Relazione per Progetto Delirium

Giovanelli Joseph, Magnani Matteo, Pisano Giuseppe

28 Febbraio 2016

Indice:

Capitolo 1
Analisi

1.1 Requisiti

L'applicazione consiste in un videogame 2D a scorrimento orizzontale.
Esso sarà diviso in tre livelli da completare in sequenza senza morire, come tipico del genere Roguelike.
-I livelli sono semplici arene, che seguono lo stile Platform, in cui ci si può muovere liberamente a destra e a sinistra, saltare e cadere.
-Saranno presenti elementi statici, dinamici, con danno a contatto o meno e che eventualmente potranno sparare.
-Ogni livello vedrà come protagonista un eroe differente.
-Nel menù sarà possibile gestire il volume dell'audio e la difficoltà. Durante il gioco sarà possibile mettere in pausa e ritornare al menù principale.

1.2 Analisi e modello del dominio

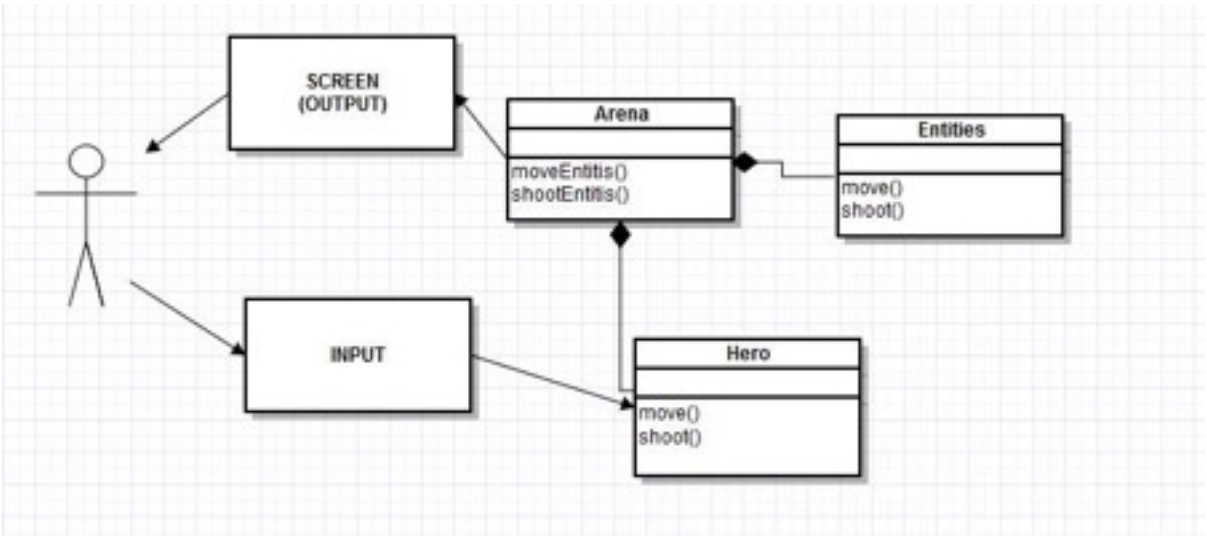
Delirium dovrà essere in grado di gestire più arene nelle quali le entità all'interno potranno muoversi di loro spontanea volontà o a causa di terzi.
Questo fatto comporterà più fattori:

-gestire movimenti mostri;
-gestire movimento dell'eroe;
-gestire numerose collisioni;
-gestire la fine di un livello.

Le collisioni potranno portare a un calo della vita delle entità in questione e la loro possibile morte.
Sarà inoltre possibile per le entità sparare e quindi creare proiettili che al momento della collisione infliggeranno danno.

Il personaggio sarà gestito in maniera autonoma dato che riceverà input dall'utente mentre le altre entità avranno un loro pattern di movimento e di sparo.

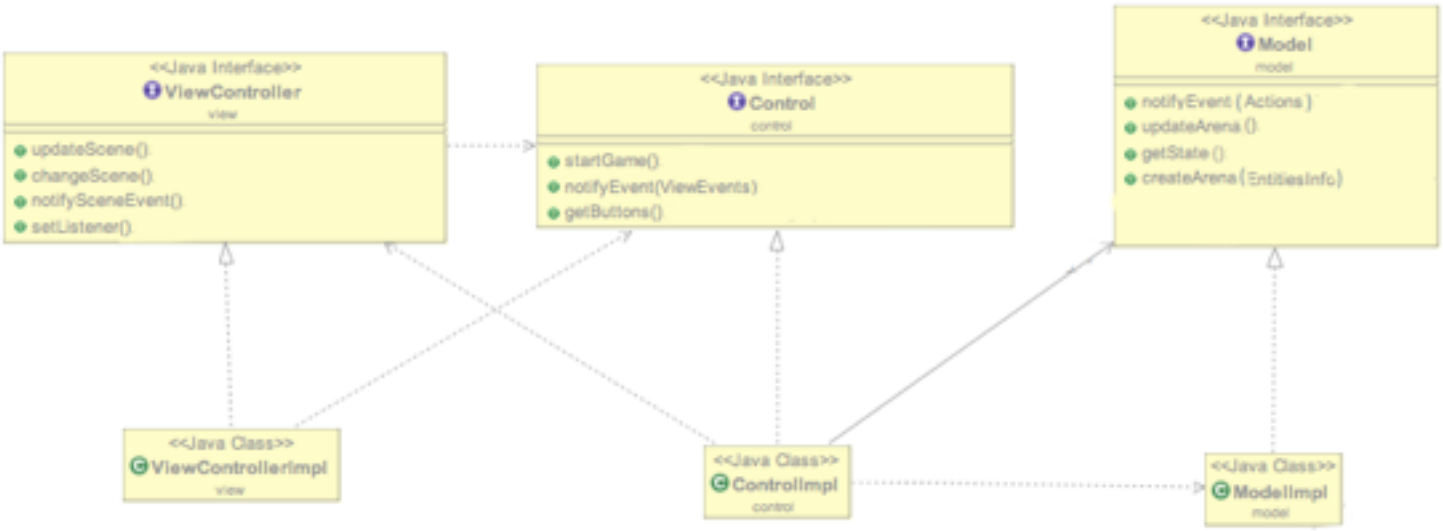
Tutto ciò dovrà essere renderizzato a schermo in maniera real-time.



Capitolo 2
Design

2.1 Architettura

Il pattern progettuale utilizzato per realizzazione del nostro software è l'MVC (Model View Control).



Il control contiene un'istanza sia del model che della view.
Questo utilizza l'istanza del model per:

- effettuare interrogazioni;
- provocare cambiamenti all'interno del mondo e quindi scandire il tempo.

La view potrà notificare gli input dell'utente attraverso il pattern oberserver oppure quando necessario chiedere al control delle informazioni aggiuntive volte a portare a termine un determinato compito.

Ad esempio potrà richiedere al control, in caso di creazione di un menù, la struttura di quest'ultimo.

Il control, invece, informerà periodicamente la view dell'attuale stato del mondo di gioco, oppure di particolari eventi quali la pausa o il cambio di schermata.
Questi possono essere scatenati sia da input utente che da particolari stati del'Arena.

L'entry Point della nostra applicazione è una classe launcher che istanza control e view mettendoli in comunicazione. Il model viene richiesto direttamente dal Controller in quanto è in Singleton.

L'architettura da noi creata è stata strutturata in modo tale da permettere un cambio indolore di tutta la parte della View. Infatti quest'ultima non contiene nessun riferimento alla logica o alla formattazione dei vari menù e del gioco che sono gestiti rispettivamente dal Control e dal Model.

