal nostro software ha raggiunto standard che riteniamo soddisfacenti.

2.2 Design Dettagliato

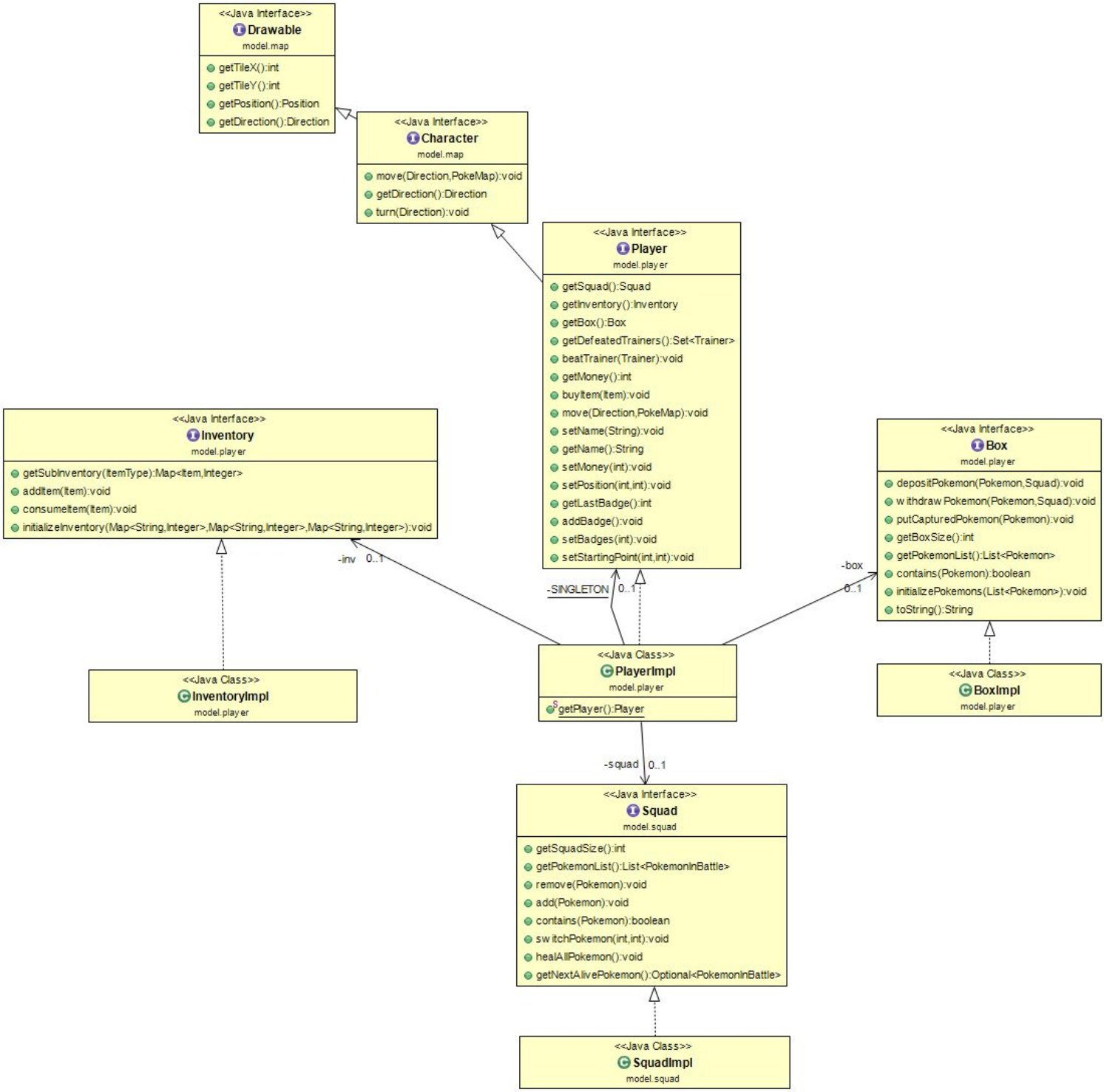
Model: Christian Serra



Il model ha la responsabilità di gestire i dati, le entità del dominio, garantendone una correttezza logica, come specificato dal pattern architetturale MVC.
Model è la classe principale dell'omonimo package, implementa l'interfaccia ***ModelInterface*** ed è utilizzata per accedere alle principali entità del dominio e per inizializzare i dati, sia in caso di caricamento che di nuova partita. L'interfaccia innestata ***Save***, necessaria al controller per salvare tutti i dati necessari in un file di testo, viene implementata da una classe anonima all'interno del metodo *getModelSnapshot*.

Come accennato prima, uno degli elementi fondamentali appartenenti al model è il ***Player,*** un'entità unica che rappresenta il personaggio giocabile dall'utente. Tramite input da mouse e tastiera, gestito dal controller, è possibile far eseguire al Player una serie di azioni tra cui combattere, comprare oggetti, depositare pokémon e tanto altro.

La struttura è cosi suddivisa:

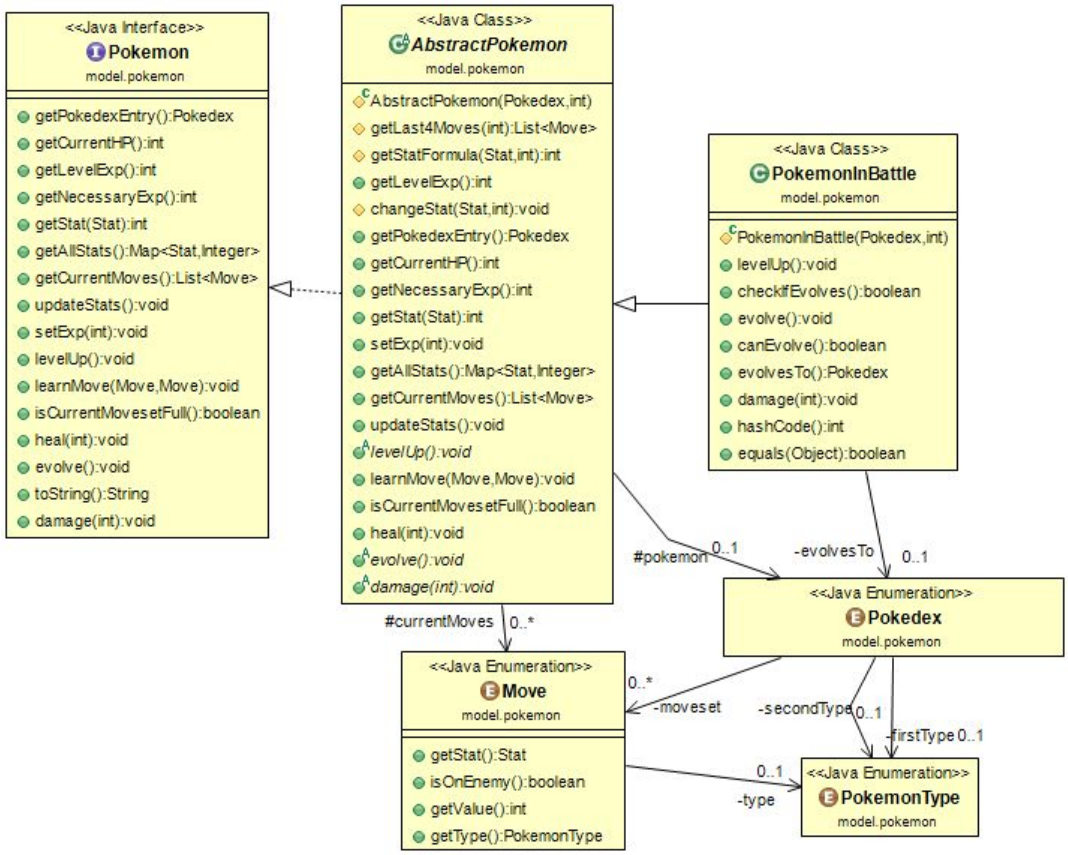


L'interfaccia è implementata da ***PlayerImpl,*** il quale segue il pattern creazionale Singleton poiché non è previsto che ci siano più entità di questo stesso tipo. L'unico svantaggio di questo approccio è dato dall'aumento della visibilità dell'oggetto a livello globale. Il Player, come mostrato in figura, contiene anche altre entità che sono necessarie per tener traccia dei suoi ***Item*** e ***Pokemon***.

Inventory è il deposito di oggetti utilizzabili dal Player e organizza la disposizione degli ***Item*** in ***HashMap***(con gli ***Item*** come chiave e la quantità in ***Integer*** come valore), una per ogni sottotipo. In questo modo una qualsiasi istanza particolare di ***Item*** appare una ed una sola volta come chiave, da una parte per evitare riempire la mappa con troppi oggetti, dall'altra perché costa meno l'accesso e la modifica al valore (O(1) "ammortizzato") rispetto ad una lista con molteplici ricorrenze. Il metodo di ***Inventory*** permette di poter aggiungere e rimuovere un ***Item*** ma anche di inizializzarlo se si parte da un salvataggio pre esistente.
Inizialmente adottava anche il pattern Singleton ma verso la fine ho optato per rimuoverlo per evitare di avere troppe istanze globali.
Squad contiene tutti i ***Pokemon*** da utilizzare in battaglia e il metodo per poterli aggiungere, depositare e spostare di posizione.

Naturalmente l'implementazione effettiva ***SquadImpl*** ha anche un limite di pokémon massimi, 6. Ha inoltre un metodo che permette di far recuperare tutti gli Health Points (HP) ai pokémon della squadra. ***PlayerImpl*** non è l'unica classe che dispone di un campo ***Squad***: anche ***Trainer,*** e per estensione ***GymLeader,*** ne hanno una.
Box è invece uno storage per tutti i ***Pokemon*** che non partecipano alla lotta, con tutti i metodi necessari per depositare, rimuovere ed accedere a tutti gli elementi.

Un ***Pokemon*** è una creatura in grado di combattere, guadagnare esperienza e livelli, evolversi, imparare mosse e tanto altro.



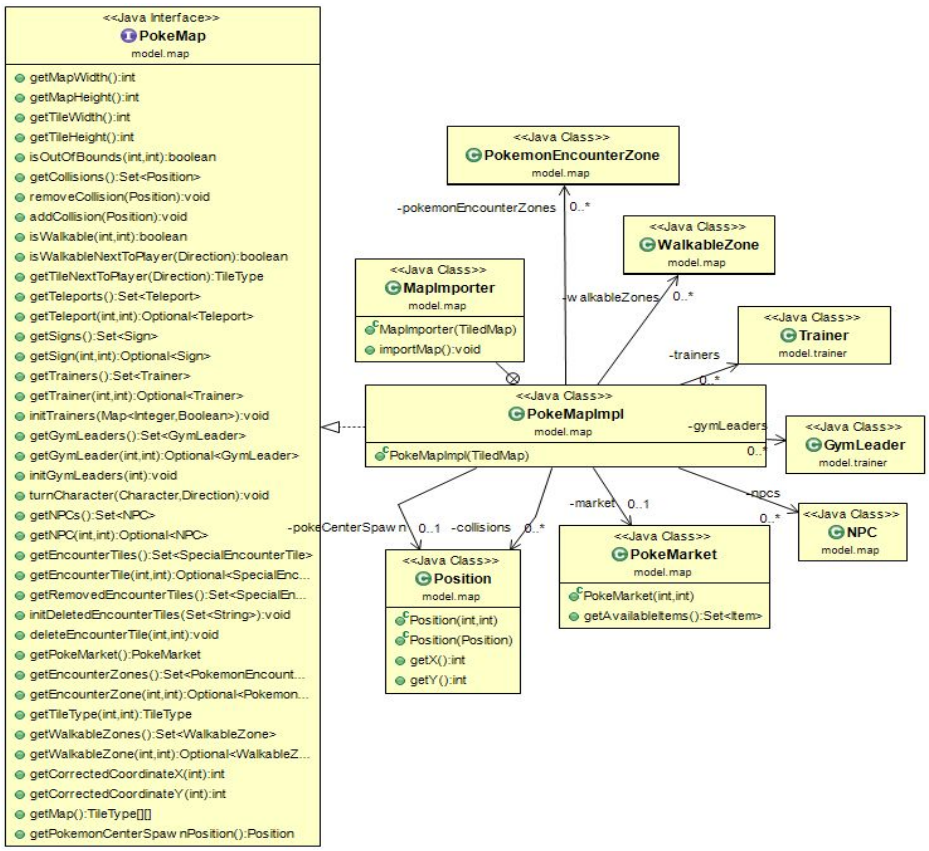
Essenzialmente un Pokémon deve avere:

- Delle ***Stat*** di attacco, difesa, velocità, punti vita (HP), esperienza (EXP), e livello;
- Un set di ***Move*** (MAX 4) da poter usare in combattimento. Al momento è possibile apprendere mosse soltanto durante l'aumento del livello (un dato Pokemon può apprendere solo mosse prefissate per livelli stabiliti), ma è facilmente implementabile anche il fatto che possa apprenderle in altre modi, tipo da oggetti;
- Fino a due ***PokemonType*** che rappresentano i suoi elementi caratteristici; entrambi determinano moltiplicatori di danno per ***Move*** subite, come previsto dalla ***WeaknessTable***, ma anche inflitte (http://bulbapedia.bulbagarden.net/wiki/Same-type_attack_bonus);
- Una ***PokemonRarity*** per stabilire il tasso di cattura tramite ***Pokeball*** e il tasso di apparizione in una ***PokemonEncounterZone;***
- Una possibile evoluzione, che al momento, può avvenire solo ad un livello specifico per ogni Pokemon.

Tutte le proprietà "costanti" di un certo Pokemon, cioè quelle che non variano all'aumentare del livello (quindi ***Stat*** base, ***PokemonRarity***, ***PokemonType,*** evoluzioni, tutte le ***Move*** imparabili…) sono contenute dentro tutti i valori dell'Enum ***Pokedex*** che praticamente agisce come una sorta di database statico. In particolare tutte le ***Move*** apprendibili sono inizializzate staticamente in una classe separata (***InitializeMoves)*** per dividere il codice che se no sarebbe risultato ancora più lungo dato che ogni singolo ***Pokemon*** impara una quindicina di mosse diverse. L'inizializzazione avviene tramite mappe (livello, mossa) ottenute grazie all'***ImmutableMap*** della libreria *Google Guava* che permette una più semplice e pulita creazione di mappe non modificabili.

L'implementazione di ***Pokemon*** adotta il principio Interface, Abstract Class, Concrete Class ed inoltre le istanze è stato preferito crearle tramite una ***StaticPokemonFactory,*** perciò tutti i costruttori sono *protected.*
In ***AbstractPokemon*** sono implementate tutte le funzioni di calcolo delle ***Stat*** per livello, tenuto traccia degli HP, mosse, EXP… mentre ho deciso di lasciare abstract i metodi levelUp(), evolve() e damage(int) perché ***PokemonInBattle*** è al momento solo una delle implementazioni possibili di ***Pokemon*** siccome ne esistono altre varietà che applicano in maniera diversa queste funzionalità (es: evoluzione per scambio invece che per livello).
La realizzazione dell'entità ***Pokemon*** è stata ridotta al minimo indispensabile dal momento che, se si da un occhiata alle versioni delle ultime generazioni, si notano molte più variabili riguardo a statistiche, evoluzioni...

