Relazione BrosAdventure 2D

ANALISI

Il progetto commissionato dal professore Viroli Mirko mira allo sviluppo di un Platform 2D.
Un Platform è un tipo di videogioco dove la meccanica di gioco implica l'attraversamento di livelli costituiti da piattaforme disposte su più piani.
Il termine 2D (2 dimension) indica che il videogioco sarà strutturato su due dimensioni.

Il software esporrà le seguenti funzionalità:

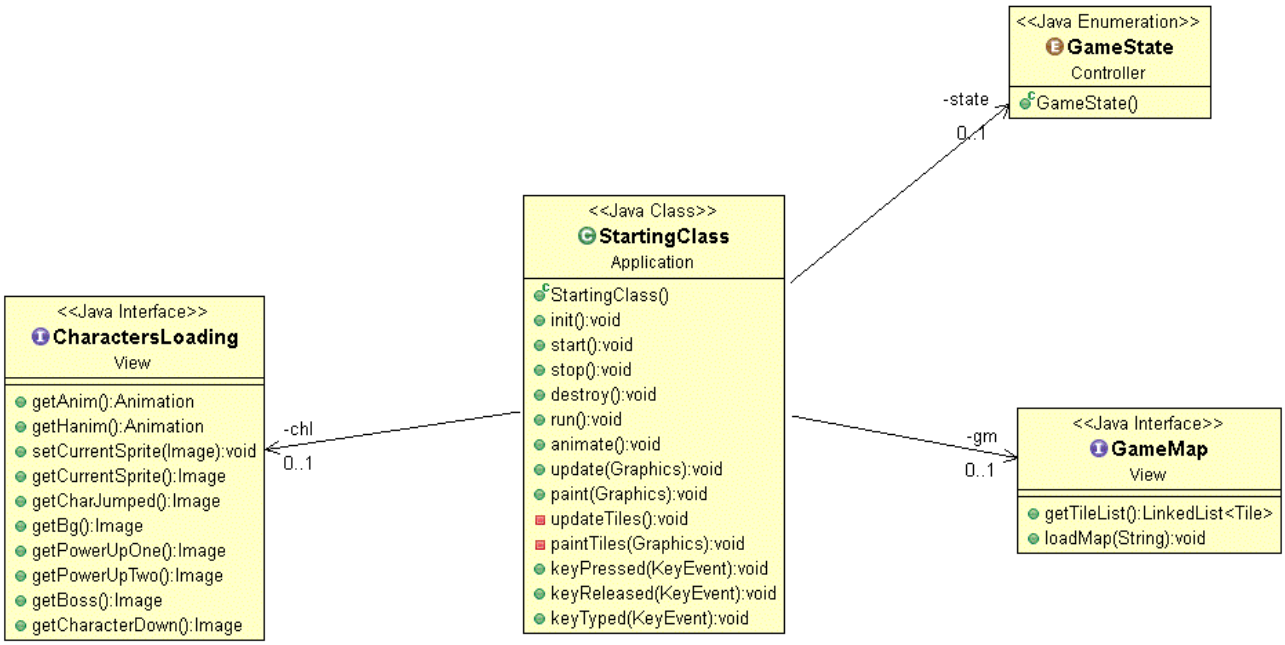
- Navigazione di livelli in uno spazio 2D.
- Presenza di un tutorial iniziale per illustrare i comandi di gioco.
- Nemici affrontabili con l'utilizzo di armi.
- Punteggio in base ai nemici sconfitti.
- Possibilità di game over restando senza vite disponibili.

Feature opzionali:

- PowerUp (Miglioramenti temporanei delle armi).
- Mappa con elementi dinamici.
- Selezione di un personaggio.
- Classifica con punteggi.
- Musica di sottofondo.

MODELLO

Il gioco BrosAdventure2D fornirà all'utente la possibilità di controllare un personaggio che potrà muoversi attraverso una mappa.
Il personaggio avrà la possibilità di combattere contro dei nemici con l'ausilio di un'arma, potenziabile lungo il livello ottenendo dei potenziamenti detti Power Up.
Il gioco termina nel momenti in cui il personaggio finisce le vite disponibile oppure sconfigge il nemico finale detto Boss.
Servirà quindi gestire gli elementi di gioco, le relative immagini e una mappa di gioco.



Starting class si occupa di gestire gli elementi del gioco, characters loading e game map si occupano di caricare rispettivamente le immagini relative agli elementi e alla mappa di gioco.