Le uniche modifiche che al massimo si renderebbero necessarie sarebbero ad alcuni metodi di traduzioni da parte del Control, che adattano le entità presenti nella logica del Model a quelle disponibili nella View per la rappresentazione.

2.2.1 Design Dettagliato Model

Per quanto riguarda la parte di Model, essendo il mondo di gioco molto complesso si è deciso di adottare un approccio “dividi et impera” e quindi dividere il problema in macro problemi da risolvere, e questi in altri problemi più piccoli, e così via.
Questa suddivisione, infatti, si riflette ed è visibile nella suddivisione del package model.

Prima di tutto è stata stabilita la comunicazione con il controller.
Essa avviene tramite:
- Un’interfaccia chiamata Model che stabilisce come interagire con il mondo creato e quali strutture dati utilizzare per interagire;
- La sua relativa implementazione, che sfrutta il pattern Singleton dato che deve essere presente uno, e uno soltanto, ModelImpl in qualunque istante dell’esecuzione.

All’interno del Model dovranno sicuramente essere mantenuti e gestiti i dati di ogni singola entità del mondo. Per comodità si sono quindi create un’Arena e un’ArenaManager.
- La prima si occupa principalmente di aggiornare le entità in base alle richieste. In seguito si è deciso di affidare a questa anche il compito di riconoscere le varie entità del gioco al momento della creazione e posizionarle nella struttura dati opportuna per far si che

vengano gestiste. Per questa esigenza il Pattern Visitor è stato ottimale.
- Le seconda, ArenaManager, si occupa della gestione collisioni (vedi Magnani Part).

Qui sopra viene illustrato il diagramma UML del Pattern Visitor.
Di seguito porto i link delle risorse web che ho sfruttato dato che non è stato usato durante le lezioni.
http://stackoverflow.com/questions/29458676/how-to-avoid-instanceof-when-implementing-factory-design-pattern

https://en.wikipedia.org/wiki/Visitor_pattern

Una volta creata l'architettura che avrebbe contenuto le entità, la prima cosa a cui pensare era strutturare ogni entità in modo tale che sarebbe stata il più estendibile possibile.
Per come il gioco è stato ideato avremmo potuto avere in ogni arena, entità statiche che avrebbero potuto sparare (Torrette) e non (Casse), entità dinamiche che avrebbero potuto sparare (Mostri) e non (Piattaforme), le stesse avrebbero potuto essere viventi (Mostri) oppure no (Piattaforme, Casse ecc..). Per ovviare a questo problema si è deciso di creare un'entità componendola di diversi moduli, tramite Composizione, poi lo stato di ogni entità (avere o non avere certi moduli come quello di movimento, di sparo, di vita ecc..) avrebbe determinato la sua vera natura.
Approcciando questa idea l'ereditarietà e la suddivisione in Classi Astratte, Interfacce e applicazioni di metodi come lo Strategy e l'applicazione di Functional Interface si sono spostate all'interno di ogni "modulo".
Scendendo nel dettaglio, ogni entità avrà un codice univoco che lo identificherà, una posizione che lo collocherà in un determinato spazio dell'arena e un gestore del suo danno che deciderà se applicarlo in funzione se sia con o senza vita.
Elementi opzionali che invece potrà avere ogni entità saranno:

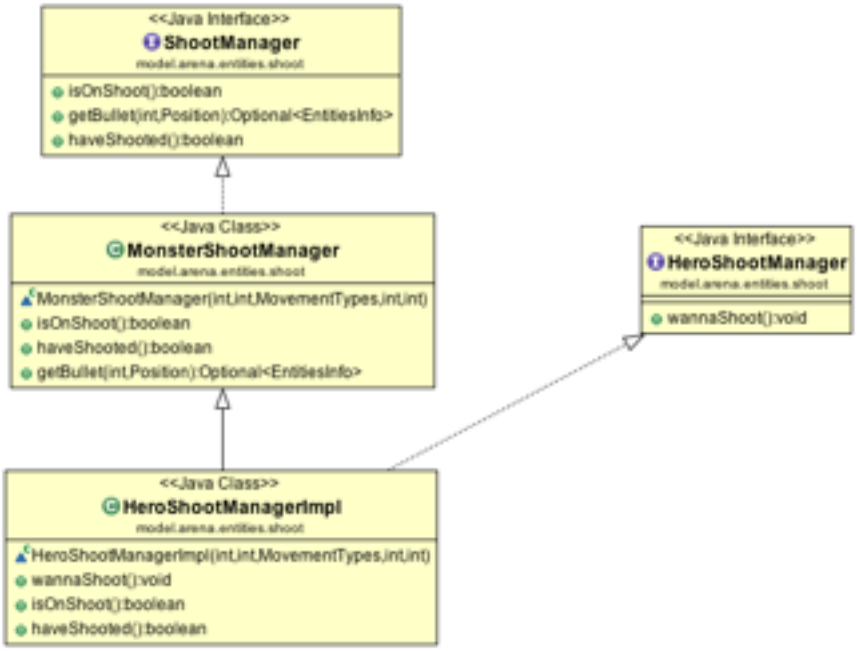
Gestore del movimento;

Gestore dello sparo;

Gestore del danno che lui potrà dare.

Il gestore del movimento è basato su un'interfaccia MovementManager la quale può essere contenuta in ogni entità tramite un Optional. Il fattore che differenza le varie implementazioni è l'override del metodo getNextMove che contiene la logica del movimento e restituisce la prossima posizione dell'entità.
La prima implementazione di questa interfaccia è un classe astratta dato che si è notato il numeroso codice in comune determinato da campi e metodi getter e setter.
Da esso si diramano le implementazioni figlie come la ReactiveMovementManager che determina il movimento di un oggetto solo se spostato da terzi, e quindi viene applicata solo la gravità. Successivamente incontriamo la LinearProActiveMovementManager che invece stabilisce un movimento lineare orizzontale o verticale di qualsiasi entità, piattaforme, mostri ecc.. Da questa eredita la

RandomProactiveMovementManager che utilizzerà il metodo della superclasse in modo tale da renderlo pseudo-randomico.



Lo Strategy dello Shoot Manager invece è molto più semplice, l'interfaccia viene implementata dallo Strategy del Monster che viene ereditata dalla classe HeroShootManagerImpl che implementa anche un'altra interfaccia.
Questo perché l'eroe deve scegliere se sparare o meno, mentre i mostri hanno uno sparo periodico.

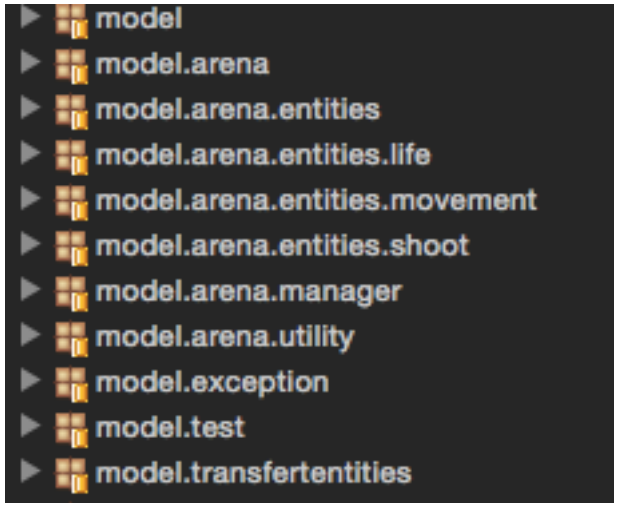
Altri Pattern usati sono :

-il Builder con cui vengono generate tutte le entità con vari controlli opportuni e lanciate le eccezioni.