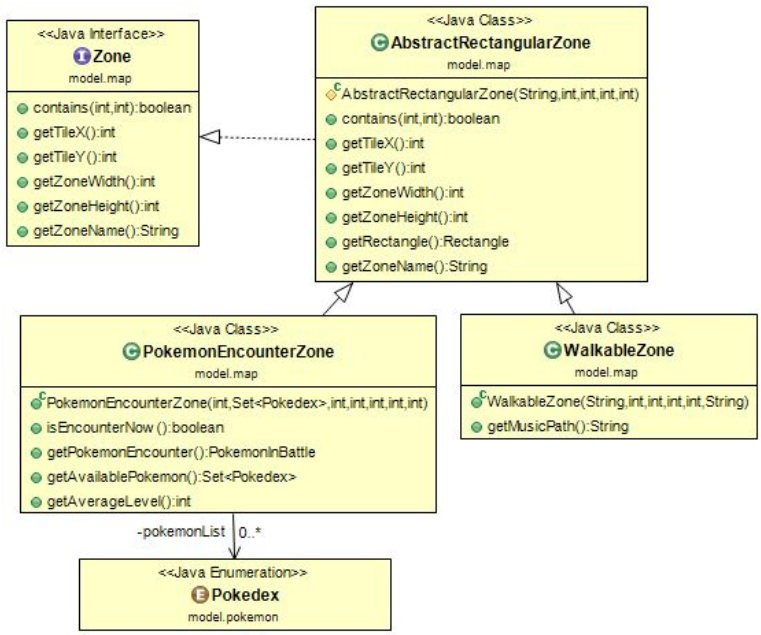
PokeMap è il terreno su cui il ***Player*** si sposta e su cui sono presenti tutte le entità ***Drawable***, quindi ***Tile*** speciali, ***NPC*** vari, ***Zone***, collisioni etc… L'implementazione, ***PokeMapImpl***, si avvale di una classe interna (***MapImporter***) per importare e generare i dati necessari dall'oggetto ***TiledMap*** generato dal file di mappa map.tmx dalla libreria *Libgdx*, che viene utilizzata anche per il suo rendering.



Per gestire il tassellamento di tutti ***Tile***, ho deciso creare una matrice [mapWidth][mapHeight] per salvare semplicemente i ***TileType*** di tutti i ***Tile***, piuttosto che creare mapWidth*mapHeight istanze delle varie implementazioni per motivi di performance e spazio, dato che essenzialmente i valori delle Enumerazioni hanno prestazioni simili agli ***Integer***.
PokeMapImpl inoltre contiene tutti i metodi per controllare uno specifico ***Tile*** della mappa e cercare/restituire altre entità della mappa come ***NPC, Zone, Trainer, PokeMarket…***
Il metodo più utilizzato è isWalkable(int,int), il quale controlla se il ***TileType*** di quella specifica posizione è walkable **e** se lì è presente una collisione.
Sono consapevole che si tratta di una "God class" in quanto contiene troppe funzionalità, ed è per questo che in uno degli sviluppi futuri ci sarà l'impegno di suddividere la mappa in sotto mappe per le varie entità, quindi NPCMap, TrainerMap, ZoneMap, per suddividere meglio il codice e i compiti delle sottoparti.

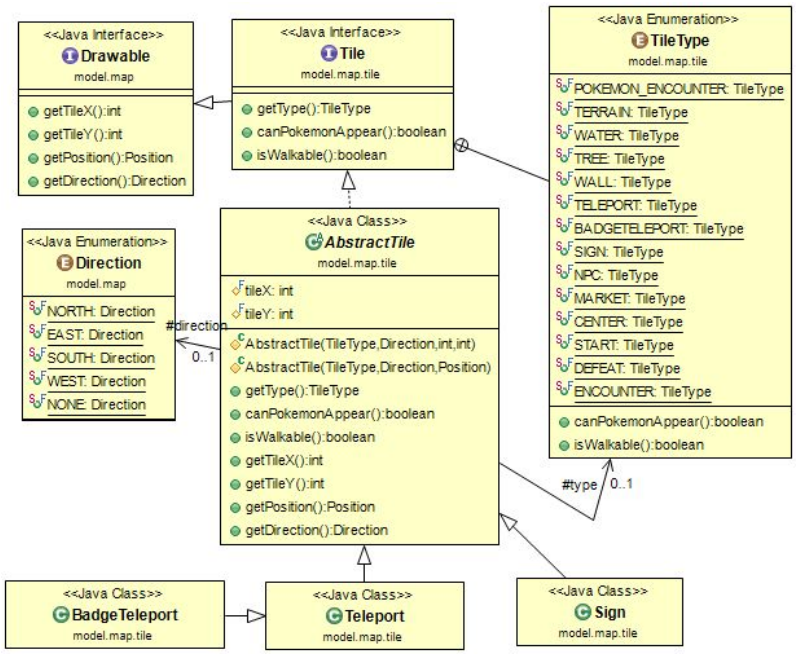
In particolare le ***Zone*** sono essenzialmente aree di gioco.

Mentre l'interfaccia lascia spazio a implementazioni con varie forme, a patto che sia mantenuta la correttezza del metodo boolean contains(int, int), quella da me proposta utilizza un'area rettangolare utilizzando un ***java.awt.Rectangle*** che combacia esattamente con i ***Tile*** sottostanti.



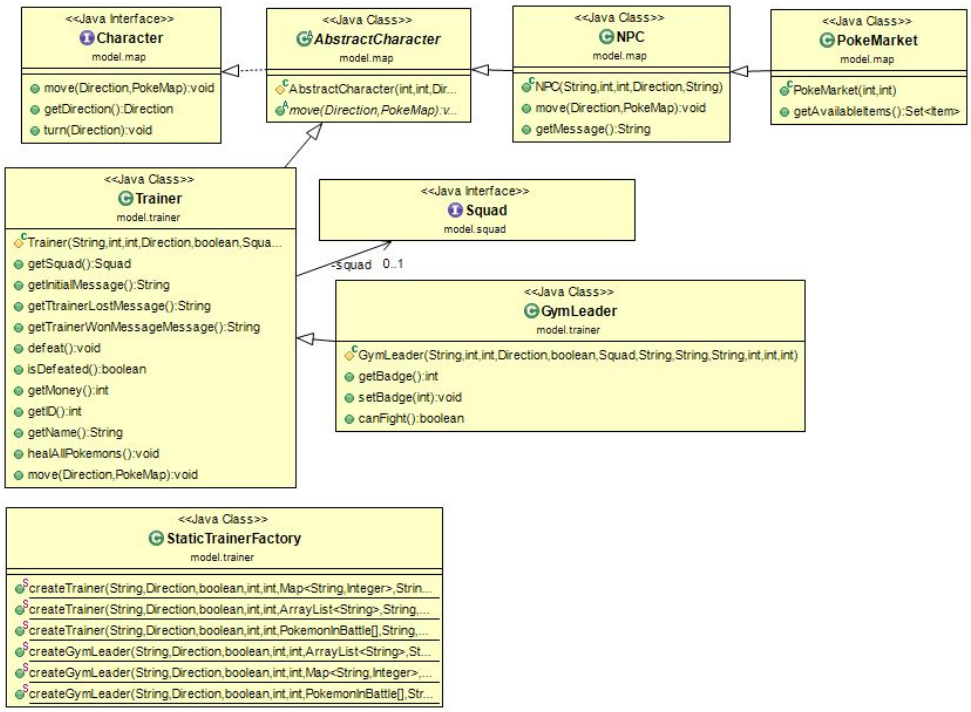
La classe astratta ***AbstractRectangularZone*** semplicemente fattorizza il codice comune, nonché lo stesso oggetto di ***java.awt.Rectangle***, mentre le due implementazioni concrete, ***PokemonEncounterZone*** e ***WalkableZone,*** servono a gestire rispettivamente l'incontro fortuito con ***Pokemon*** selvatici e la riproduzione di una canzone specifica quando il ***Player*** la attraversa.

Un ***Tile,*** invece, è la più piccola unità percepibile di spazio all'interno della mappa. Le attuali dimensioni in pixels dipendono dalla costruzione del file map.tmx, nel nostro caso si parla di ***Tile*** 16x16 pixels.



Come spiegato nella guida alla fine, ogni ***Tile*** può avere diverse proprietà (oltre alla posizione nella mappa), a seconda della sua complessità: quelli più semplici (Terreno, Erba, Acqua, Muri...) in realtà non vengono instanziati come veri e propri ***Tile*** ma vengono mantenuti nella ***PokeMapeImpl*** come semplici valori dell'enum ***TileType.*** Quelli complessi (***Sign, Teleport, BadgeTeleport***) sono invece implementati seguendo anche qui il principio Interface, Abstract Class, Concrete Class.
Sign è semplicemente un ***Tile*** che mostra un messaggio. ***Teleport***, se il ***Player*** ci cammina sopra, lo teletrasporta alla specificata posizione. ***BadgeTeleport*** è un ***Teleport*** che funziona solo quando il ***Player*** ha almeno il numero minimo di medaglie richieste, che si ottengono sconfiggendo i ***GymLeader.***

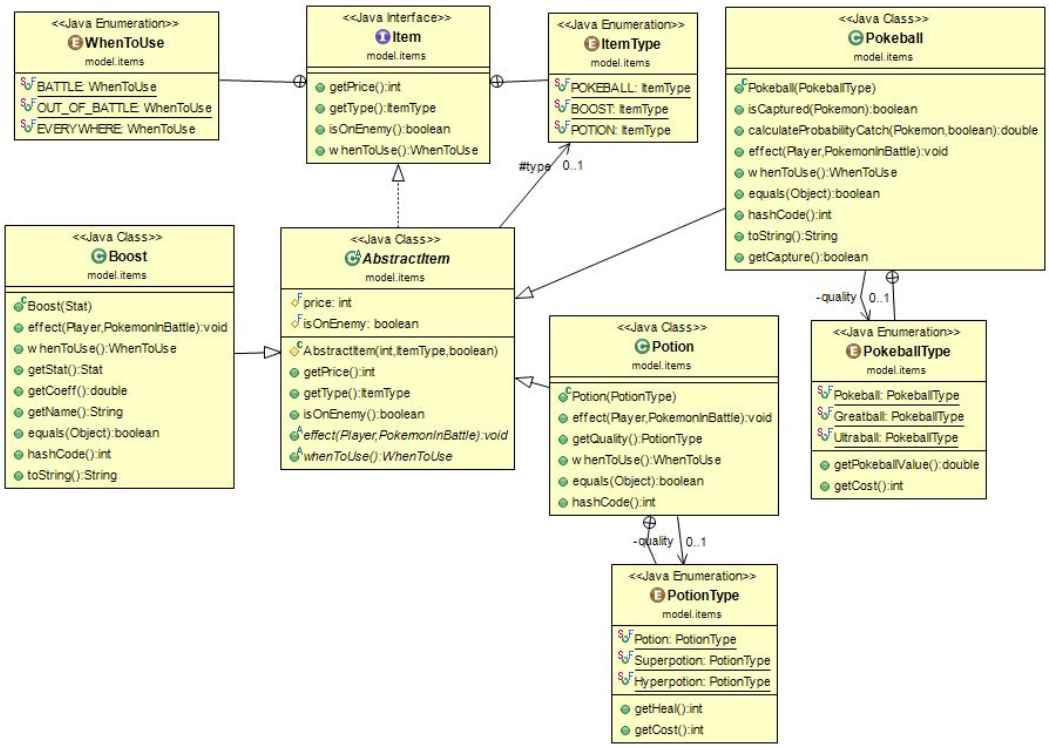
I vari ***Character*** sono personaggi con cui il ***Player*** può interagire parlandoci, in alcuni casi mostrando solo un messaggio e in altri facendo partire una lotta fra ***Pokemon***. Ecco lo schema:



Anche qui è stata seguita la sequenza Interface, Abstract Class, Concrete Classes, per fattorizzare il codice in comune fra tutte le classi e per aumentare la separazione.

- ***NPC*** è semplicemente un personaggio con una frase che dice ogni volta che viene interpellato;
- ***Trainer*** è un personaggio con cui il ***Player*** combatte appena dopo averci parlato, come lo stesso ***Player***, anche lui possiede una sua ***Squad*** con dentro fino a 6 ***Pokemon***;
- ***GymLeader***, estendendo ***Trainer*** possiede tutte le sue caratteristiche, in più, quando sconfitto, da una medaglia (badge) al ***Player***, aumentando di 1 il totale. Le medaglie servono per aprire delle porte e per sbloccare nuovi oggetti dal ***PokeMarket***;
- ***PokeMarket*** è un ***NPC*** particolare che può vendere ***Item*** al ***Player***. La varietà di oggetti che può vendere aumenta con il progresso delle medaglie.

Infine gli ***Item*** sono consumabili ed hanno la seguente struttura:

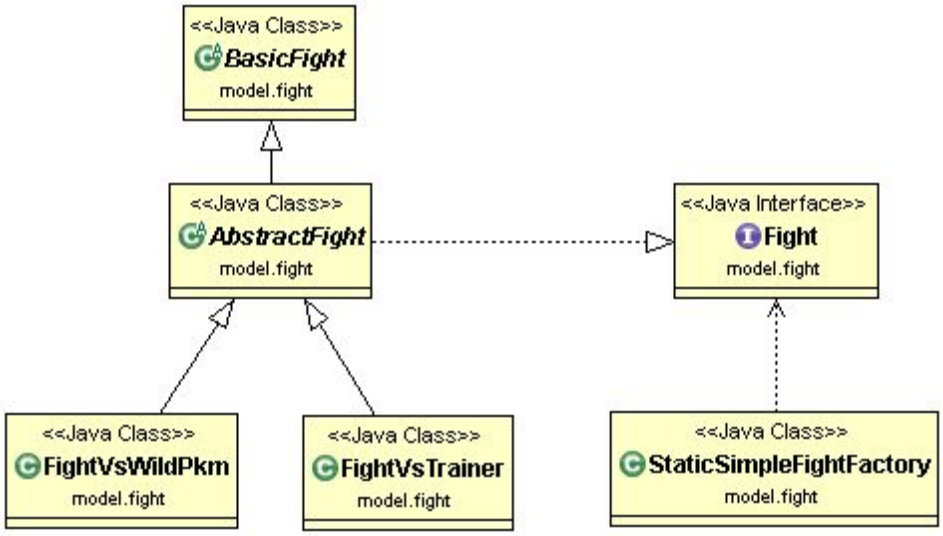


Ancora una volta ho utilizzato Interface, Abstract Class, Concrete Classes per raccogliere il codice condiviso. ***Item*** contiene anche due enumerazioni per stabilire quando è possibile utilizzarlo(***WhenToUse***) e per definire il tipo (***ItemType***).

- ***Boost*** è un ***Item*** utilizzabile su un ***Pokemon*** in battaglia e aumenta di un moltiplicatore la ***Stat*** specificata nel costruttore
- ***Potion*** è utilizzabile sia in battaglia che fuori e cura gli HP di un ***Pokemon*** a seconda del valore specificato nel ***PotionType*** della ***Potion***
- ***Pokeball*** contiene tutti i metodo per calcolare la probabilità di cattura e, se usata, per provare a catturare un ***Pokemon*** selvatico.

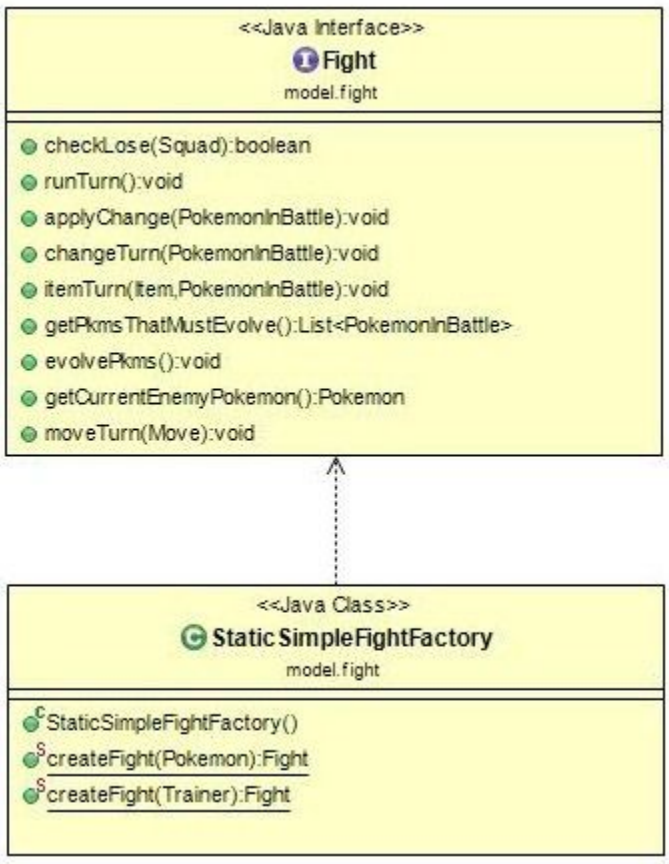
Il package *model* è stato organizzato in subpackages per avere una visione del complesso e per garantire una facile navigazione fra le classi.