DESIGN

Il design architetturale utilizzato è quello MVC (Model, View, Controller).
Questo tipo di design è contraddistinto dal fatto che le 3 parti lavorano su elementi diversi e indipendenti.
Il Model si occupa di definire i vari elementi del gioco e i loro comportamenti, la View gestisce la parte grafica e il Controller gestisce gli elementi di gioco e il loro update a tempo di esecuzione.

DIVISIONE IN PACKAGE

Model:

- Contiene le interfacce e le classi appartenenti al model.

View:

- Contiene le interfacce e classi appartenenti alla view.

Controller:

- Contiene le interfacce e le classi appartenenti al controller.

Data:

- Contiene tutte le risorse come immagini e suoni utilizzate nel progetto.

Sound:

- Contiene le interfacce e classi che si occupano al caricamento di clip audio.

Ranking:

- Contiene le classi per la gestione del ranking.

Testing:

- Contiene le classi utilizzate per eseguire il testing.

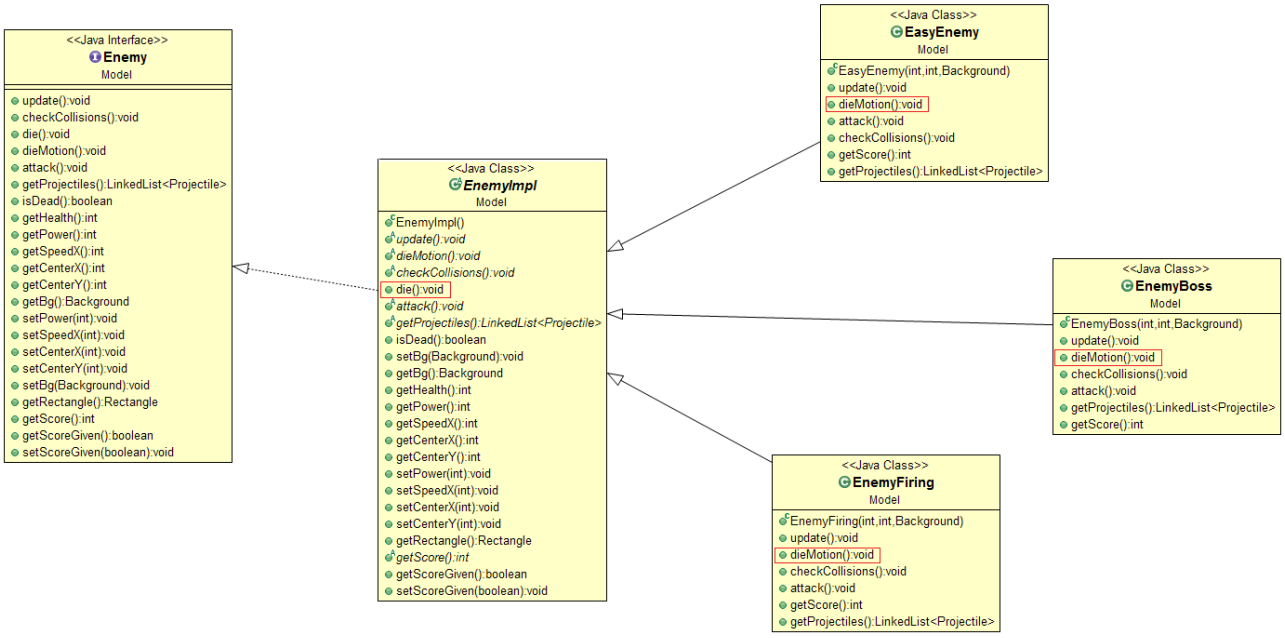
MODEL

L'idea di base nella progettazione del model è stata quella di definire delle macro classi da cui possono poi essere definiti oggetti diversi.
Per fare questo sono state utilizzate classi astratte che definiscono un standard da cui poi è possibile sviluppare vari elementi.

Esempio:
Per definire un Enemy, sono stati raccolti gli elementi base di un nemico all'interno della classe astratta EnemyImpl, da cui sono state poi estese le classi EasyEnemy, EnemyFiring, EnemyBoss, che definiscono tipi di nemici con le caratteristiche di un EnemyImpl, più caratteristiche proprie di ogni classe.