-la Factory: alle entità vengono assegnati gli opportuni Manager di Shoot e Movement con delle Factory.

è stata creata, inoltre, una Functional Interface per creare delle lambda all'interno delle enumerazioni delle azioni.
In questo modo basterà applicare la funzione all'azione assegnata all'entità.

La gestione del package, come detto prima e riportato nell'immagine sottostante rispecchia la scaletta della spiegazione fatta fin'ora.
Questa divisione è stata effettuata per rendere il codice più navigabile e accessibile.
Le macro sezioni sono basate sulle funzionalità e i servizi che offrono e le relazioni che le classi all'interno hanno tra di loro.
Il model conterrà un'arena che sarà gestita dall'arenaManager e conterrà le entità.
Anch'esse saranno costituite dai vari moduli di movimento e sparo.
Al''interno dell'arena sono necessarie alcune classi di utility. Sul model sono state sviluppate le relative eccezioni e sviluppati i test.
Nel transfertEntities sono presenti le strutture di comunicazioni con il controller.



2.2.2 Design Dettagliato View

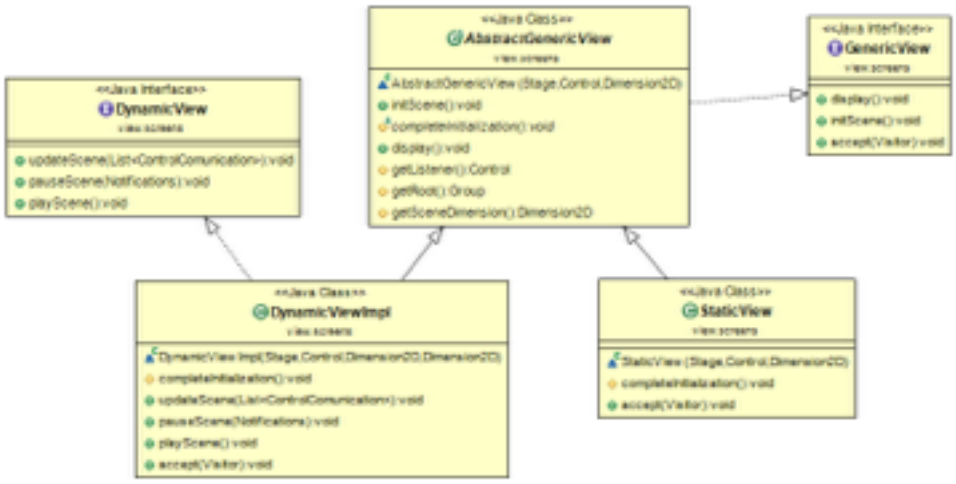
La View espone al controller una singola classe per l'interfacciamento, in modo che un aggiornamento nell'architettura interna progettata non comporti delle modifiche anche nel resto dei moduli dell'applicazione.

Le funzionalità offerte da questa interfaccia sono:

- La creazione e la visualizzazione di una nuova schermata, la cui tipologia è decretata da un elemento di un enumerazione che definisce i tipi di scena disegnabili da questo modulo;
- L'aggiornamento della schermata creata, se questa supporta l'aggiornamento;
- La notifica alla schermata creata di un evento particolare, quali per esempio lo stato di vittoria o di sconfitta, definiti grazie ad un enumerazione;
- La definizione di un oggetto fornito dal controller a cui questo modulo potrà fare eventuali richieste di informazioni o tramite cui inviare le propie.

La prima difficoltà sorta nella realizzazione di queste funzionalità è stato la creazione di un meccanismo di cambio della schermata che

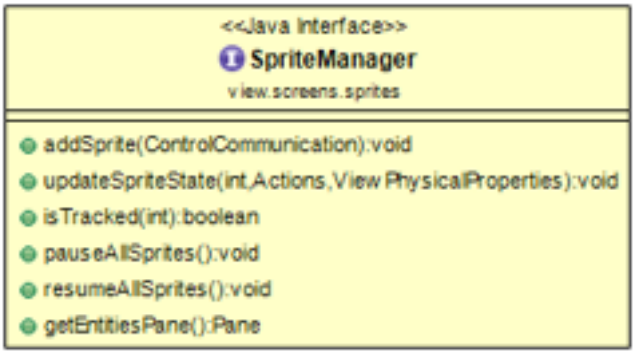
permettesse la creazione di scene con caratteristiche e proprietà anche molto diverse tra loro, come quelle che possono avere un menù statico e una schermata di gioco.
Perché potessero essere gestite in modo simile dal ViewController le diverse scene sono state sviluppate a partire da una stessa interfaccia generica, la cui implementazione si va a specializzare grazie all'uso del Template Method pattern applicato al momento della loro inizializzazione.



Questo ha permesso di utilizzare al momento della creazione delle scene il Factory pattern, garantendo così un maggior grado di pulizia nel codice e la libertà futura di aggiungere altre tipologie di scene senza modificare il funzionamento dell'architettura creata.
Le schermate aggiornabili però, necessitando di metodi aggiuntivi per la loro gestione, vanno a ereditare da una seconda interfaccia che ne definisce le ulteriori funzionalità. Per utilizzare questi metodi si sarebbe reso necessario un cast dall' interfaccia generica a quella più specifica. Per ovviare a questo problema ottenendo un riconoscimento del tipo dinamico delle scene istanziate dalla factory, e quindi avere un comportamento diverso in base a questo, ci si è avvalsi del pattern Visitor.
Un altro metodo molto più semplice per ovviare al problema sarebbe

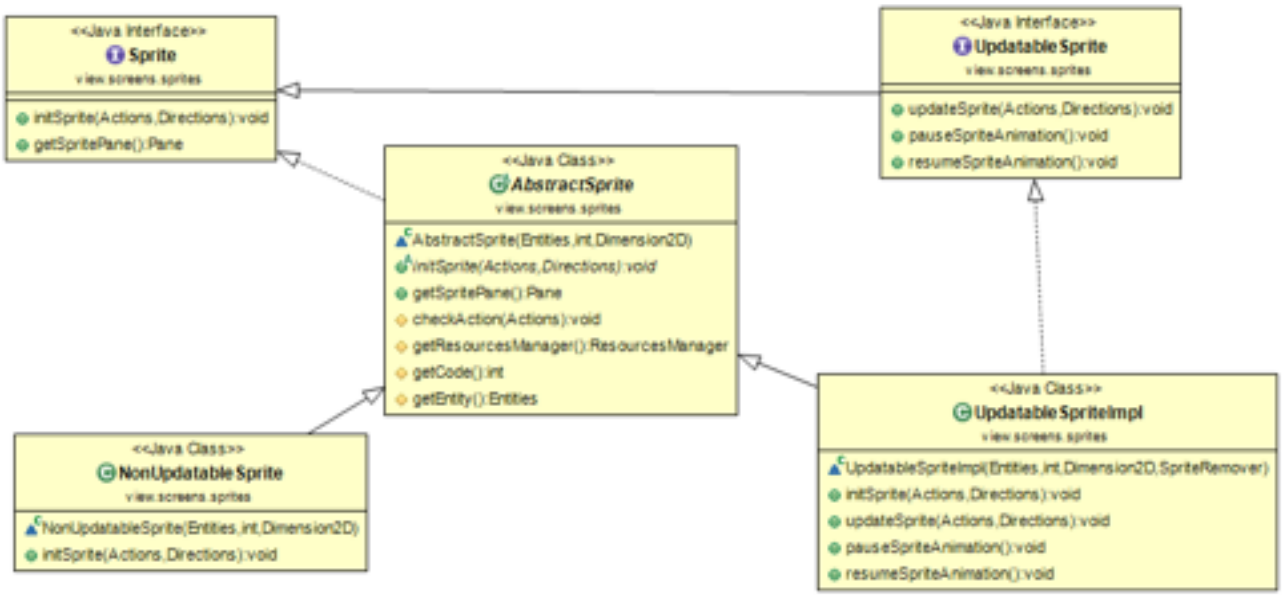
stato quello di eliminare completamente la factory e istanziare direttamente le scene con l'interfaccia più opportuna, considerato comunque che allo stato attuale ne sono presenti solamente due tipologie. La decisione di avvalersi comunque di questi pattern, e quindi articolare ulteriormente un'architettura che sarebbe stata molto più semplice, è stata presa pensando a una possibile, e molto probabile, espansione futura, quando i tipi di schermata gestiti da questo modulo saranno molti di più. Un altro possibile tipo di schermata per esempio sarebbe potuta essere una per la visualizzazione di filmati, utilizzabili come intermezzo fra i vari livelli. La metodologia adottata invece permette di aggiungere qualsiasi tipo di schermata, sempre sotto l'interfaccia GenericView, senza andare a modificare la ViewControllerImpl che le gestisce.