Fight: Davide Bondi



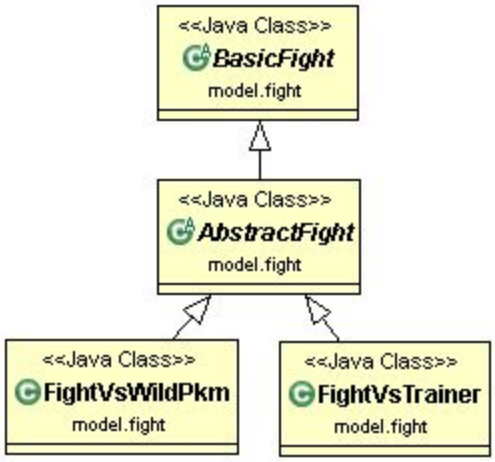
Il combattimento si occupa di gestire e calcolare tutto quello che succede durante la lotta tra pokemon. Ogni classe che realizza un combattimento deve implementare l'interfaccia ***Fight***.

Il Controller crea un combattimento chiamando il metodo ***createFight,*** presente nella classe ***StaticSimpleFightFactory,*** che implementa il pattern di una simple factory statica. Inserendo il nemico da affrontare come input al metodo viene fatta ritornare una istanza relativa. Da ciò ne deriva che si devono avere tante versioni di ***createFight*** pari alle classi concrete che implementano ***Fight***, che a loro volta saranno equivalenti alle tipologie di nemici affrontabili.



Come descritto in precedenza, ogni volta che si entra in un combattimento, i pokemon che combattono sono quelli che si trovano nella prima posizione di ogni squadra coinvolta. Il combattimento si svolge a turni e in genere in un turno si svolgono l'azione alleata e quella nemica. L'utente ha la possibilità di scegliere tra quattro opzioni: Fight, Bag, Squad, e Run. Un combattimento ha fine se il ***Player*** riesce a fuggire oppure quando tutti i pokemon di una squadra sono esausti.

I metodi dell'interfaccia ***Fight*** si possono suddividere in base alla logica con cui vengono chiamati. La prima suddivisione riguarda i metodi ***moveTurn***, ***itemTurn***, ***changeTurn*** e ***runTurn***, i quali sono chiamati dal controller in seguito alla scelta di azione effettuata dall'utente. Pertanto gli eventi che innescano tali metodi hanno origine sempre e solo nella view. Gli altri metodi invece vengono utilizzati dal controller e/o dall'istanza fight stessa per svolgere varie operazioni che non vengono scelte dall'utente.



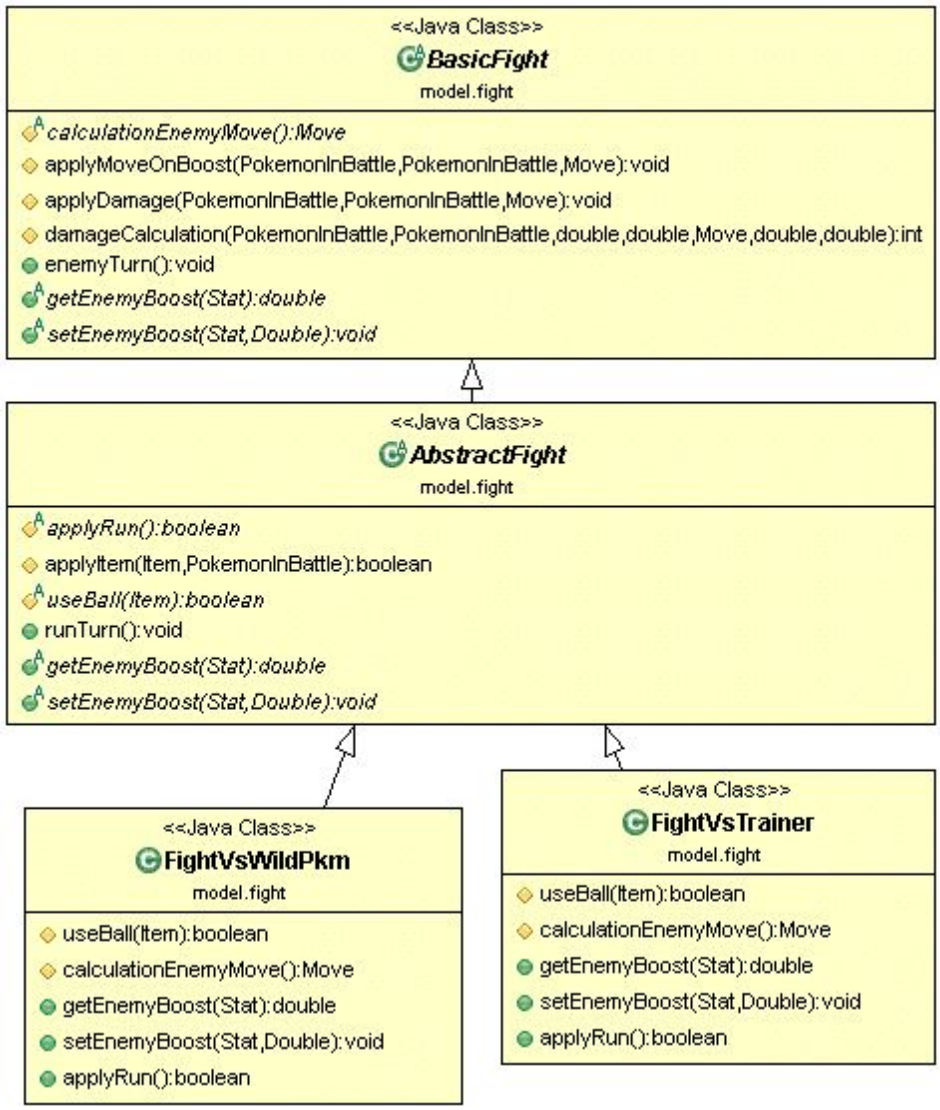
La classe ***BasicFight*** include i campi necessari alla gestione dei parametri del ***Player*** e l'implementazione della maggior parte delle procedure di base intrinseche che vengono utilizzate all'interno dei metodi di ***Fight***.
La classe ***AbstractFight*** completa ulteriormente il codice fornito da ***BasicFight,*** implementando ***Fight,*** introduce nuovi metodi che gestiscono le operazioni fornite dalla sopra classe e gestisce la maggior parte della logica di battaglia.
Le uniche cose che non vengono trattate in queste due classi, sono le operazioni specifiche che riguardano il nemico, le quali vengono realizzate direttamente nelle estensioni concrete ***FightVsWildPkm*** *e* ***FightVsTrainer***.
Da questa struttura si denotano diversi possibili livelli di estensione. Se si desidera aggiungere un'ulteriore classe che implementi un combattimento contro un nuovo nemico mantenendo la logica fornita, occorrerà estendere ***AbstractFight***. Se si desidera modificare la lotta a livello gestionale basterà implementare nuovamente ***Fight*** e in tal caso, si potrà scegliere di estendere ***BasicFight*** per sfruttare le meccaniche base.
Un'altra caratteristica riguardante le classi astratte, è l'utilizzo del pattern template method. Nella pagina seguente viene fornito un diagramma UML che, per facilitare la comprensione, mostra solamente i metodi template e le relative sotto operazioni che vengono implementate dalle sotto classi. In particolare:

in ***BasicFight***:

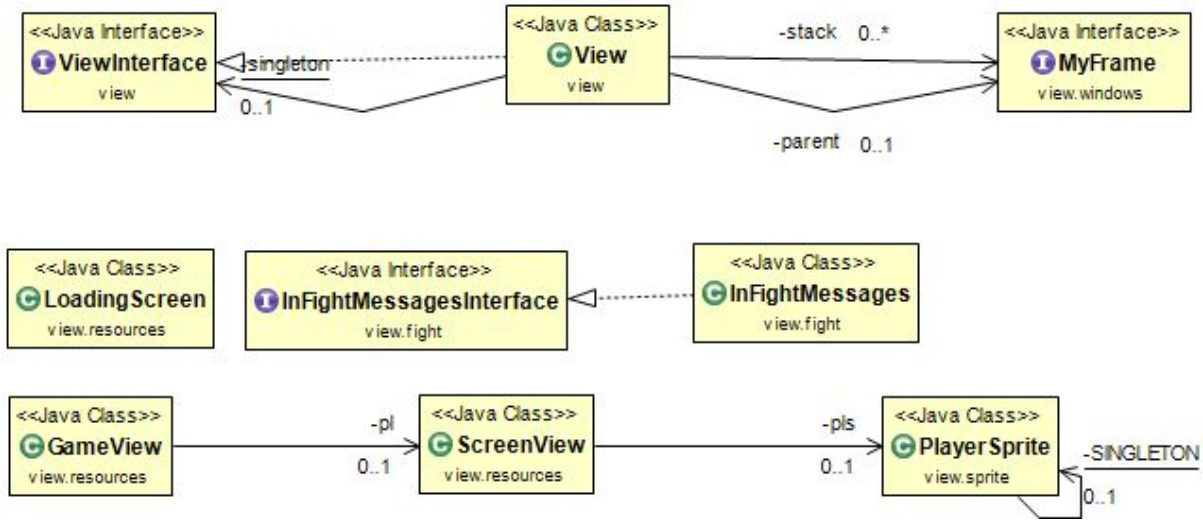
- ***applyMoveOnBoost*** richiama ***setEnemyBoost*** *e* ***getEnemyBoost;***
- ***applyDamage*** richiama ***getEnemyBoost;***
- ***damageCalculation*** richiama ***getEnemyBoost;***
- ***enemyTurn*** richiama ***calculationEnemyMove.***

in ***AbstractFight***:

- ***applyItem*** richiama ***useBall;***
- ***runTurn*** richiama ***applyRun.***



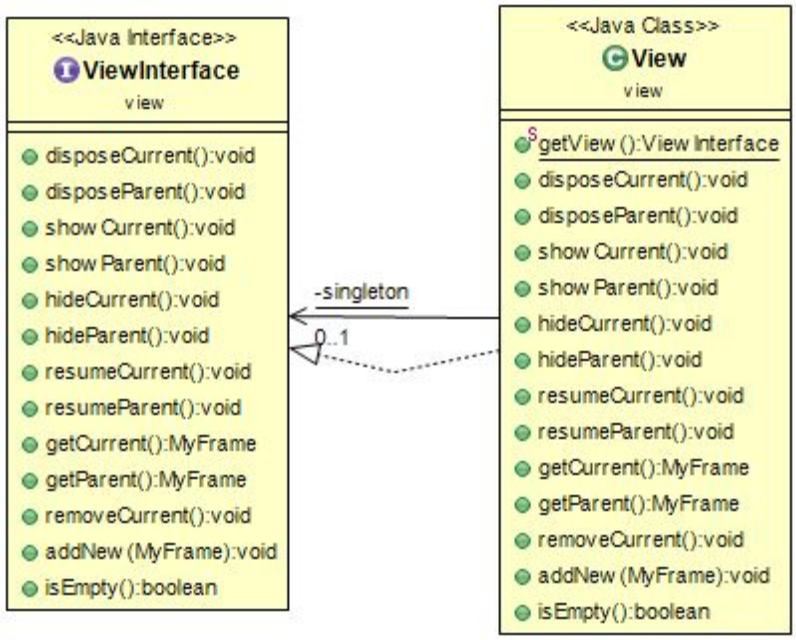
View: Daniel Veronesi



Per motivi di spazio sono state omesse le classi che implementano ***MyFrame***.

La View, sviluppata sulla base del pattern architetturale MVC, mostra la rappresentazione grafica dello stato del Model e si occupa di gestire l'interazione con l'utente. Per la realizzazione è stato scelto di usare quasi interamente la libreria Swing, al fine di sviluppare i concetti studiati durante il corso.
La View non è a conoscenza di alcun dettaglio implementativo relativo al Model e ogni informazione inerente esso è ottenuta dal model attraverso il Controller.

ViewIntarface e *View*

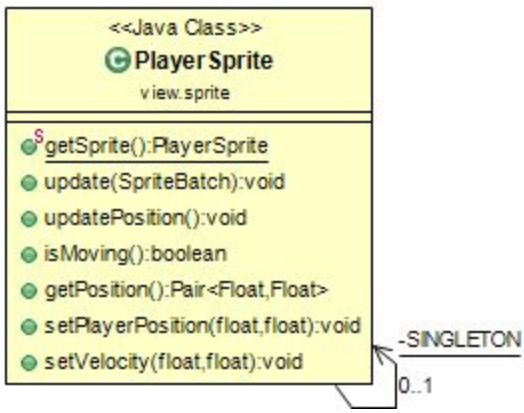


Data la gestione di numerose finestre all'interno del gioco, si sono fatte alcune scelte durante la progettazione. Al momento, avendo usato Swing come libreria, si è scelto di creare una *stack*, in modo da regolare l'ordine di comparsa dei menù ed eliminare gli eventuali errori che si sarebbero potuti manifestare. Quando il gioco è in pausa, infatti, l'utente è impossibilitato a compiere alcuna azione all'infuori del menù appena aperto; quest'ultimo deve anche essere l'unica finestra navigabile in tale momento.

All'inizio si era pensato di usare un *CardLayout*, in modo da poter anche risparmiare sul codice, ma alcuni problemi, come la grandezza molto diversa dei vari menù, hanno fatto propendere di più per l'utilizzo della pila.

Avendo fatto questa scelta, si è scelto di creare le finestre usando dei *JWindow*, in modo tale da non creare altre schede ogni volta che si apriva un menù, così da evitare di avere numerose icone nella barra applicazioni dell'utente.

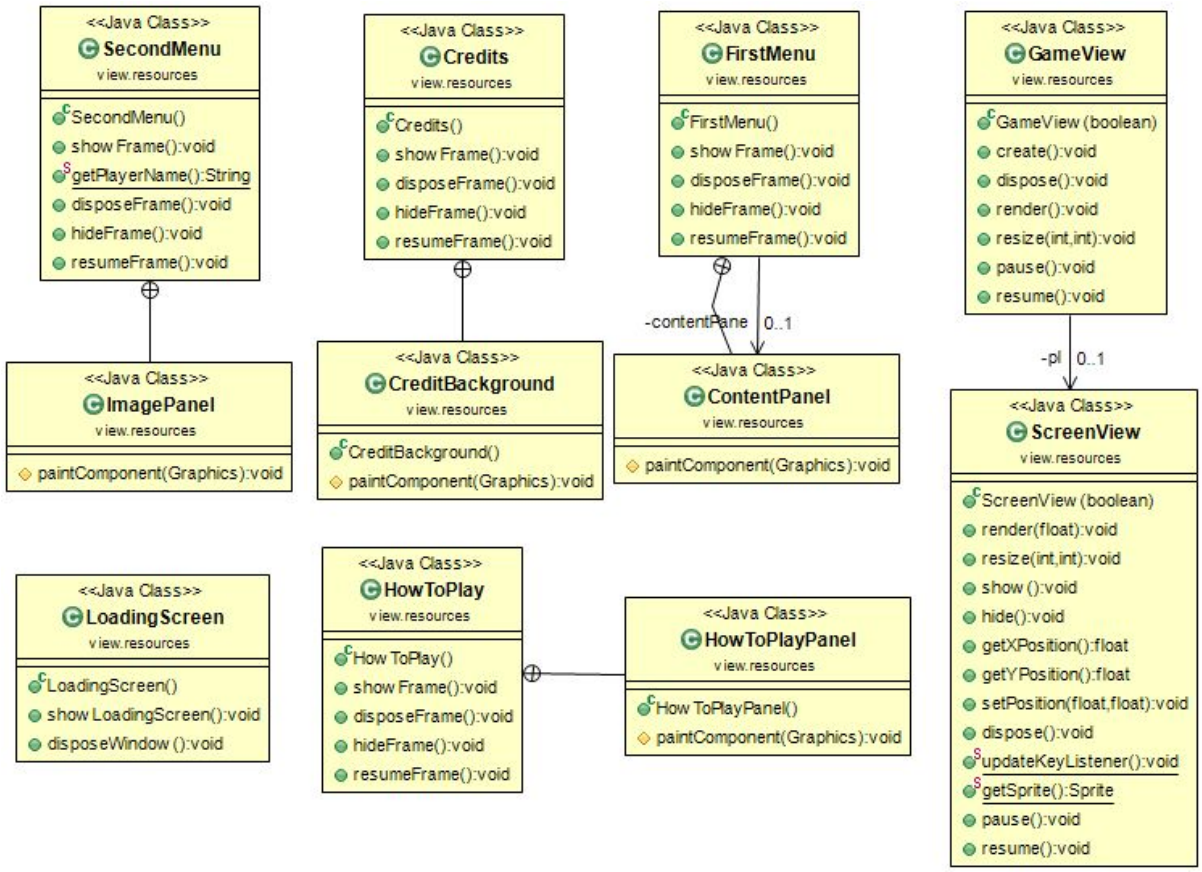
Sprite



Il package Sprite controlla tutto ciò che riguarda l'animazione del personaggio giocabile. L'uso di uno sprite sheet piuttosto che di una gif animata permette di accedere allo sprite del trainer con un solo caricamento, mentre l'utilizzo del Singleton consente di creare una e una sola istanza e di fornirle un accesso globale.