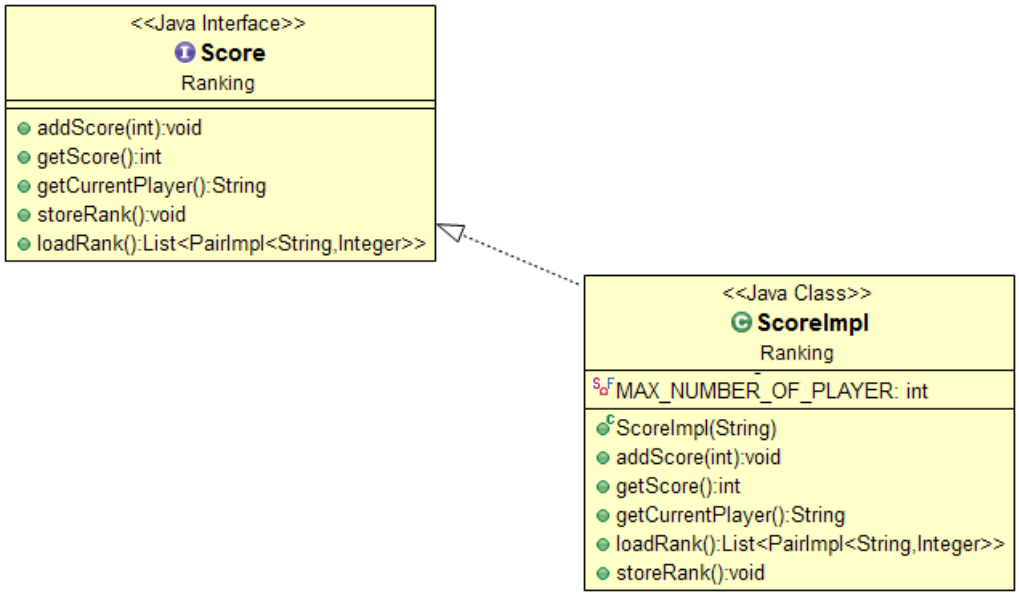
In questo modo sarà più semplice aggiungere in futuro elementi nuovi al progetto estendendo dalle classi astratte presenti.

Pattern:
Template Method: La classe astratta EnemyImpl utilizza un template method definendo lo scheletro del metodo die(), ma lasciando l'implementazione del metodo dieMotion() alle classi che estenderanno da EnemyImpl.
In questo modo i nemici seguiranno tutti die(), ma potranno avere dieMotion() differenti.



RANKING

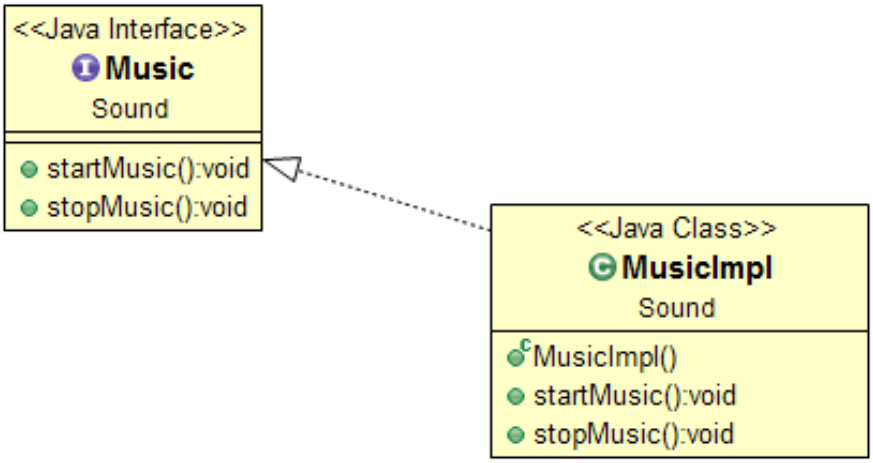
La classe relativa al ranking si occupa di fare store e load su file di una classifica ordinata in modo decrescente di un numero MAX_NUMBER_OF_PLAYER.
Questo è stato reso possibile attraverso l'utilizzo di un ObjectInputStream e un ObjectOutputStream.



Strategy: La classe ScoreImpl incapsula la strategia di ordinamento degli score in un comparatore che ordina i Pair<String, Integer> (che rappresentano i player in classifica: Nome, Score) confrontando gli score.
L'ordine ottenuto sarà decrescente.

```
//Sort the list with the comparator based on the score in decreasing order.
Collections.sort(this.listScore,new Comparator<PairImpl<String, Integer>>(){
    public int compare(PairImpl<String, Integer> p1,
            PairImpl<String, Integer> p2) {
        return (int) (p2.getY() - p1.getY());
    }
});
```