Definita questa parte, la sezione di questo modulo che ha richiesto una maggiore complessità architetturale è stata quella che gestisce la rappresentazione del mondo di gioco e quindi l'aggiornamento delle entità che lo compongono e delle relative animazioni.
La movimentazione e la gestione delle entità del gioco è delegata dalla schermata di tipo dinamico a una seconda classe, lo SpriteManager.



Questa classe possiede al suo interno una lista degli sprite da visualizzare a schermo, a cui è possibile in qualsiasi momento aggiungere ulteriori elementi, e si occupa di gestire l'aggiornamento della posizione e dell'animazione corrente, entrambe basate sulle informazioni inviate dal controller.

Basandosi sull'analisi di particolari azioni dell'entità, come quella di morte, si occupa anche della rimozione di queste dallo schermo. Anche le classi che mantengono le informazioni sugli sprite attivi sono state create basandosi sui principi di ereditarietà e polimorfismo, ottenendo così una diversificazione fra le tipologie di figure potenzialmente visualizzabili.

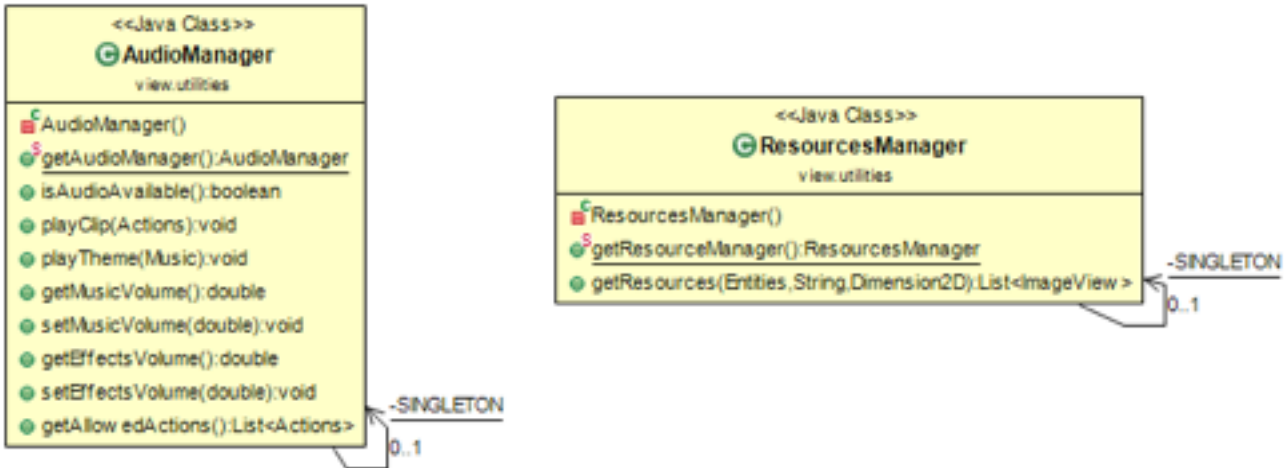


La distinzione è stata fatta principalmente tra sprite animati e non, aggiungendo ai primi anche le funzionalità per l'aggiornamento dell'animazione. In questo modo si sono ottenuti due tipologie di soggetti che possono popolare il mondo di gioco :

-Statiche, che sono usate per rappresentare le parti invariabili del mondo, quali possono essere i muri e le piattaforme;

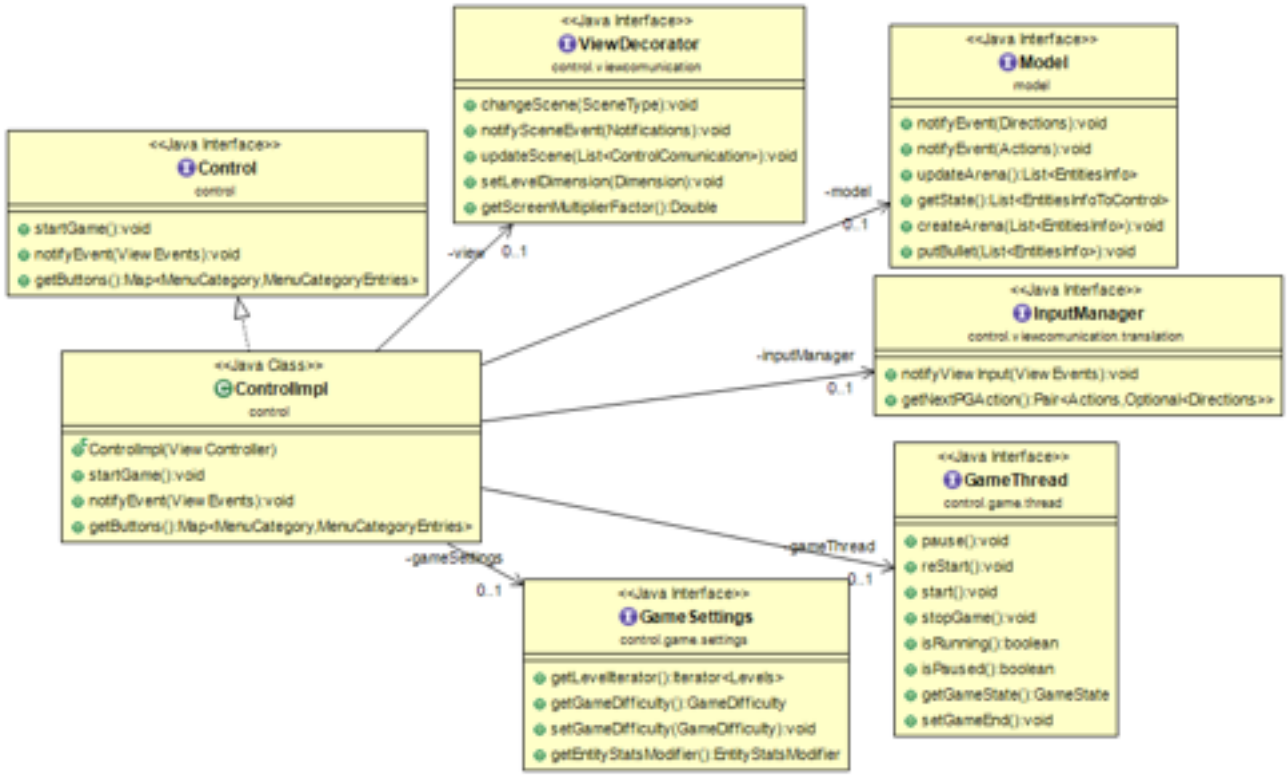
-Dinamiche, usate per gli elementi attivi del mondo di gioco, come personaggi e mostri.