La classe ***PlayerSprite*** mette poi a disposizione dei metodi per accedere ai singoli frame presenti all'interno dello sprite sheet. Tramite le coordinate d'interesse setta la velocità del personaggio e il movimento della sua immagine. La classe, inoltre, aggiorna la posizione del personaggio ogni volta che questo entra in una porta o in un teletrasporto. Tutti i dati sono presi dal controller.

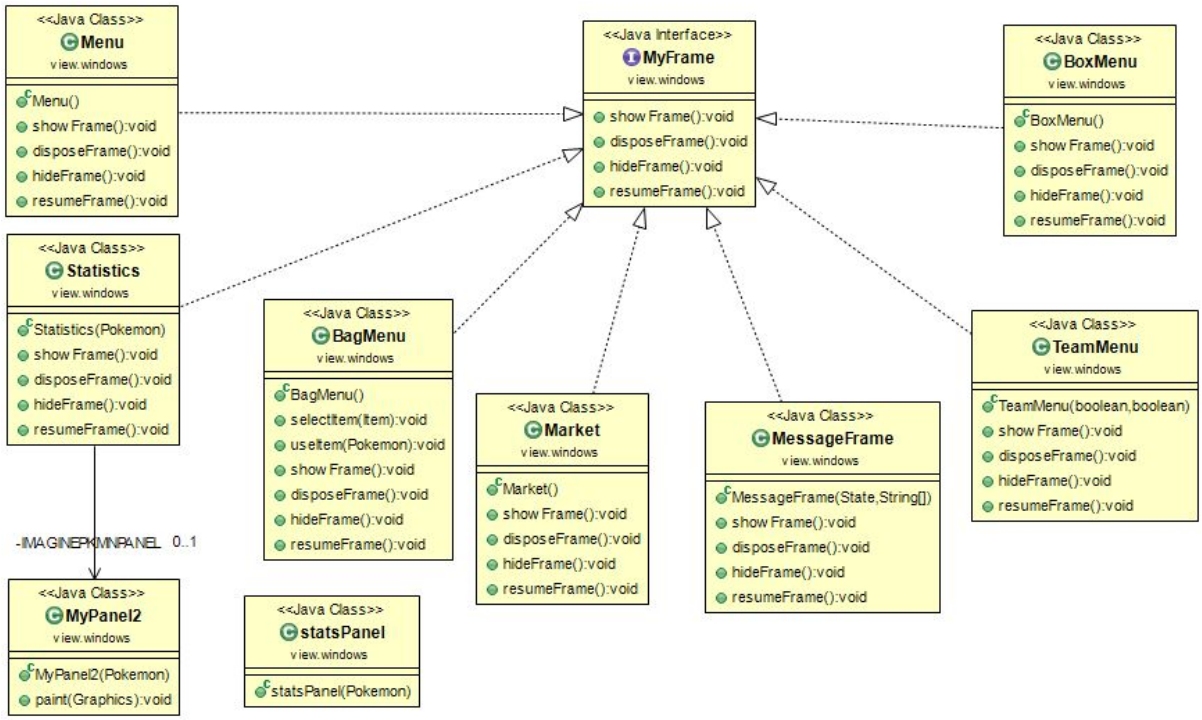
Resources



Le classi ***FirstMenu***, ***SecondMenu***, ***Credits*** e ***HowToPlay*** implementano tutte l'interfaccia ***MyFrame***, che per motivi di chiarezza non è stata inserita in questo diagramma.

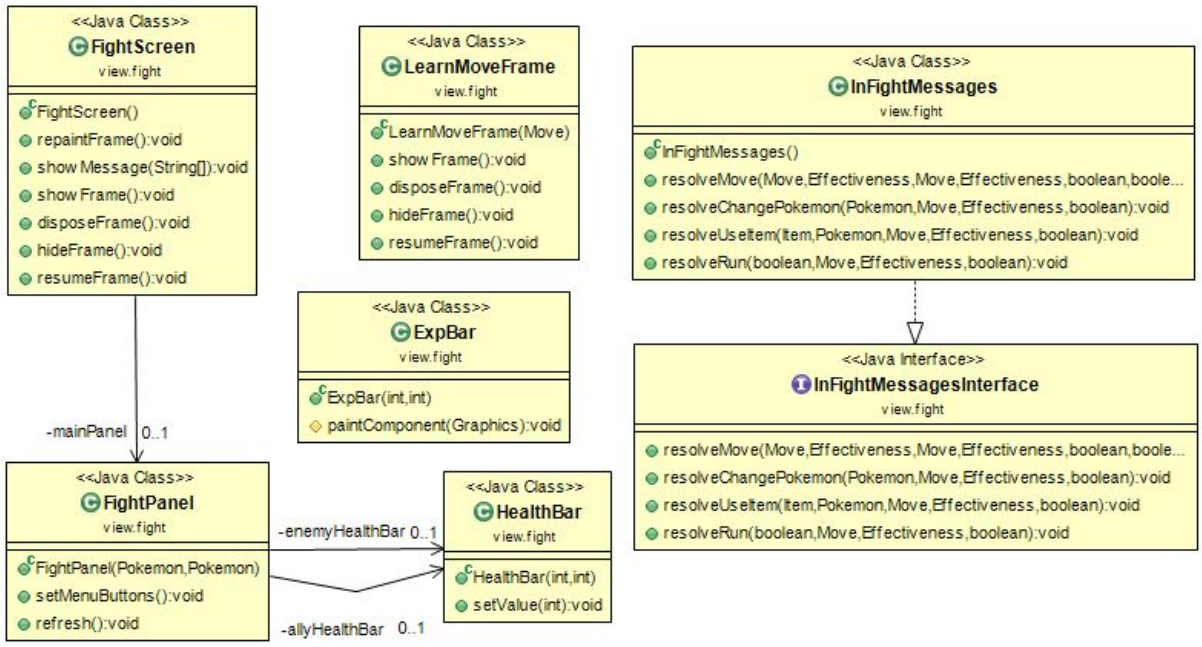
Il package *Resource* si occupa della visualizzazione del menu principale (***FirstMenu***) e di tutte le finestre aperte da esso.
Nello specifico, le possibili scene che possono essere consultate sono:
HowToPlayPanel*,* che si occupa di mostrare i comandi di gioco;
Credits*,* che mostra gli autori che hanno preso parte al progetto e le loro mansioni,
SecondMenu*,* aperto dal pulsante *new game*, che permette di cominciare una nuova partita scegliendo il nome e il pokémon iniziale col quale partire;
infine **Continue** che, se il controller trova un salvataggio di gioco esistente, permette alla view di dare la possibilità all'utente di continuare la partita dall'ultima volta che ha salvato.
Selezionando una delle ultime due opzioni, si lancia il gioco vero e proprio, e quindi si carica la mappa. Le classi che si occupano di prendere e mostrare tutte le informazioni di questo frame sono ***GameView*** e ***ScreenView,*** che utilizzano la libreria *libgdx.* Tra i metodi che implementa c'è ***updateKeyListener***. Esso prende come parametro un ***InputListener*** dal controller e lo setta come input listener corrente. Ciascuno di essi permette la gestione dei tasti da parte dell'utente.

Windows



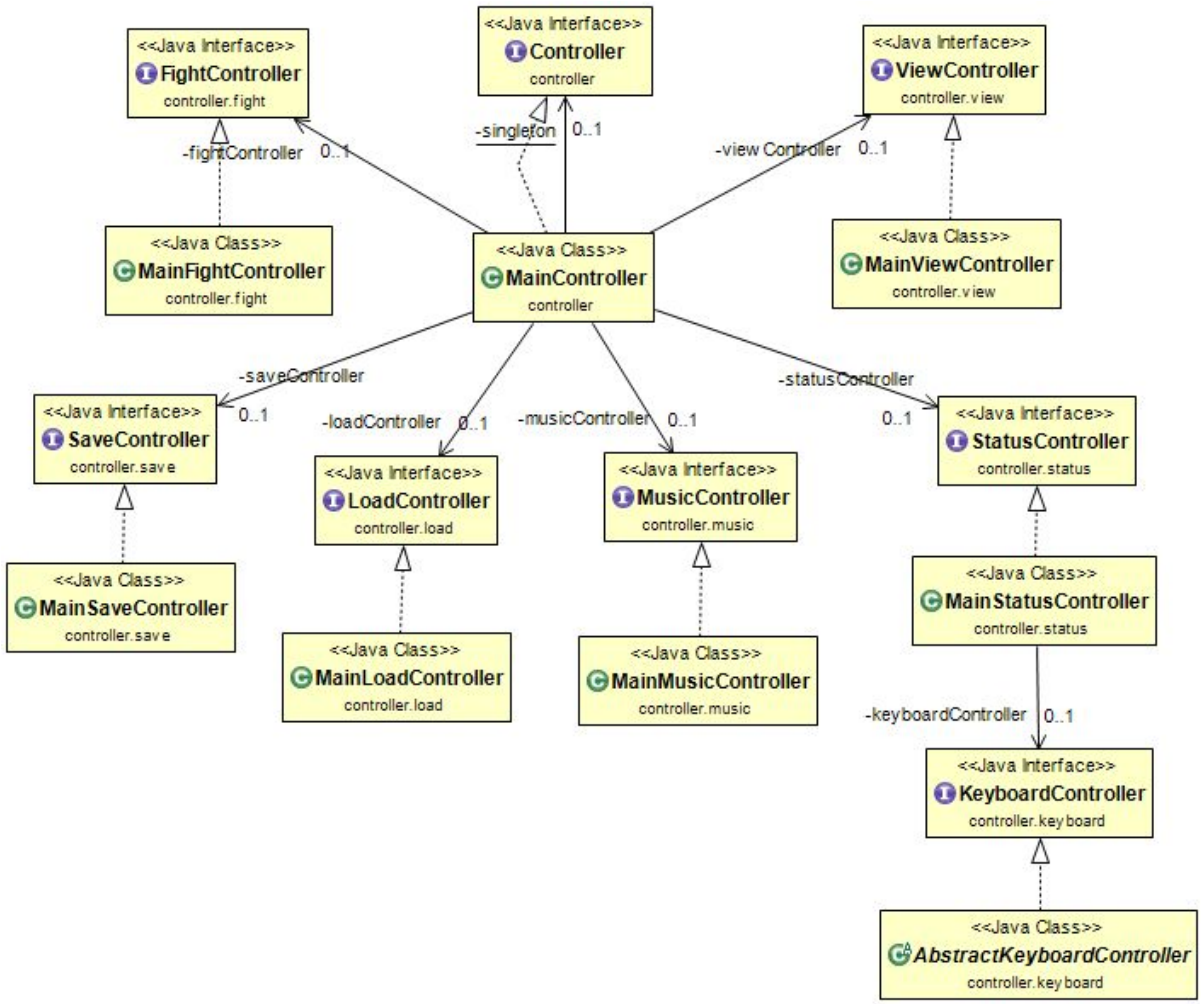
Il package ***Windows*** contiene dentro di sé tutte le informazioni delle varie finestre del gioco vero e proprio, escludendo la parte riguardante il fight. Molto importante è la classe ***MessageFrame***, che permette di mostrare tutti i messaggi non riguardanti i turni di battaglia.
La base dello stack di JWindows è ***Menu***, apribile con il tasto [ESC], esso contiene i pulsanti per navigare in tutti i menù secondari o per dare indicazioni su alcuni dati di gioco.
BoxMenu mostra tutti i pokémon all'interno del box ed è programmato in modo tale da ricevere un pokémon sia manualmente, tramite un pulsante in ***TeamMenu***, sia automaticamente nel caso la squadra sia già composta da 6 membri. Se in un futuro si volesse rendere questo menù apribile dal *pokémon center* come nel gioco originale, basterà semplicemente rimuovere il pulsante da ***Menu*** (che si riproporzionerà automaticamente) e rendere il ***BoxMenu*** apribile tramite il tasto di interazione [INVIO] mentre il player è davanti al tile del computer.
TeamMenu, invece, apre la finestra della squadra e permette di strutturarla a proprio piacere. Questa classe è la stessa sia se aperta da ***Menu*** che dal ***FightScreen*** in modo da non dovere duplicare inutilmente del codice. Oltretutto, se in futuro si volesse cambiare il numero di pokémon disponibili per la battaglia, il ***TeamMenu*** sarebbe già aggiornato senza alcuna modifica poiché nel ciclo for definito al suo interno si prende direttamente la grandezza prefissata dal model.
Tutte queste caratteristiche di estensione e di risparmio del codice, inoltre, si applicano anche alla classe ***BagMenu***, che ha la funzione di aprire la schermata che rende possibile l'uso degli oggetti. Al limite, in questo caso, potrebbe essere necessaria l'aggiunta di una *scrollbar* per rendere più agevole l'utilizzo della finestra, nell'eventualità che si aggiunga un numero molto elevato di nuovi oggetti.

Fight



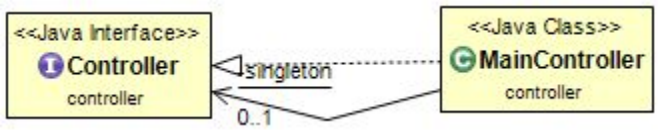
Infine, data la complessità e il grande numero di informazioni richieste dalla battaglia, si è deciso di creare un package interamente su questa parte del gioco.
La base di tutto è il ***FightScreen***, che è la base su cui viene implementato il ***FightPanel***. Questa finestra contiene tutte le opzioni che il giocatore può decidere di compiere durante la battaglia. Come già detto in precedenza, tra queste scelte ci sono l'apertura del ***BagMenu*** e del ***TeamMenu***. Oltre a queste due, però, c'è la possibilità di fuggire dalla lotta, oppure di lottare e sconfiggere il pokémon avversario. Tutte le informazioni dello scontro vengono prese dal model tramite il controller. Per la visualizzazione dei messaggi e il proseguimento della battaglia sono state create la classe e l'interfaccia ***InFightMessages*** e ***InFightMessagesInterface*** che permettono di lanciare su schermo il giusto messaggio per ogni tipo di azione scelta all'inizio del turno.
Sempre per il ***Fightpanel***, sono state poi create altre 2 classi: ***HealthBar*** e ***ExpBar***.
La prima contiene all'interno la creazione e la colorazione della barra della salute del pokémon alleato e del pokémon nemico. La seconda, invece, contiene la sola creazione della barra dell'esperienza dedicata al pokémon usato dal giocatore.
Tutti i cambiamenti che accadono alla fine di ogni turno di battaglia, vengono mostrati grazie a *refresh(),* che permette l'aggiornamento di ogni dato e di ogni immagine.
Infine, nel caso un pokémon dovesse imparare una nuova mossa mentre ne conosce già 4, è stata creata la classe ***LearnMoveFrame,*** che permette all'utente di avere una finestra a parte in modo da avere tutte le informazioni necessarie per scegliere se dimenticare una vecchia mossa o rifiutarsi di imparare la nuova.