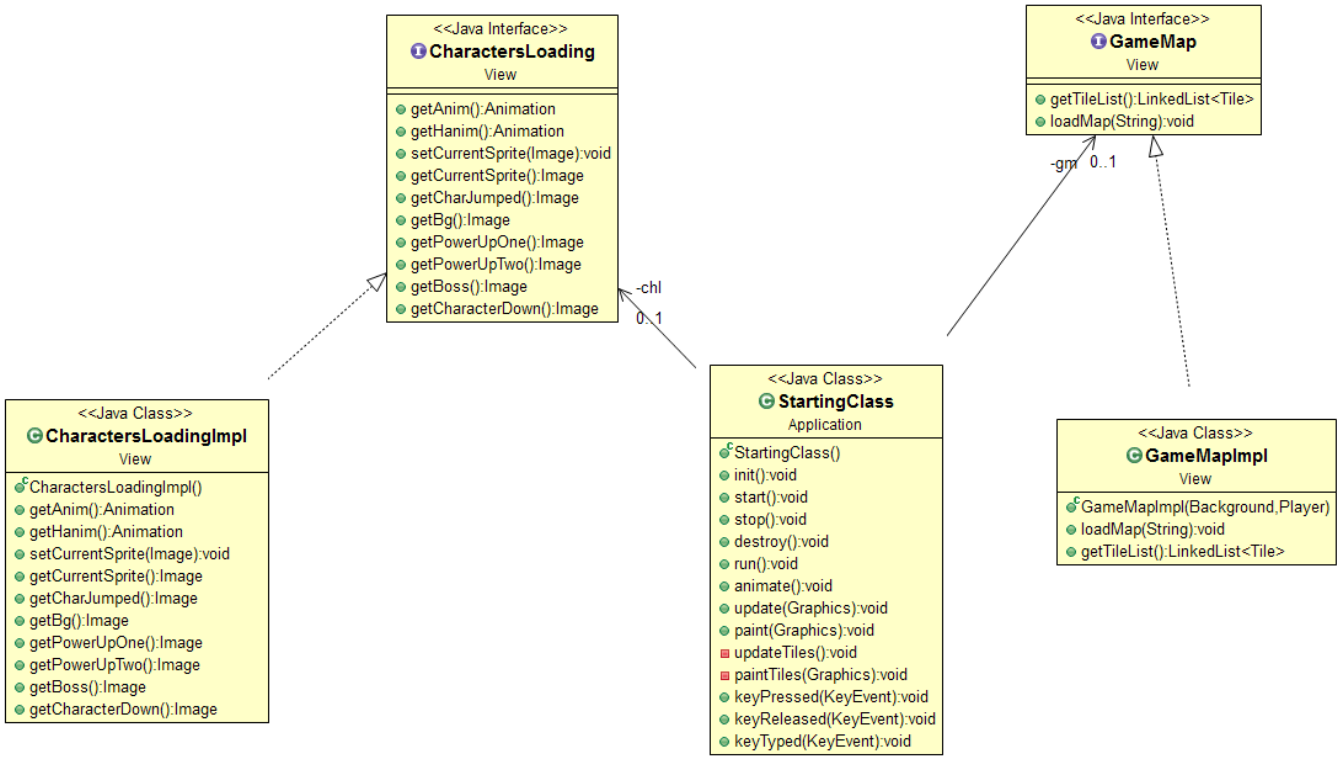
SOUND



Sound è costituito da una classe MusicImpl che implementa l'interfaccia Music.
Questa classe da la possibilità di caricare una clip audio attraverso un audio input stream, farla partire attraverso il metodo startMusic(), e fermarla con stopMusic().