La gestione delle immagini è stata affidata a una singola classe, il ResourceManager, creata utilizzando il Singleton pattern in modo da avere una singola istanza possibile durante l'esecuzione dell'applicazione, e così poter utilizzare una sorta di meccanismo di caching delle risorse, riducendo gli accessi in lettura al disco.
Una struttura analoga è stata usata anche per l'AudioManager, incaricato di gestire le musiche di gioco e gli effetti sonori.

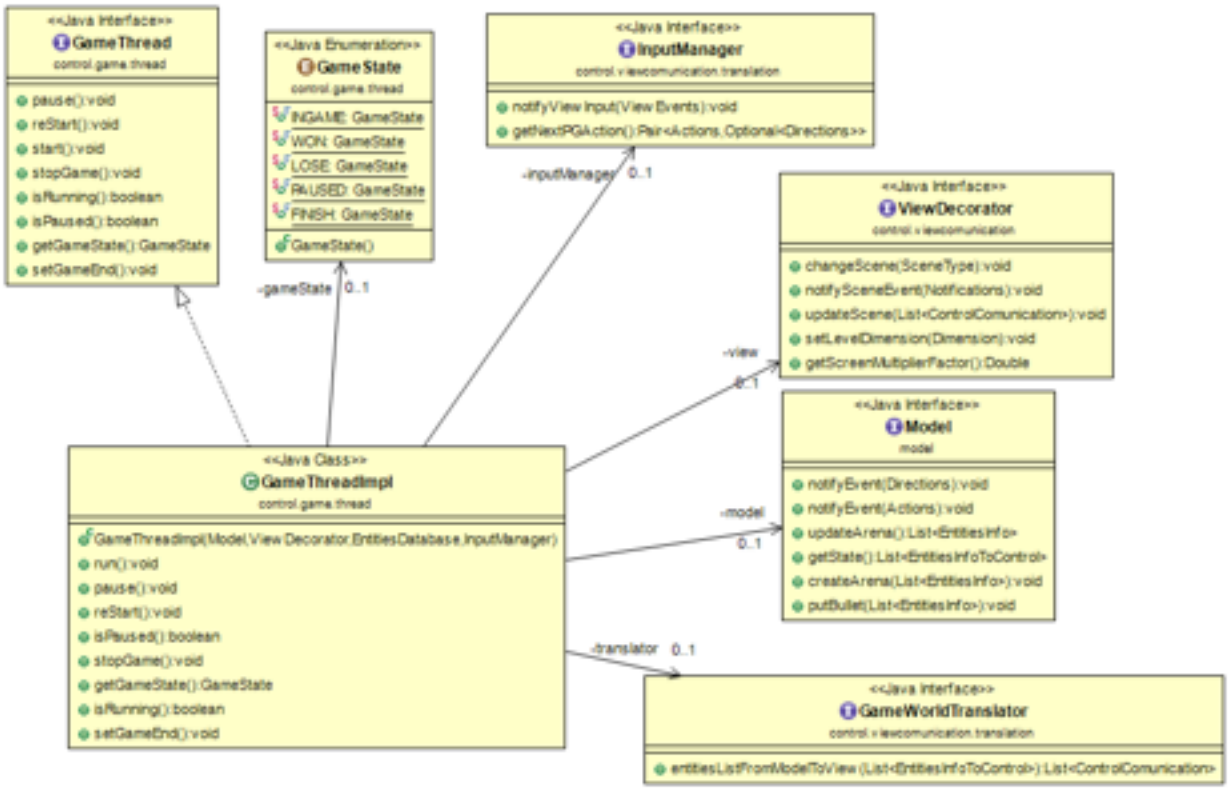


Per quanto riguarda la divisione in package, è stata fatta cercando di tenere divise per ambiti di funzionamento le diverse classi e quindi ottenere un accesso veloce ai diversi moduli che compongono la struttura. Si è cercato anche di ridurre la visibilità delle classi non necessarie al controller in modo da avere due domini completamente separati.