Controller: Michael Camporesi



Per motivi di spazio non sono stati inclusi in questo diagramma i vari ***KeyboardControllers*** che estendono tutti ***AbstractKeyboardController***.

Controller e *MainController*

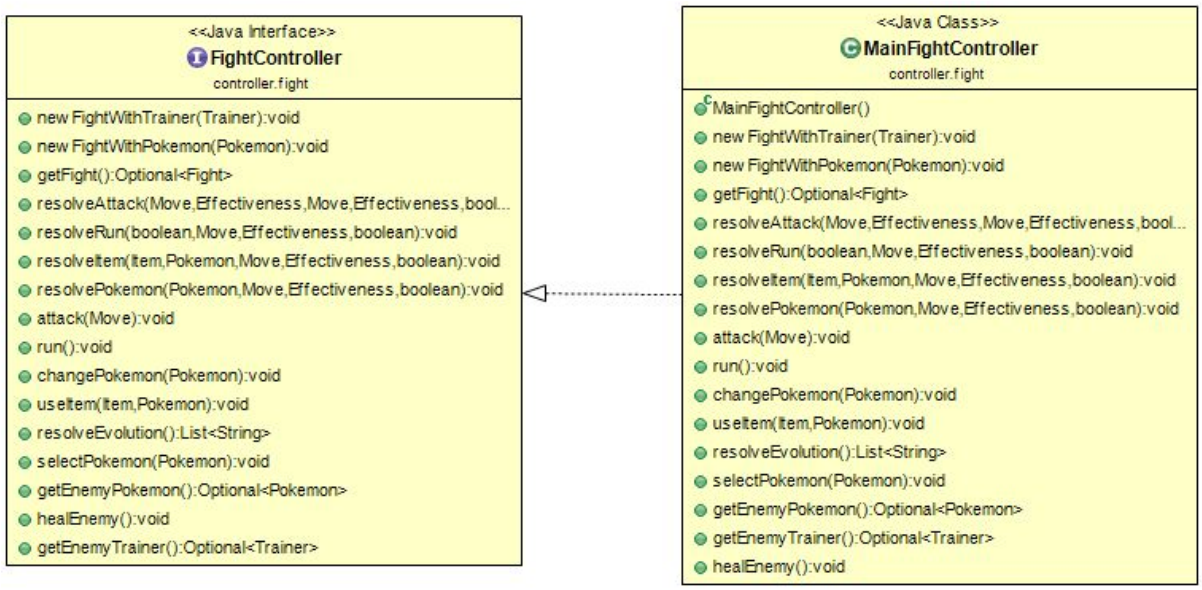


Per motivi di spazio sono stati omessi da questo grafico tutti i metodi e i campi.

La classe principale del controller è ***MainController***, che estende l'interfaccia ***Controller***.
MainController utilizza il pattern di programmazione SINGLETON, il quale consiste nell'avere una sola istanza statica della classe che può essere chiamata in qualsiasi punto del programma tramite il metodo ***getController***.
Questa viene inizializzata alla prima chiamata di tale metodo, tramite un costruttore privato.

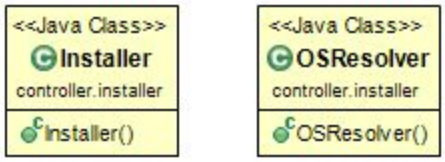
La classe ***MainController*** contiene al suo interno un'istanza di ogni controllore principale, e fornisce tutti i metodi, dei vari controllori, richiesti all'interno dell'applicazione; cosicché invece che avere all'interno del programma diverse istanze dei vari controllori, uno ed uno solo di ciascuno dei controllori viene conservato dentro la classe, fornendo solo i metodi necessari.
Ciò permette inoltre di nascondere metodi sensibili all'utente.
La classe contiene inoltre un'istanza del ***Model***, e fornisce i metodi del model richiesti all'interno dell'applicazione. Questo garantisce che la view non agisca direttamente sul model, ma passi sempre attraverso il controller, l'unico che opera direttamente sul model.
All'interno della classe viene inoltre conservata la ***TiledMap***, la mappa principale del gioco, che viene passata al model al momento dell'inizializzazione.

FightController e *MainFightController*



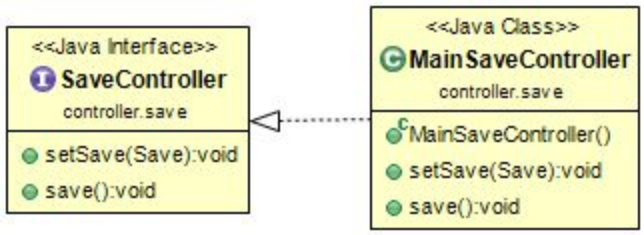
La classe ***MainFightController*** si occupa di gestire il combattimento contro un ***Trainer*** o contro un ***Pokemon*** selvatico.
I metodi ***newFightWithTrainer*** e ***newFightWithPokemon*** servono ad inizializzare il combattimento, i metodi ***resolveAttack***, ***resolveRun***, ***resolveItem*** e ***resolvePokemon*** vengono chiamati dal model, dopo aver calcolato l'andamento del turno, per passare alla view le informazioni da mostrare a video. Infatti questi metodi chiamano dei metodi sulla view, passando i dovuti parametri, affinché questa mostri i messaggi necessari all'utente per capire come effettivamente si è risolto il turno.
I metodi ***attack***, ***run***, ***changePokemon*** e ***useItem*** vengono chiamati dalla view per comunicare la scelta del giocatore durante un turno di combattimento. Questi metodi passano al model le informazioni necessarie per calcolare come si risolverà il turno.
Il metodo ***resolveEvolution*** serve a controllare se ci sono pokémon che si possono evolvere, e nel caso evolverli.
Il metodo ***selectPokemon*** serve per comunicare al model il pokémon selezionato per sostituire quello corrente esausto. Viene chiamato dalla view che comunica così il pokémon scelto dall'utente.
Infine, il metodo ***healEnemy*** serve a curare un pokémon leggendario, o i pokémon di un allenatore dopo la battaglia in caso il giocatore sia risultato sconfitto.

Installer e *OSResolver*



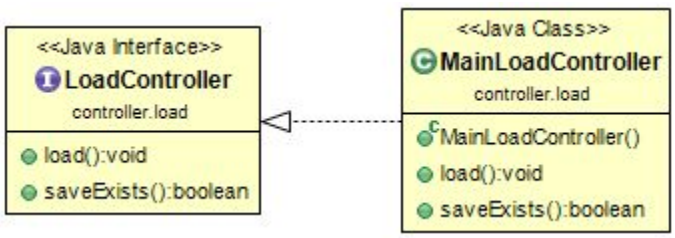
La classe ***Installer*** si occupa di installare tutte le risorse necessarie al gioco in una cartella nella home directory dell'utente.
Tutti i metodi importanti sono stati lasciati privati.
La classe ***OSResolver*** serve a modificare una libreria nel caso il sistema operativo corrente sia OSX. In tale sistema, infatti, una libreria necessaria al corretto funzionamento del gioco è presente con un'estensione diversa da quella predefinita. Il costruttore della classe, se rileva che il sistema operativo è OSX, grazie a metodi privati modifica l'estensione di tale libreria.

SaveController e MainSaveController



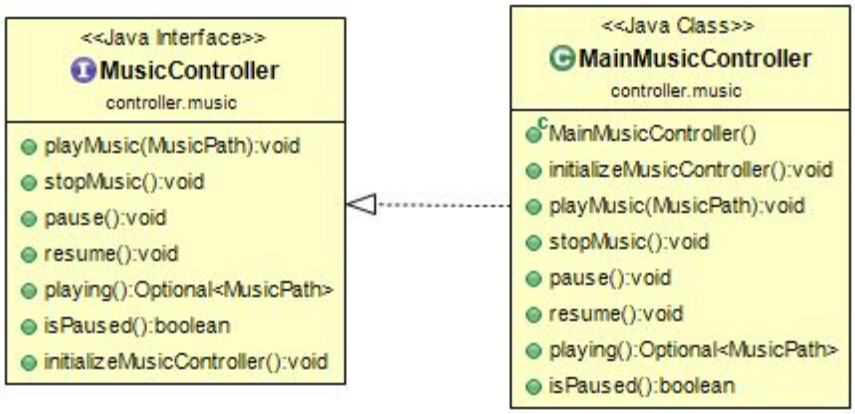
La classe ***MainSaveController*** si occupa di salvare i dati necessari su un file, al fine di permettere al giocatore di caricare una partita precedente.
Il metodo ***setSave*** recupera dal model i dati da salvare, mentre il metodo ***save*** salva i dati.
I dati vengono salvati in un file di tipo XML.

LoadController e *MainLoadController*



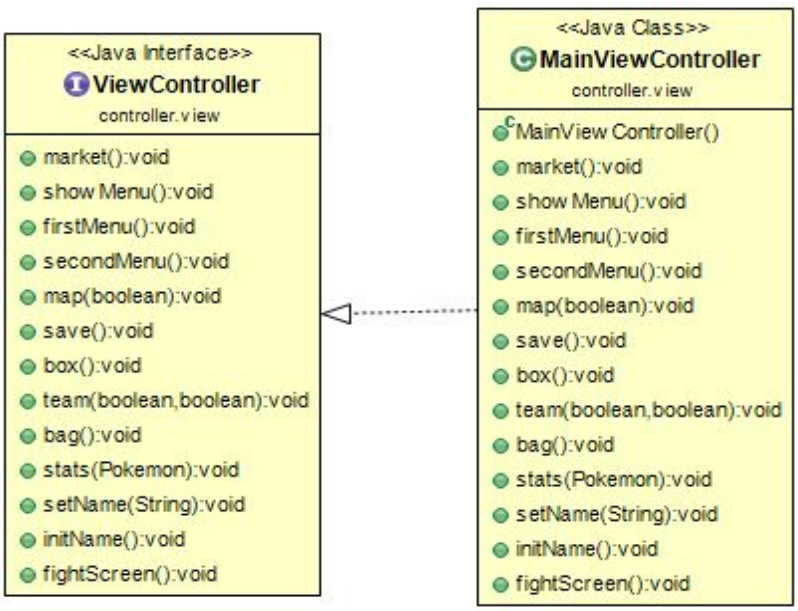
La classe ***MainLoadController*** si occupa di caricare i dati necessari dal file di salvataggio.
Il metodo ***load*** carica i dati dal file e li passa al model; mentre il metodo ***saveExists*** ritorna un valore booleano che indica il fatto che il salvataggio esista oppure no.

MusicController e *MainMusicController*



Questa classe si occupa di controllare la musica all'interno del gioco. I file musicali sono in formato MP3, e per la riproduzione, come concordato in fase di analisi, viene usala la libreria LIBGDX.
Il metodo ***initializeMusicController*** si occupa di inizializzare il controllore, caricando fin da subito tutte le canzoni. In questo modo, nonostante ciò causi un tempo di caricamento all'inizio dell'applicazione, si evita che, al passaggio da una canzone all'altra, ci siano caricamenti che rallenterebbero il gioco.
I metodi ***playMusic*** e ***stopMusic*** si occupano rispettivamente di far partire o stoppare la riproduzione di una canzone, mentre i metodi ***pause*** e ***resume*** servono a mettere in pausa e riprendere la riproduzione della canzone corrente.
Mentre i primi servono per "caricare" e "scaricare" una canzone dal buffer audio corrente, gli altri consentono semplicemente di "mettere in pausa" e riprendere la riproduzione della canzone corrente, senza rimuoverla dal buffer.
Il metodo ***playing*** restituisce la canzone corrente, e quello ***isPaused*** restituisce un booleano che permette di capire se la riproduzione è in pausa oppure no.

ViewController e *MainViewController*

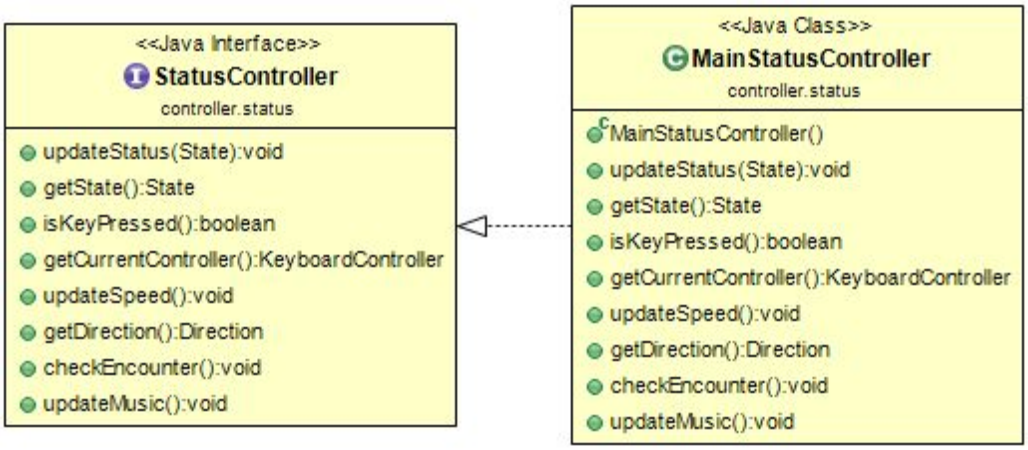


Questa classe serve ad aprire i menù della view.

- ***market***: apre la schermata del market;
- ***showMenu***: apre il menù di gioco;
- ***firstMenu***: apre la prima schermata di gioco;
- ***secondMenu***: apre la seconda schermata di gioco, quella della scelta del nome del giocatore e del pokémon iniziale;
- ***map***: apre la mappa di gioco. Il booleano indica se si tratta di una nuova partita o se si vuole caricare una partita precedente;
- ***save***: ordina il salvataggio di gioco;
- ***box***: apre il menù del box;
- ***team***: apre il menù del team. I parametri sono richiesti dalla view, a cui dovranno essere passati. Per ulteriori informazioni riferirsi alla javadoc;
- ***bag***: apre il menù degli strumenti;
- ***stats***: apre il menù delle statistiche;
- ***fightScreen***: apre la schermata di battaglia.

Il metodo ***setName*** setta il nome del giocatore. Questo metodo è chiamato dalla view dopo aver ricevuto il nome scelto dall'utente in caso di nuova partita.
Infine il metodo ***initName*** inizializza il nome corrente nel model.