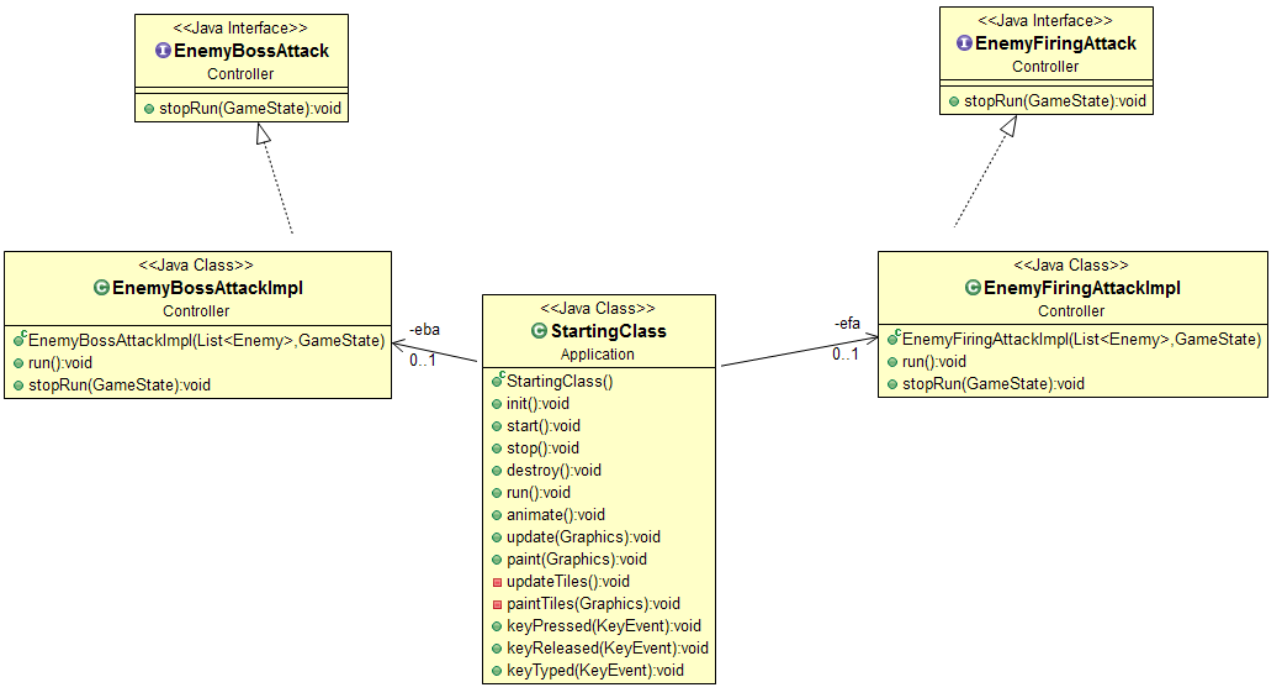
VIEW

La view si occupa di caricare le immagini dei vari elementi del gioco.
La classe GameMapImpl carica la mappa di gioca da un file txt associando ad ogni numero preso dal file un immagine di una “mattonella” detta Tile.
La classe CharactersLoadingImpl carica le immagini degli elementi di gioco e imposta le animazioni del player e dei nemici.
Starting Class si occupa di fare l’update degli elementi grafici.



CONTROLLER

Il controller si occupa di gestire l'update di tutti gli elementi del model e della view costantemente, utilizzando classi che estendono da Thread.
L'update è eseguito da StartingClass finchè lo stato di gioco è running.
Gli stati possibili di gioco sono definiti da un Enum class, GameState.



TESTING

I testing sono stati effettuati manualmente da utente.
Per effettuare un test delle classi del model sono state utilizzate principalmente due classi: StartingClassTesting, StartingClassCollisions.
StartingClassTesting avvia direttamente il gioco così da renderne disponibile l'immediato testing da parte dell'utente.
StartingClassCollisions è una classe simile a StartingClassTesting ma che utilizza una mappa creata apposta per testare le collisioni con i vari elementi della mappa.

METODOLOGIA DI LAVORO

Il lavoro è stato diviso in tre parti separate: Model, View Controller.
Rispettivamente Maicol Forti | Model, Luca Benini | View, e Luca Spinosi | Controller.
Per rendere equa la divisione dei lavori abbiamo deciso di attribuire allo sviluppatore del Controller anche la parte di Testing, la parte di Ranking e di Sound.
L'implementazione delle parti è stata svolta parallelamente.
Questo è stato possibile scegliendo in partenza l'architettura e definendo le linee generali delle interfacce che avremmo implementato.
Per poter lavorare sullo stesso progetto abbiamo creato un repository mercurial centrale su Bitbucket, su cui attraverso operazioni di pull e push abbiamo integrato un po' alla volta le varie parti del progetto.

NOTE DI SVILUPPO

L'idea del progetto è stata sviluppato sulle basi di un tutorial seguito da Luca Spinosi chiamato KiloBolt.
Il tutorial è trovabile a questo link: http://www.kilobolt.com/
Le parti di codice utilizzato sono state adeguate agli standard richiesti.

COMMENTI FINALI

Forti Maicol :

- Il lavoro non rispetta come avrebbe dovuto la divisione dei ruoli e la separazione tra controller e view non è ben definita.
- All'interno del gruppo ho cercato di essere sempre disponibile verso i miei compagni, per esempio spiegando loro il funzionamento di bitbucket in relazione a mercurial.