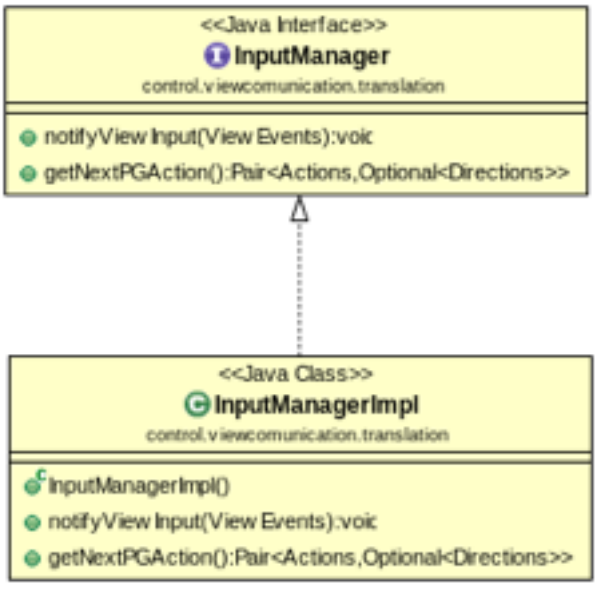
2.2.3 Design Dettagliato Control



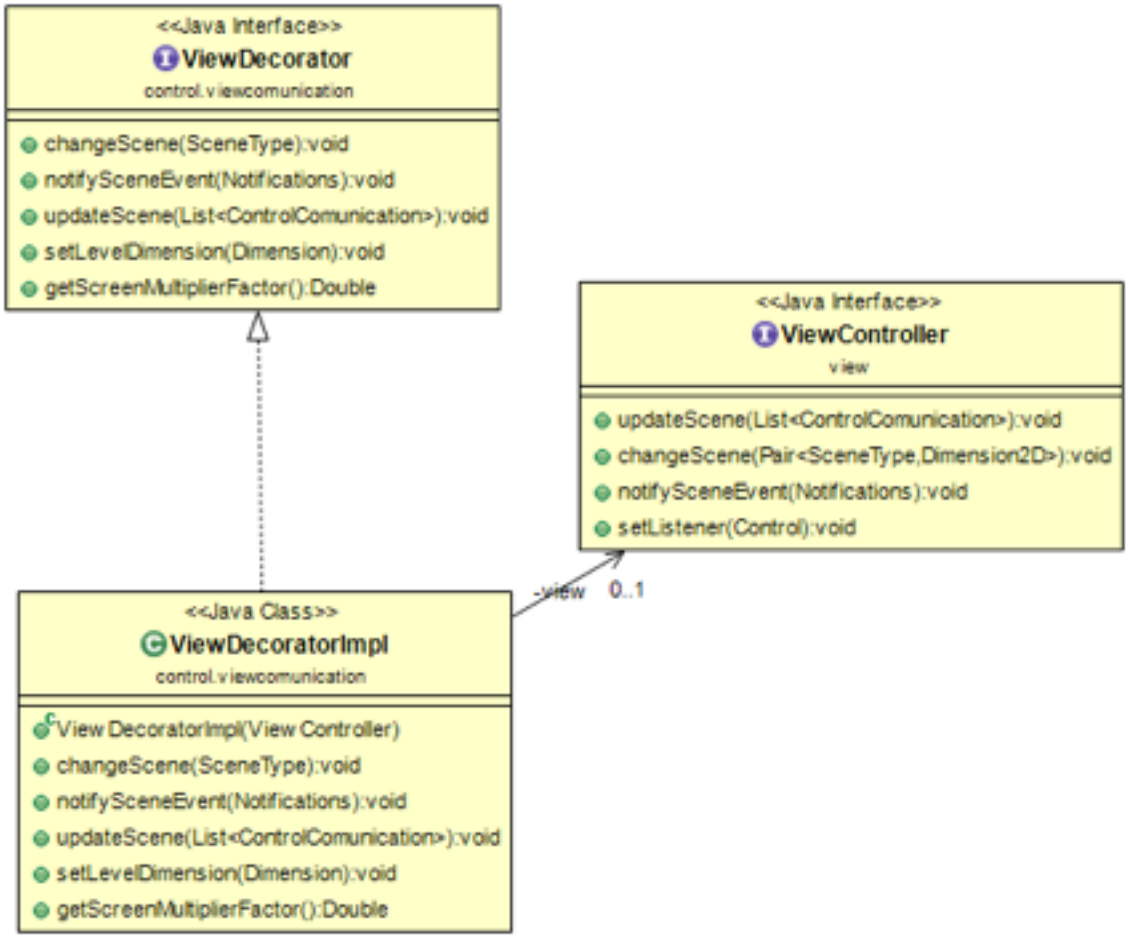
Per quel che concerne la parte architetturale da me sviluppata, come descritto nelle sezioni precedenti, la classe che interagisce direttamente con le altre due parti è il ControlImpl, che implementa l'interfaccia Control. Questa classe è registrata come observer nella view ed ha istanziati internamente sia model che view. La parte centrale dell'architettura alla base del funzionamento del gioco vero e proprio, inteso come svolgimento di un livello, è rappresentata dal GameThread. Questo thread, contenuto come campo nella classe principale ControlImpl, presenta un loop che periodicamente interagisce con il model per portare all'update del mondo di gioco, ne richiede lo stato completo e , dopo un'opportuna traduzione, lo trasmette alla view.



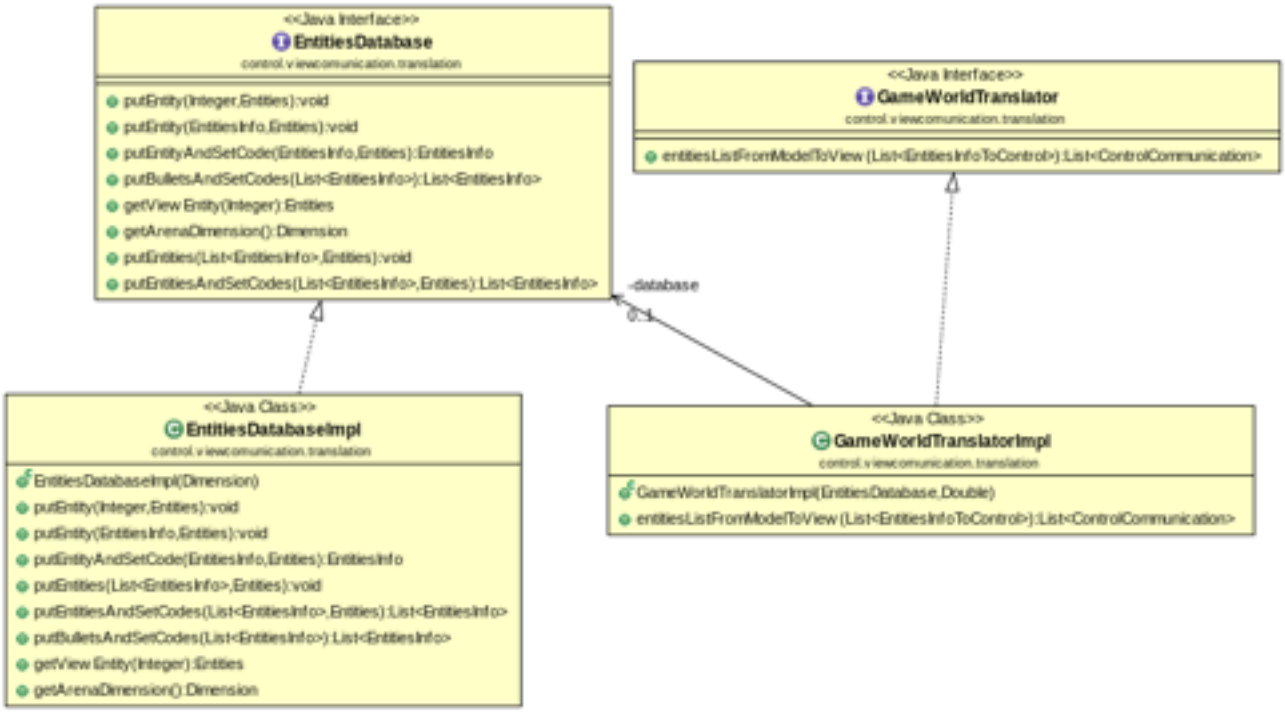
Questa architettura multi-thread permette al gioco di essere reattivo e di gestire sia input derivanti dal mondo di gioco che, in modo asincrono rispetto al game loop, dalla view. Questo ha portato però ad ovvie difficoltà che contraddistinguono la programmazione concorrente, sia per gli input derivanti dal thread della view, che sono da trasmettere al model, sia per quel che riguarda le notifiche da mandare alla view per il cambio di schermata. Il primo problema è stato risolto con un InputManager, ossia un oggetto passatto per riferimento al thread, che presenta metodi synchronized e salva al suo interno le azioni che il PG deve eseguire a seconda dei pulsanti attualmente premuti. Questa scelta ha inoltre permesso di ottimizzare il sistema di input, in quanto i segnali derivanti dalla view, di solo inizio e fine pressione dei tasti, non risultavano ottimali per la fluidità del gioco e necissitavano, quindi, di



un sistema di controllo e traduzione che li combinasse in modo opportuno.
Il secondo problema, ossia quello del cambio di scena, è stato invece da me risolto con una decorazione della view attraverso la classe ViewDecoratorImpl. Questa classe rende synchronized i metodi che lo necessitano. Se questa soluzione inizialmente rappresentava inizialmente solamente una sicurezza maggiore, in quanto con opportuni controlli era possibile evitare che due thread chiamassero lo stesso metodo contemporaneamente, si è poi rivelata conveniente quando, per limitare i problemi derivanti da un'eventuale cambio di UI, si è deciso di delegare al control la traduzione delle dimensioni del mondo di gioco in relazione alla dimensione dello schermo. Il ViewDecorator è infatti in grado, previo passaggio della dimensione del livello, di chiamare i metodi di cambio scena con un opportuno fattore moltiplicativo.