StatusController e *MainStatusController*

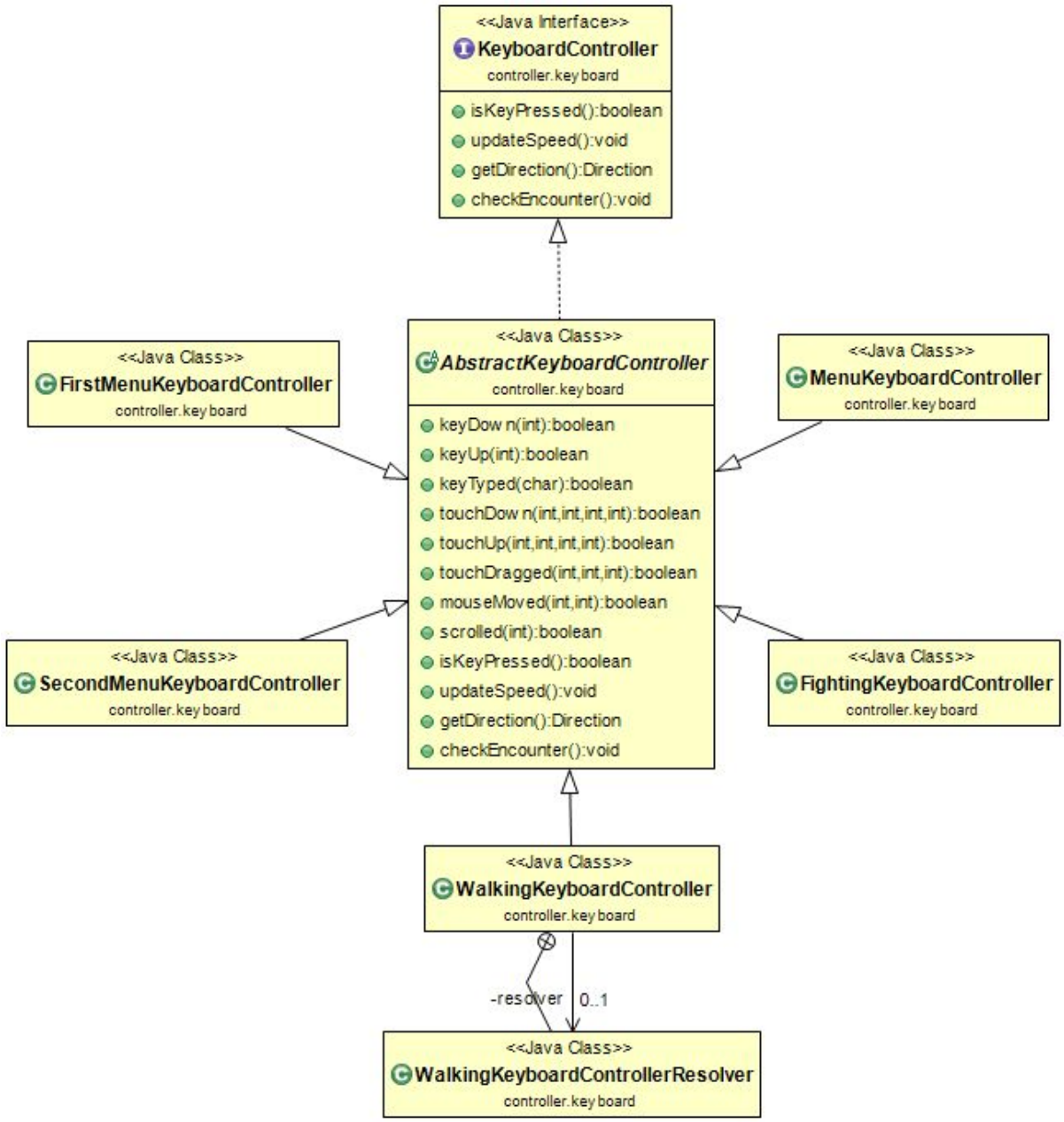


La classe ***MainStatusController*** si occupa di gestire lo stato di gioco.
Il metodo principale è ***updateStatus***, che aggiorna lo stato del gioco e, a seconda dello stato passato come argomento, apporta le dovute modifiche al gioco.
La classe contiene anche un ***KeyboardController***, che viene cambiato ogni volta che cambia lo stato, a seconda dello stato selezionato.
Il metodo ***isKeyPressed*** ritorna un booleano che indica se è premuto o meno uno degli hotkeys della tastiera.

Metodo fondamentale è ***updateSpeed***, che viene chiamato dalla view per richiedere l'aggiornamento dello stato atuale del gioco. Questo aggiornamento, in caso sia premuto un particolare tasto, aggiorna la view in base al tasto premuto, e se questo è un tasto "di movimento", a quel punto la view mostra l'animazione del movimento e sposta il personaggio sulla mappa. Allo stesso tempo il controller aggiorna la posizione nel model, mantenendoli pertanto sincronizzati.

Il metodo ***checkEncounter*** controlla se, alla posizione attuale, si verifica un incontro con un pokémon selvatico. In caso affermativo, inizia lo scontro.
Infine il metodo ***updateMusic*** aggiorna la musica, controllando nella mappa di gioco la canzone prevista nella zona attuale e, se non è quella correntemente in riproduzione, la cambia.

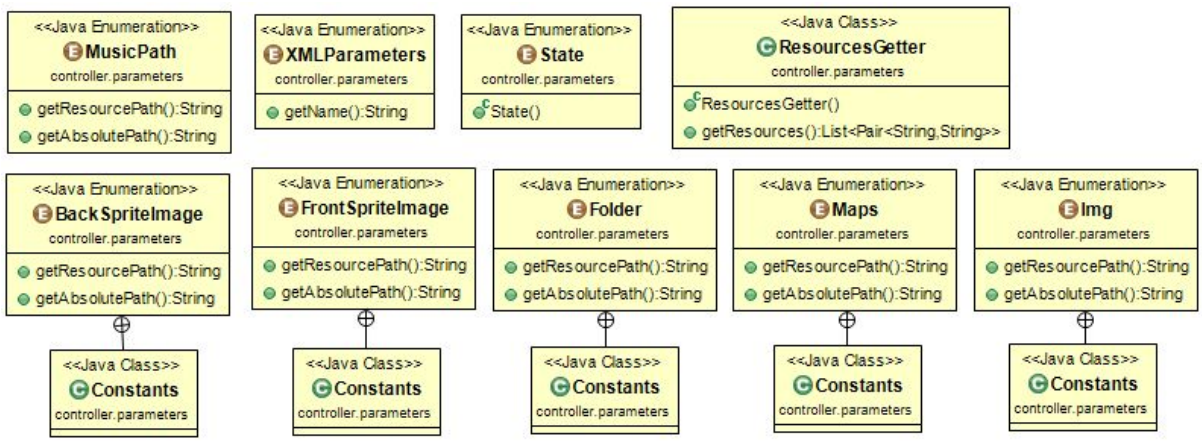
KeyboardControllers



Tutti i ***KeyboardControllers*** estendono la classe astratta ***AbstractKeyboardController***.
Questi controllori servono a gestire l'input da tastiera dell'utente.
KeyboardController estende l'interfaccia della LIBGDX ***InputProcessor***.
I metodi indicati in ***AbstractKeyboardController*** che non sono espressamente indicati nell'interfaccia ***KeyboardController,*** sono quelli appartenenti all'interfaccia ereditata dalla LIBGDX.

WalkingKeyboardController è il controllore principale e, essendo di una complessità notevole, al suo interno contiene una classe innestata, ***WalkingKeyboardControllerResolver***, che contiene metodi privati utilizzati per vari scopi dal controllore.
Il metodo ***isKeyPressed*** restituisce un booleano che indica la pressione o meno di uno dei tasti utili al gioco, ***updateSpeed*** aggiorna la velocità dello sprite del giocatore, ***getDirection*** restituisce la direzione corrente del giocatore e ***checkEncounter*** controlla la presenza o meno di un pokémon selvatico alla posizione corrente.
Ad ogni cambio di stato, lo ***StatusController*** assegna alla mappa un diverso controllore, a seconda delle funzioni che devono essere svolte.

Parameters



Il package ***Parameters*** contiene diverse enumerazioni:

- ***MusicPath*** contiene i percorsi relativi ed assoluti di tutte le canzoni;
- ***XMLParameters*** contiene le stringhe costanti utilizzate nel salvataggio/caricamento;
- ***State*** contiene gli stati di gioco;
- ***BackSpriteImage*** e ***FrontSpriteImage*** contengono i percorsi per arrivare agli sprite dei pokémon;
- ***Folder*** contiene i percorsi alle cartelle di gioco;
- ***Maps*** contiene i percorsi per arrivare ai file relativi alla mappa;
- ***Img*** contiene i percorsi alle immagini del gioco.

Alcune enumerazioni contengono al loro interno una classe privata, ***Constants***, che contiene le costanti necessarie all'enumerazione stessa.
ResourceGetter è una classe studiata per ritornare, sotto forma di lista, le coppie di path delle varie risorse da installare.

3 Sviluppo

3.1 Testing

Si descrivono molto brevemente i componenti che si è deciso di sottoporre al test automatizzato:

- ***TextBox*** contiene i test relativi al deposito e ritiro di pokémon con relativa gestione delle eccezioni;
- ***TestCaptureRate*** contiene i test specifici sulla probabilità di cattura, che costituisce una parte fondamentale del gioco;
- ***TestFight*** contiene i test di sconfitta, fuga, cambio pokémon, uso di oggetto ed evoluzione;
- ***TestWeakness*** contiene i test sulle debolezze dei tipi pokemon;
- ***TestPokemon*** e ***TestTrainer***, contengono i test riguardanti le istanziazioni delle relative classi.

Data la numerosa presenza di menu all'interno del gioco, che si sviluppa in una mappa abbastanza grande, sono poi stati effettuati anche numerosi test manuali che non potevano essere automatizzati.

- È stata testata tutta la mappa alla ricerca di eventuali bug grafici da fixare e si è controllato che la musica esistesse in ogni zona del gioco;
- si è poi testato il funzionamento di tutti gli allenatori e i capi palestra per controllare che funzionassero a dovere in caso di vittoria e sconfitta. I boss oltretutto dovevano consegnare la medaglia in caso di vincita per permettere all'utente di continuare il gioco;
- si è controllato che i pokémon leggendari presenti nel gioco scomparissero dalla mappa in caso di cattura o uccisione e che fossero ancora presenti in caso di sconfitta dell'utente.
- Si sono testati tutti i menù in modo che questi non permettessero all'utente di uscire illecitamente dalla pausa e che non presentassero bug di alcun tipo, soprattutto quando mancavano pokémon nel box, pokémon attivi in squadra o oggetti nello zaino.

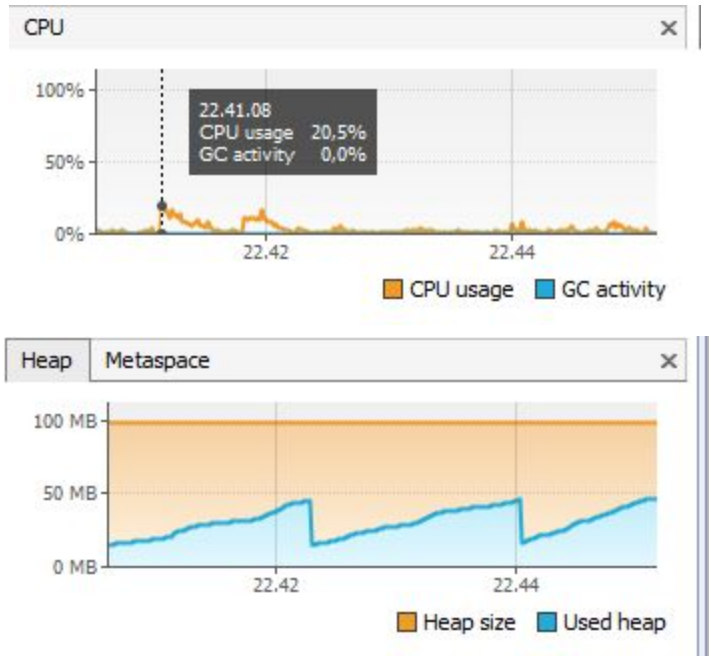
Infine si è controllato che il combattimento presentasse i giusti messaggi e pokémon in campo e che il gioco fosse interamente completabile al 100%.

Sono state inoltre analizzate le prestazioni dell'applicazione tramite il tool JVisualVM, lanciato da riga di comando sul sistema operativo Windows 10 Educational.
Il processore utilizzato è un Intel i5 6600K @ 3.5 GHz.
Si è registrato il picco nell'utilizzo della CPU durante la prima battaglia contro un allenatore.

Tale valore non ha comunque mai superato la soglia del 20%.

Il valore medio riscontratosi aggira intorno al 7-8%.

La memoria (Heap) utilizzata dall'applicazione ha mantenuto un valore medio di circa 40-50 MB, allocazione massima di circa 100MB.



L'applicazione infine è stata manualmente testata nei seguenti sistemi operativi: Windows 7 Professional, Windows 10 Home, Windows 10 Educational, Linux Ubuntu 15.04, Linux Ubuntu 16.04 (Macchina Virtuale), Mac OS X El Capitan.

3.2 Metodologia

Il gruppo ha impostato fin da subito il lavoro simulando di essere un team di sviluppo, ove ogni giorno vengono fatte discussioni riguardanti i progressi effettuati da ciascun membro.
Il team pertanto ha lavorato in un ambiente ove era possibile la comunicazione veloce con qualsiasi membro, così eventuali modifiche richieste sono state fatte al bisogno e la risposta è sempre stata ottima.
Vi è stata quindi una perenne discussione tra i componenti del gruppo, cosa che ha facilitato la realizzazione delle parti più complesse, grazie al supporto dei compagni.

Ogni componente del gruppo ha però realizzato il proprio lavoro in autonomia, richiedendo l'aiuto dei compagni solo nei casi di più difficile gestione.
L'interazione tra i componenti è stata comunque ridotta ad un semplice scambio di richieste/consigli, tranne in determinate situazioni, che rappresentano comunque una percentuale inferiore al 30% del lavoro complessivo di ognuno.

Christian Serra

Ho progettato tutto il package.model escluso la parte del fight in più ho sviluppato, partendo da uno scheletro, la parte fisica della mappa, ovvero la disposizione dei vari Tile e oggetti.
Ho inizialmente studiato la possibile struttura del dominio prima avvalendomi dei tradizionali carta e penna, poi iniziando le prime interfacce e implementazioni; dopo, con un approccio top down, ho continuato aggiungere e testare nuovi contenuti e funzionalità, facendo periodici check alla correttezza del codice scritto fino a quel momento.

È stata essenziale la comunicazione con Camporesi per riuscire a mettere all'interno del game tutta la logica creata nel model, inoltre lo scambio di informazioni tra me e Bondi si è rivelato particolarmente utile per gestire al meglio tutti i possibili aspetti del Fight.
Durante l'ultimo periodo del progetto mi sono occupato dell'ottimizzazione delle risorse utilizzate, del miglioramento della qualità del codice e anche di un perfezionamento dell'aspetto grafico della mappa.

Davide Bondi

Ho sviluppato il package model fight. Ho realizzato quindi tutta la parte logica che riguardava il meccanismo di battaglia. All'inizio vi è stato un confronto con Serra per stabilire una sintonia adeguata tra il combattimento ed il resto del model, inoltre sono state necessarie alcune precisazioni con Camporesi per ciò che riguardava l'interazione tra il model e il controller. Dopo aver completato il lavoro e i collegamenti tra model, view e controller, dai vari test automatizzati e manueali sono emersi vari bug i quali sono stati fixati man mano con il procedere del progetto.

Daniel Veronesi

Mi sono occupato di realizzare le classi e le interfacce presenti all'interno del package view.

Ho quindi lavorato sulla componente grafica del gioco e le musiche al suo interno. A inizio progetto c'è stato un confronto con Camporesi per parlare sull'uso delle interfacce da usare e stabilire i criteri di comunicazione fra la View e il Controller, in maniera tale da poter sviluppare le nostre rispettive parti di codice indipendentemente.

Quando il lavoro era già sviluppato a grandi linee, ho poi sentito le critiche costruttive del gruppo per avere conferme riguardo all'estetica delle finestre usate. In tal modo ho avuto un un parere più oggettivo ed ho reso la vista apprezzabile da un pubblico più vasto.

Michael Camporesi

Ho realizzato in autonomia interfacce e classi contenute nel package *controller*.

Mi sono occupato del salvataggio/caricamento delle informazioni necessarie, della gestione della musica, dell'installazione delle risorse necessarie e del movimento.
Il confronto con gli altri membri del team è stato costante.
Ho sempre cercato di comunicare in anticipo eventuali risorse di cui avevo bisogno, per permettere ai miei colleghi di avere tempo per rispondere adeguatamente alle richieste, ed ho sempre cercato di essere veloce nel rispondere alle richieste dei miei compagni.

3.3 Note

La realizzazione dello scheletro di un'applicazione funzionante si basa su un tutorial che spiega come realizzare un platform a scorrimento usando la LIBGDX.

Questo ci ha spiegato come impostare correttamente la libreria per il nostro progetto: https://www.youtube.com/watch?v=qik60F5I6J4&list=PLXY8okVWvwZ0qmqSBhOtqYRjzWtUCWylb

Lo scheletro della mappa, e l'iniziale tileset (l'insieme di tutte le immagini dei singoli tile), sono stati creati in maniera originale da un nostro compagno, Luca Raschi, che ringraziamo caldamente.

La mappa è stata realizzata utilizzando il potente editor grafico Tiled: http://www.mapeditor.org/, che ha permesso di attribuire ad ogni singolo Tile varie proprietà a livello di scrittura di file, cosa che si è rivelata fondamentale nella progettazione di una mappa facilmente estendibile, come spiegato in precendenza.