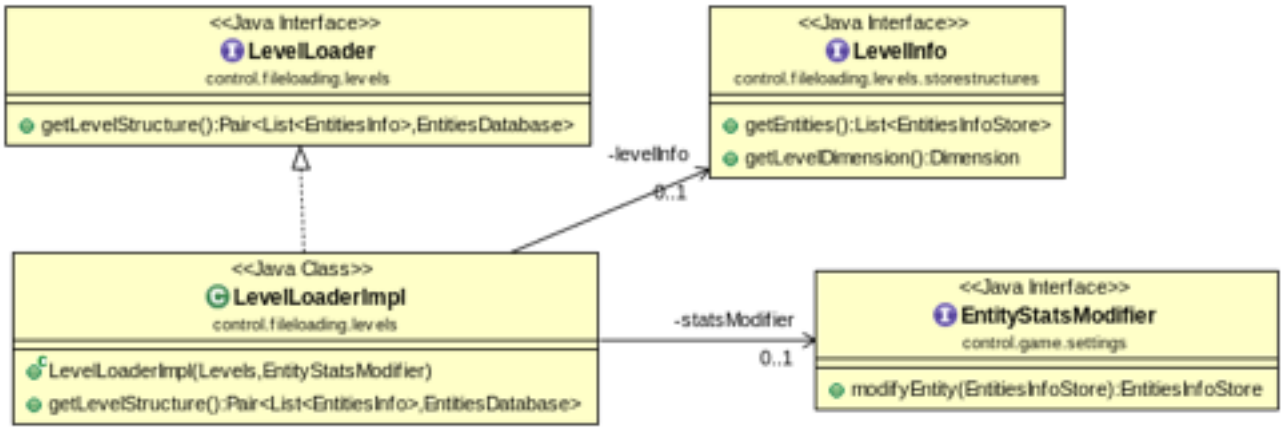


Questo fattore viene utilizzato anche per l'istanziazione dell'oggetto, il GameWorldTranslator, preposto alla traduzione della posizione delle entità da model a view che presentano, coordinate cartesiane speculari rispetto all'asse delle y. Questo oggetto di traduzione si avvale inoltre di metodi presenti in una classe di utility e di un “database” delle entità. Abbiamo infatti deciso, per comodità, di legare ad ogni entità del mondo di gioco un codice intero univoco ed, in base a questo, alla view viene passato un'identificativo, rappresentato da un'enumerazione, che le permette di reperire correttamente l'immagine da stampare a video.

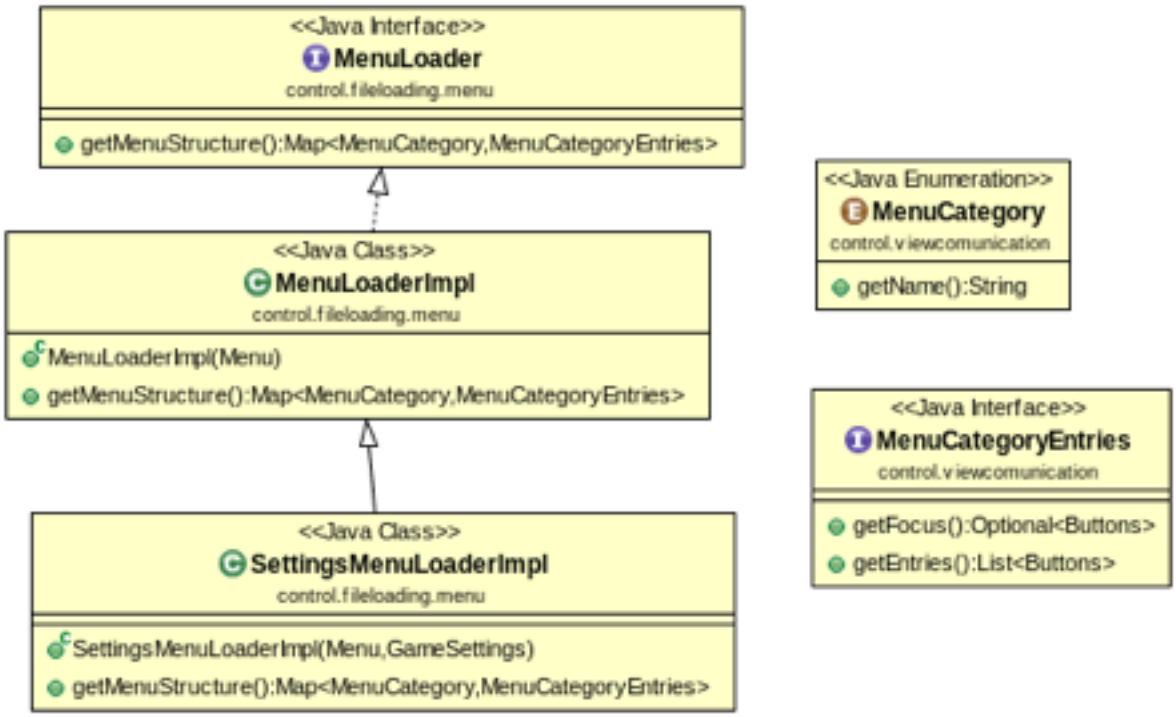


Ultima parte rilevante del controller da me sviluppata è quella di file loading. Per questa parte ho deciso di affidarmi ad una libreria esterna di serializzazione oggetti, Gson. Questa libreria serializza gli oggetti in modo largamente comprensibile e questo rende possibile modificare i file nei quali sono salvati con un semplice editor di testo. Questa caratteristica è risultata fondamentale, sia per la creazione di livelli, sia per il cambiamento dei settings del gioco o della struttura dei menu durante lo sviluppo. Il sistema di caricamento dei file è stata una parte critica della mia architettura. Inizialmente ho pensato di affidare questo compito ad una classe statica alla quale passare i riferimenti dei

file. Questa soluzione mi avrebbe permesso di risparmiare una leggera ripetizione di codice presente nell'architettura da me infine adottata, ma presentava, secondo il mio parere, una flessibilità minore. Il caricamento da file è stato affidato quindi a degli oggetti “loader”, che tentano di caricare nel costruttore gli oggetti serializzati e li salvano al loro interno. Questo porta a dover sviluppare “loader” diversi per il caricamento di oggetti diversi, senza la possibilità di adottare un'ereditarietà significativamente conveniente a causa, appunto, dei diversi parametri necessari per caricare oggetti di natura diversa. Questo sistema permette però di adottare sistemi di elaborazione degli oggetti serializzati. Questi “loader” infatti fungono da decoratori degli oggetti che caricano e con semplici strategie ereditarie, che consistono in semplici override di metodi, è possibile modificare, in caso di necessità, questi parametri al momento del get. Questa strategia è stata attuata sia al momento del caricamento del menu di settings che al momento di caricamento del livello. Al caricatore di livelli viene infatti passato un oggetto che elabora le statistiche delle entità a seconda della difficoltà, mentre il menu di settings viene correttamente settato a seconda delle opzioni attualmente attive.