Ogni informazione riguardante i pokémon e le loro meccaniche di gioco è stata tratta dal sito http://bulbapedia.bulbagarden.net/wiki/Main_Page

Molti dubbi relativi all'uso di librerie o alla risoluzione di vari errori riscontrati durante la programmazione sono stati risolti grazie al sito StackOverflow: http://stackoverflow.com/

La stesura della maggior parte delle mosse presenti nell'enumerazione *Move* (si parla di un 85% mentre il 15% restante era stato fatto a mano inizialmente per avere pronte le prime mosse base funzionanti), sono state importate grazie a script in linguaggio Bash creati personalmente da Serra su Ubuntu 15.04, mentre per la stesura di tutti i 152 pokémon all'interno dell'enumerazione *Pokedex* si ringrazia questo sito (http://www.dragonflycave.com/list.aspx), che permette facilmente di poter stilare liste di pokémon con diversi possibili attributi.

Sempre in ***Pokedex*** per inizializzare staticamente l'intero moveset di ogni entry si è fatto uso, in una classe statica separata: ***InitializeMoves****,* della libreria Guava (https://github.com/google/guava) per costruire con facilità *mappe di livello mosse* non modificabili, che rendono il codice più facilmente leggibile e di facile ritocco, visto che specialmente la parte di mosse imparate da un pokémon è soggetta a continui cambiamenti persino nei giochi originali.

Le musiche di gioco sono state prese dalla seguente compilation: https://www.youtube.com/watch?v=dHFNllQQM28&index=2&list=PL22604C8CC8FCB417

Gli sfondi sono stati trovati in Google Immagini, e la proprietà di tali è da attribuirsi ai siti di appartenenza.

Inoltre, dopo aver già progettato inizialmente tutte le varie schermate e menù vari, si è deciso di sfruttare un più potente tool per la creazione di GUI: http://www.eclipse.org/windowbuilder/, come suggerito durante le lezioni.
Esso permette di avere più flessibilità e possibilità nella disposizione dei vari componenti all'interno delle schermate: si è reso particolarmente utile nella creazione del *FightScreen* dove immagini, scritte e barre si accostano.

4 Commenti Finali

4.1 Autovalutazione

Christian Serra

Mi ritengo particolarmente soddisfatto del lavoro svolto per la mia parte di model e mappa, in quanto credo di avere raggiunto una buona maturazione ed applicazione del linguaggio Java e di buone pratiche di programmazione ad oggetti. In particolare, ho lavorato fin dall'inizio pensando di dovere trattare questo progetto non come un mero elaborato da consegnare ai fini del voto bensì come una sfida, un'occasione per apprendere e migliorare ma anche come un lavoro per cui essere orgoglioso di condividerlo con amici e, perché no, anche con futuri datori di lavoro.

Naturalmente non ritengo che il mio operato sia perfetto, tutt'altro: credo che abbia ancora bisogno di un po' di "polishing"sia a livello di codice (ridurre ulteriormente dipendenze tra classi, aumentare il livello di incapsulazione, suddividere il ruolo di alcune "god class", adattare alcune classi con pattern di programmazione più avanzati…), che a livello di contenuti.
Per quanto riguarda quest'ultimi penso di aver fatto un ottimo lavoro in quanto, pensando a possibili sviluppi futuri, molte soluzioni sembrano facilmente implementabili con pochissimi ritocchi necessari al codice già esistente.
Facendo un esempio sulla mappa, su cui ho lavorato particolarmente per farla diventare facilmente importabile, aggiungere intere nuove zone (con all'interno i vari oggetti, insegne, NPC…) non richiede alcuna modifica sul codice sorgente e può essere fatto semplicemente con l'editor grafico Tiled.
Infine ho trovato molto stimolante lavorare ad un progetto di un gioco che ha segnato la mia infanzia e ciò mi ha spronato ad essere in continua ricerca di miglioramenti affinché il nostro prodotto finale fosse il più fedele possibile all'originale, mantenendo comunque una qualità di codice accettabile.

Davide Bondi

Sono piuttosto soddisfatto del lavoro eseguito.
Lavorare a questo progetto è stato decisamente utile poiché mi ha permesso di sfruttare e consolidare gli insegnamenti acquisiti durante il corso, ed è stato particolarmente istruttivo poiché mi ha fatto comprendere realmente cosa significa fare parte di un team di sviluppo.

E' stato molto stimolante realizzare un'applicazione di questo tipo, ho capito la complessità che sta dietro alla realizzazione di un videogioco, elaborarne le procedure giuste ed implementarle, aggiustare i bug e tutto ciò che riguarda esso; inoltre pokemon è uno dei giochi più famosi al mondo ed è stato appassionante realizzarne una propria versione.

Per quanto riguarda la qualità del codice credo di aver realizzato componenti discretamente incapsulati e facilmente estendibili.

Daniel Veronesi

Sono soddisfatto del lavoro compiuto sulla view e ritengo che l'impegno dedicato alla progettazione di questo gioco abbia portato a un ottimo risultato finale.

Data la gran quantità di menu e messaggi mostrati all'interno del programma, mi sono concentrato subito sulla realizzazione di un codice il più possibile privo di errori, cercando comunque di non penalizzare la sua flessibilità ed estensibilità.

Penso di aver raggiunto l'obiettivo che mi ero imposto, anche se magari il codice potrebbe essere reso ancora meno corposo.

Le dispense studiate a lezione si sono rivelate utili per muovere i primi passi e per avere delle fondamenta di base sull'argomento, ma, continuando a lavorare sul progetto, si è rivelato importante anche lo studio online per approfondire le conoscenze del settore.
Java 2D come motore di rendering si è rivelato una buona scelta per lo sviluppo del gioco e anche la CPU registra livelli molto bassi di utilizzo, grazie anche al lavoro dei miei colleghi.
Il progetto, comunque, ben si presta a futuri ampliamenti e migliorie. In particolare, in futuro, mi piacerebbe basare tutta la view sulla più moderna libreria LibGDX siccome, trattandosi di una libreria totalmente incentrata sui giochi, essa ha molti vantaggi rispetto alle altre.

Penso che l'esperienza fatta con questo lavoro sia stata istruttiva e gratificante. Nonostante infatti sia stato un progetto impegnativo sono stato molto contento di fare parte di questo gruppo perché si è arrivati a un punto importante della programmazione. Era dal primo anno che desideravo fare un esame come questo, che permettesse di vedere crescere il progetto mano a mano che si continuasse a lavorarvici su.

Michael Camporesi

Per quanto mi riguarda, sono molto soddisfatto del risultato finale del nostro lavoro.

Fin da subito mi sono reso conto che questo progetto sarebbe stato molto impegnativo, e mi sono posto l'obiettivo di riuscire a realizzare un'applicazione il più completa possibile.
Non ho infatti interpretato questo progetto come un progetto universitario, bensì come un vero e proprio gioco da realizzare.
Proprio per questo motivo sono ancora più felice del risultato, poiché il nostro gioco risulta completo, godibile graficamente, giocabile e, per mia opinione personale, divertente.

Dopo la fase di progettazione, dalla quale è emersa la decisione di usare la libreria LibGDX, mi sono immediatamente dedicato ad una fase di studio della suddetta libreria, unita al consolidamento delle basi di Java fornite dal corso.

In seguito, ho approfondito la conoscenza degli XML, formato in cui abbiamo deciso di salvare le informazioni che necessitano di persistere per poter riprendere una vecchia partita.
Ho iniziato quindi a fare test di salvataggio e caricamento di varie informazioni.

Fatto ciò ho iniziato ad imbastire i primi controllori, mentre cercavo di studiare per la nostra applicazione un sistema di movimento che fosse il più funzionale possibile, senza compromettere l'aspetto grafico.
Ho speso molto tempo nel modificare e migliorare tale sistema, grazie anche ai suggerimenti dei miei colleghi che, testando il gioco, scoprivano via via tutti gli errori.

Finita questa parte, mi sono dedicato totalmente al package controller, cercando di strutturarlo in modo che fosse il più chiaro ed estendibile possibile. Ho cercato inoltre di rispondere in maniera tempestiva con modifiche, quando queste si rendevano necessarie, in seguito a vari cambiamenti che occorrevano nel codice a seguito del lavoro dei miei compagni.

Infine ho cercato di migliorare progressivamente il codice, cercando di seguire al meglio possibile i criteri di buona programmazione indicati nel corso.

Purtroppo, per questioni di tempo, non mi è stato possibile migliorare ulteriormente il codice, cosa che sicuramente farò in seguito.

Nel complesso, comunque, ringrazio tutti gli insegnanti di questo corso per l'opportunità che mi è stata concessa, poiché reputo sia stata un'esperienza altamente formativa, che risulterà sicuramente utile nel mio futuro.
Reputo sia inoltre importante comunicare l'importanza che Bitbucket ha ricoperto, perché si è rivelato uno strumento potentissimo per condividere con i miei colleghi ogni singola modifica, cosa che sarebbe stata altrimenti difficile.

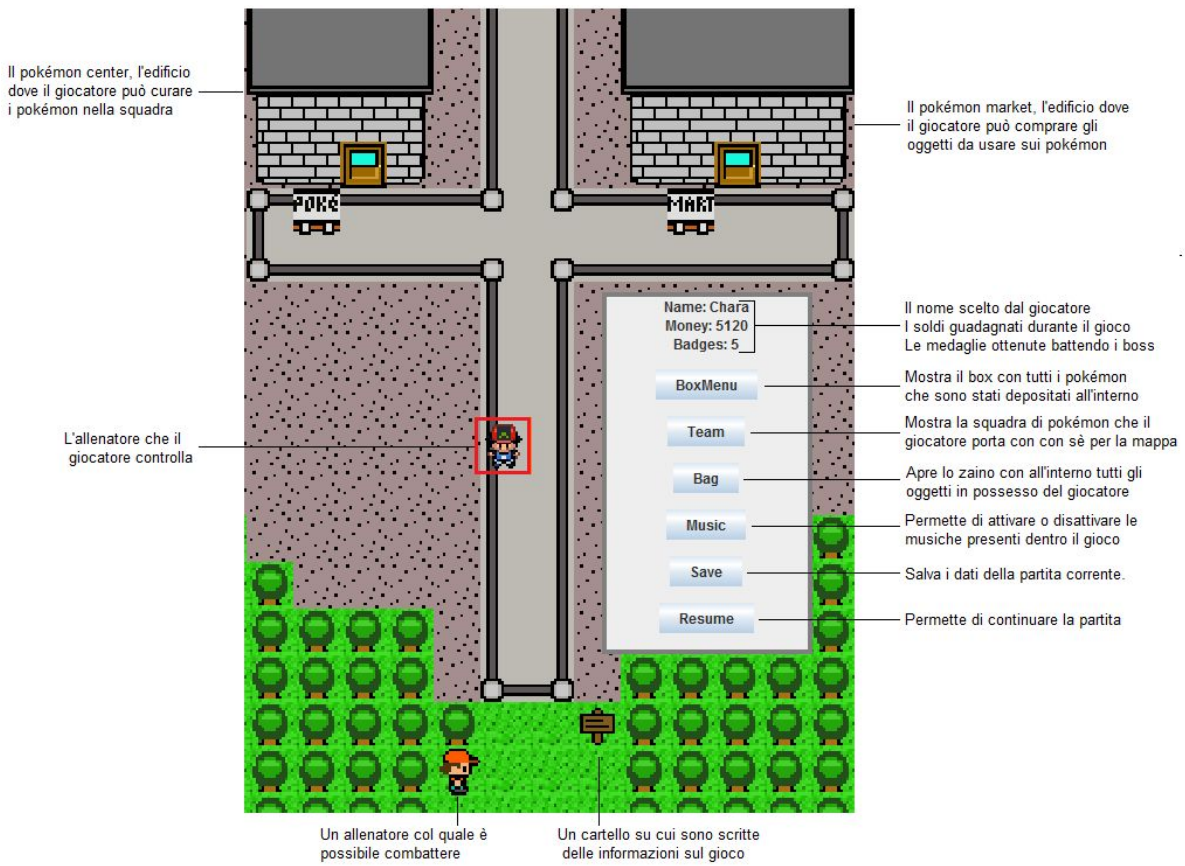
Spero che venga riconosciuto il nostro lavoro per il quale, almeno personalmente, ho speso molto tempo e dedicato molto impegno.

5 Guida Utente

5.1 Guida

Menù principale

Schermata di gioco



Durante il gioco è possibile muovere il personaggio tramite l'utilizzo delle frecce direzionali oppure con l'uso dei tasti [W], [A], [S], [D]. L'interazione con gli npc e i cartelli avviene tramite il pulsante [INVIO] mentre per aprire i menù di gioco è necessario premere [ESC]. Per navigare e interagire nelle finestre, invece, si usa il mouse.

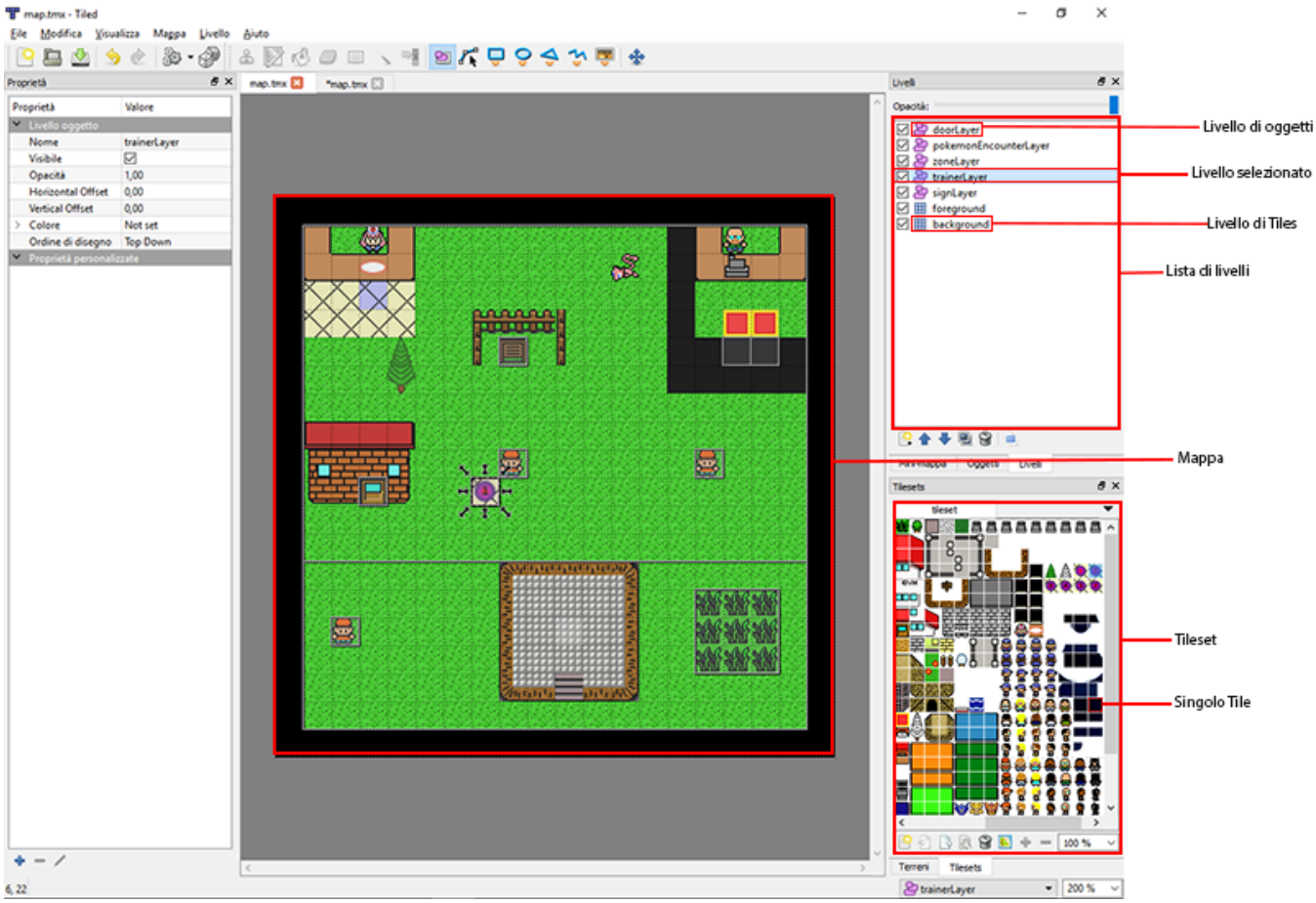
Schermata di battaglia

Lo scopo del gioco è quello di sconfiggere i capipalestra in modo da guadagnare medaglie per continuare ad esplorare la mappa. Durante questa avventura il giocatore deve allenare i propri pokémon facendoli combattere, in modo da creare una squadra forte abbastanza da affrontare nemici sempre più potenti.

Data l'enorme varietà di pokémon presenti nel gioco è consigliato tentare di catturarli tutti, per capire e sfruttare i punti forti di ogni specie.

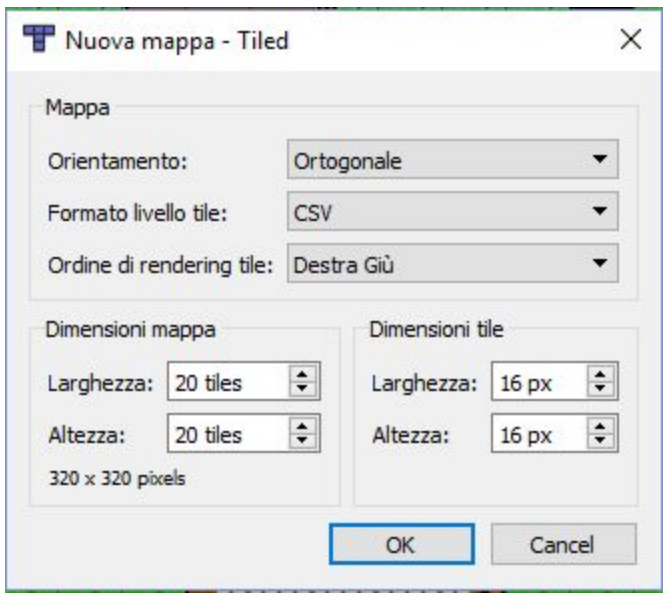
Infine, è possibile trovare dei pokémon leggendari in luoghi segreti. Questi saranno pokémon molto più forti del normale e metteranno a dura prova il giocatore se questo vorrà provare a catturarli.

5.2 Guida alla creazione della mappa



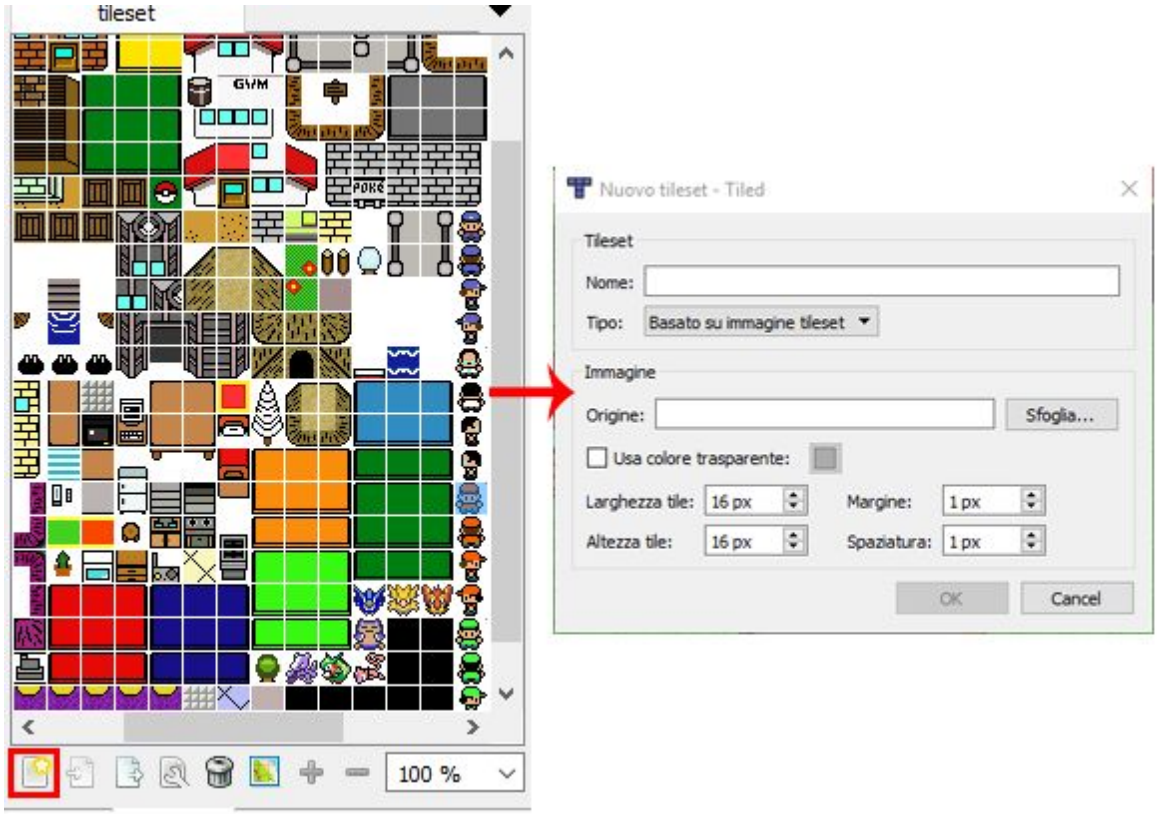
Overview di **Tiled** e dei suoi componenti base

Questa guida ha come proposito la spiegazione delle basi per creare da zero una mappa pienamente utilizzabile.
Prima di tutto occorre creare una nuova mappa: File->Nuovo ed apparirà questa schermata.



Orientamento, Formato livello tile e Ordine di rendering tile vanno settati come in figura mentre Dimensioni mappa e Dimensioni Tile sono a preferenza (tuttavia per utilizzare il Tileset già presente, come vedremo poi, è consigliabile settare le Dimensioni Tile a 16 pixel per entrambi).

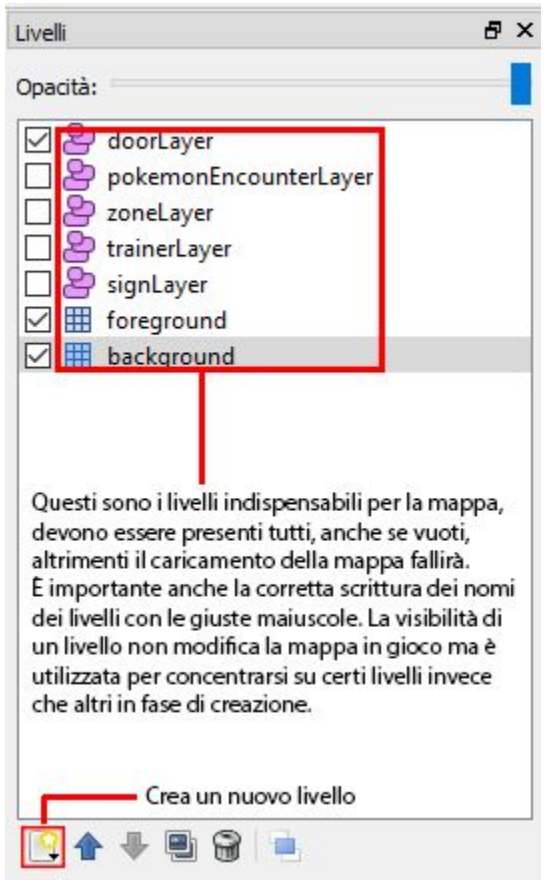
Ora serve aggiungere un *Tileset,* che praticamente è un file di immagine con all'interno tutti i vari Tile disposti a scacchiera.



Scegliere l'origine dell'immagine, attribuire un nome qualsiasi e impostare le dimensioni, margine e spaziatura a seconda di come si abbia suddiviso l'immagine.

Per usare il tileset già presente settare i valori come nell'immagine soprastante.

A questo punto occorre settare tutti i livelli (layers in inglese) correttamente, come spiegato nella figura sottostante.



- **doorLayer** conterrà tutti gli oggetti di tipo *TELEPORT* che dovranno avere come proprietà *DOOR_X* e *DOOR_Y* che rappresentano il punto d'arrivo del teleport. Per ottenere un *BADGE_TELEPORT* occorre semplicemente aggiungere un'altra proprietà chiamata "badgesRequired" che indicherà il numero di medaglie richieste per farlo funzionare.

  È importante che sotto ogni oggetto di questo tipo ci sia un Tile con la proprietà *tileType* settata su TELEPORT, altrimenti il teleport non funzionerà. Inoltre la posizione e la dimensione dell'oggetto devono combaciare perfettamente con il singolo Tile sottostante;

- **pokemonEncounterLayer** conterrà tutte le zone dove è possibile catturare pokémon selvatici. Ogni oggetto appartenente a questo livello dovrà per forza avere le seguenti proprietà: *avgLvl* (indica il livello medio dei pokémon trovabili nella zona), *pokemonList* (deve contenere i nomi in maiuscolo di tutti i pokémon trovabili in quella zona separati da uno spazio. NB: non è possibile inserire pokémon la cui rarità sia "Legendary", "Unfindable"o "Starter"), *zoneID* (l'identificatore univoco della zona).

  La zona è effettiva se il Player cammina su un Tile con tileType settato con *POKEMON_ENCOUNTER.*
  È altresì importante che le dimensioni e le posizioni di ogni zona combacino con l'insieme di Tile che deve racchiudere;

- **zoneLayer** conterrà tutte le zone dove è possibile ascoltare le canzoni di gioco. Ogni tipo di oggetto dovrà avere le seguenti proprietà: *music* (la canzone che si sente quando il giocatore entra nella zona, deve contenere anche l'estensione) e *zoneType* (essenzialmente il nome univoco della zona).

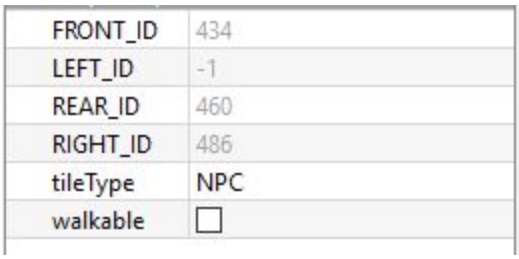
  Al momento si possono usare solo le canzoni già presenti perché aggiungere/rinominare una canzone richiederebbe la modifica parziale del codice, ciò verrà migliorato nei futuri sviluppi del gioco.

  La lista di canzoni include: home.mp3, opening.mp3, lab.mp3, wild.mp3, trainer.mp3, center.mp3, mart.mp3, cave.mp3, town.mp3, route.mp3. È tuttavia possibile sostituire una di queste canzoni con una nuova, a patto che quest'ultima abbia lo stesso nome di quella sostituita.
  Come per l'altro livello di zone è essenziale che anche qui il rettangolo della zona coincida al pixel con l'insieme di tile sottostanti;

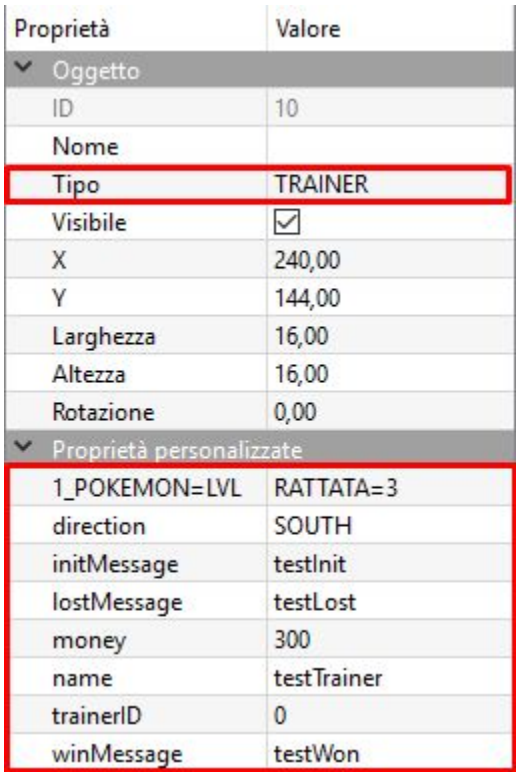
- **trainerLayer** conterrà tutti gli allenatori, capi palestra o NPC. Ogni singolo oggetto dovrà essere posizionato esattamente sopra un Tile con tileType settato su *NPC*.

  In particolare il Tile stesso (non l'oggetto presente in trainerLayer) dovrà essere proposto in 4 varianti (una per ogni direzione), ed ognuna di esse avrà le proprietà mostrate in figura sottostante. Le proprietà *_ID rappresentano i tileID delle varie direzioni del NPC, il valore dovrà essere -1 quando è esso stesso il Tile con quella direzione.

  Questo è necessario per poter cambiare la direzione all'NPC all'interno della mappa.



  Ritornando all'oggetto del **trainerLayer**, ecco le proprietà necessarie per creare un semplice Trainer:



Proprietà	Valore
Oggetto	
ID	10
Nome	
Tipo	TRAINER
Visibile	☑
X	240,00
Y	144,00
Larghezza	16,00
Altezza	16,00
Rotazione	0,00
Proprietà personalizzate	
1_POKEMON=LVL	RATTATA=3
direction	SOUTH
initMessage	testInit
lostMessage	testLost
money	300
name	testTrainer
trainerID	0
winMessage	testWon

I pokémon della squadra possono essere aggiunti, fino ad un massimo di 6, aggiungendo le proprietà *2_POKEMON=LVL*, ..., *6_POKEMON=LVL* specificando in ognuna di esse il nome del pokémon in maiuscolo seguito da un "=" ed infine il livello desiderato del pokémon. Gli altri campi sono abbastanza descrittivi.
Se invece si desidera creare un GymLeader, essenzialmente bisogna aggiungere agli stessi campi la proprietà *badgeID*, che rappresenta il numero del badge che si ottiene sconfiggendolo, e settare il *Tipo* con *GYM_LEADER*.
Invece per creare un semplice NPC semplicemente basta settare il *Tipo* con *NPC* e avere come uniche proprietà personalizzate *message* e *name*;

- **signLayer** conterrà tutti le insegne varie del gioco, il cui unico scopo è quello di mostrare un messaggio a video. L'oggetto che dovrà stare esattamente sopra ad un Tile con *tileType* settato a *SIGN,* inoltre, dovrà avere un'unica proprietà personalizzata, chiamata *signMessage*, il cui contenuto è semplicemente il messaggio da mostrare;

- **background** conterrà tutti i Tile che fanno da terreno, e che devono stare come base per i possibili Tile superiori del foreground che hanno il bordo trasparente (Sono inclusi i TELEPORT). In generale può contenere tutti i Tile che hanno uno sfondo non trasparente ma bisognerebbe spostare quelli camminabili nel foreground.
Inoltre ogni mappa *deve* per forza avere uno ed un solo Tile di tipo *DEFEAT* e uno di tipo *START*, che indicano rispettivamente il punto di spawn dopo aver perso una battaglia e il punto di inizio senza salvataggio. Entrambi devono essere necessariamente *walkable;*

- **foreground** conterrà tutti i Tile che devono stare in primo piano, in particolare strutture, NPC, ostacoli vari e Tile non walkable.
  Se un Tile walkable si trova nel foreground, il Player che ci camminerà sopra apparirà dietro il Tile. Questo effetto è utile per esempio per far passare il Player dietro alberi alti e per dare un senso di profondità al territorio. Inoltre, il foreground dovrebbe contenere i Tile di tipo *MARKET* e *CENTER* con cui il Player può interagire, e rispettivamente apre il menu del PokeMarket e cura tutti i suoi pokémon.

Per altre informazioni utili dare un'occhiata a come è strutturata la mappa principale nella cartella di installazione (/home/.pokejava dove home è la vostra home directory). E' interamente modificabile, e in caso non si sia fatto un backup della mappa originale basterà cancellare la cartella di installazione e riavviare il gioco, che riformerà correttamente tutti i file necessari.