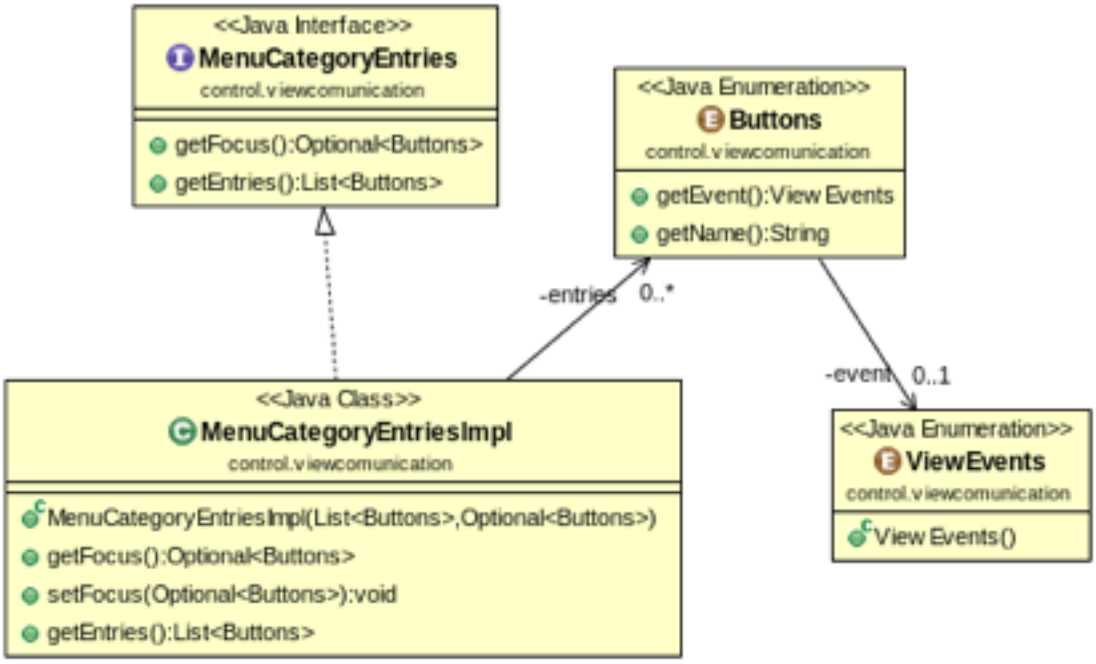


Questo sistema, a mio parere, rende il software altamente estendibile, permettendo di cambiare l'interpretazione dei file senza necessariamente andare a modificarli direttamente. Inoltre questi oggetti sono particolarmente adatti per creare una sorta di buffer, è infatti possibile evitare di caricare un menù già precedentemente caricato. Questa funzione non è stata implementata per mancanza materiale di tempo.



L'oggetto di elaborazione precedentemente citato nel caricamento dei livelli è l'EntityStatsModifier, oggetto che viene generato dall'oggetto che contiene i settaggi correnti del gioco. Al momento questo "elaboratore" risulta essere un mero decoratore della difficoltà di gioco, che contiene già al suo interno funzioni per le modifiche delle statistiche, ma è stato inserito per rendere maggiormente estendibile il sistema. Non si esclude, infatti, che in futuro la difficoltà possa eseguire trasformazioni più complesse sulle entità.

E' infine necessario descrivere brevemente le strutture di comunicazione che utilizzo con model e view. Le strutture che mando in input al model sono state da me reimplementate ed espanse aggiungendo i setter, questa scelta è stata presa per facilitarne l'elaborazione eseguita a seguito del file loading. La struttura di comunicazione con la view permette invece di decidere in modo specifico la struttura dei menu, compresi eventualmente pulsanti da visualizzare già selezionati.



Oltre alla parte di control è stata affidata a me la parte riguardante la gestione delle collisioni del model e quella riguardante l'HeroMovementManager , da svillupare in modo che risulti adeguato al movimento derivante dall'input utente. Durante lo sviluppo di quest'ultimo ho inoltre sviluppato un metodo, l'applyGravity, che si occupa di applicare la gravità sulle entità e che quinidi è stato utilizzato su tutte le entità del gioco. Se l'HeroMovementManager non necessita di commenti architetturali, in quando risulta essere più che altro un sistema di controlli, è necessario commentare la parte delle collisioni. Questa, infatti, oltre che presentare vari metodi di controllo e modifica delle entità, volte al fix delle posizioni ed alla simulazione di una banale intelligenza in caso di collisione, ha portato alla creazione di due classi. La prima contiene informazioni riguardanti le passate posizioni delle entità, mentre l'altra fornisce informazioni utili alla gestione del movimento re-attivo (e non pro-attivo) delle entità, ossia il movimento che dipende delle azioni di altri, come il movimento di un'entità appoggiata su una piattaforma. Si tratta sostanzialmente di una classe che restituisce tutte le entità che al momento si trovano sopra ad un'entità fornita.
La divisione in package del control rispetta la struttura base dell'architettura, con il package base “control” contenente la sola classe principale e con package separati per le classi relative al thread, ai diversi file loader, ai game settings, e alla comunicazione con la view. Questo ultimo package, che a livello logico poteva essere unico, è stato diviso in due ulteriori parti, uno contenente le mere strutture di

comunicazione, l'altra contenente le classi di traduzione, vistala grande dimensione che avrebbe avuto come package unico. Questa divisione in package è stata fatta principalmente per avere una divisione logica delle classi e rendere quindi più facile la navigazione del progetto.

Capitolo 3
Sviluppo

3.1 Test Automatizzato

Il team ha ritenuto opportuno effettuare un test automatizzato per la parte che riguarda il Model mentre alcuni test manuali per quel che riguarda la parte di Control e View.
Il test JUnit è formato da tre Test.

Il primo è stato sviluppato da Giovanelli e controlla che il model risponda in modo positivo a tutti gli input, che l'eroe si muova in modo corretto in base alle azioni che gli vengono settate e che i mostri seguano il loro pattern di movimento.
Il secondo è stato sviluppato da Magnani e controlla che le posizioni vengano fixate correttamente in caso di collisioni.

Il terzo è stato sviluppato da Giovanelli e controlla che il model in caso di input errati reagisca con eccezioni. Per esempio: duplicazione di codici, eroe assente, obiettivo assente e mostro che non rispetta i limiti di instanzia.

La parte di Control e View è stata testata manualmente per verificare che la formattazione e la resa dell'interfaccia su schermi a diversa risoluzione e computer con diversi sistemi operativi(Linux, derivata Debian ed Arch, MacOSX e windows) fosse coerente ed efficiente.

La view è stata testata anche su sistemi Arch Linux, dato che non presenta un completo supporto alla piattaforma di JavaFX. I test hanno avuto esito positivo.

Inoltre è stato testato che con frequenti cambi di menù, essi vengano caricati e visualizzati in modo corretto.

3.2 Metodologia di lavoro

Il team è riuscito con successo a far sviluppare ad ogni componente la proprio parte in modo autonomo.

Per far sì che questo accadesse abbiamo chiarito in modo esaustivo, in fase di Design, le interfacce di comunicazione tra le varie parti.
Giuseppe Pisano si è dedicato interamente alla View, non toccando altra parte del progetto.
Quest'ultimo si è dedicato alla ricerca e allo studio approfondito di JavaFX in modo tale da utilizzare con efficenza questo motore grafico.

Per quel che riguarda la parte di Magnani e Giovanelli, la divisione iniziale era quella del Control per il primo e del Model per il secondo.

Si è sentita l'esigenza, però, di affidare a Magnani una parte più corposa dello sviluppo ottenendo in questo modo una divisione più equa del lavoro.

Si è quindi giunti alla divisione finale:

-Matteo Magnani: Control e parte del Model (gestione delle collisioni e movimento dell'eroe).
-Joseph Giovanelli ha sviluppato la parte restante del Model, ovvero la creazione del mondo di gioco con le relative entità e i movimenti di esse.

Dato che Magnani si è occupato di parte del Model, egli e Giovanelli hanno dovuto sinergicamente stabilire delle interfacce più dettagliate relative al movimento.

Per ottenere una buona integrazione fra le parti sviluppate disgiuntamente dai membri del team sono state stabilite delle micro scadenze mano a mano che il lavoro veniva portato a compimento in modo da poter avere periodicamente un applicativo funzionante e testabile.

Uno strumento molto importante durante lo sviluppo è stato l'uso del DVCS Mercurial.

Esso ha permesso un efficiente lavoro a distanza tra i vari componenti del team.

3.3 Note di sviluppo

Per realizzare la View, avendo dovuto lavorare con delle API non spiegate a lezione e mai usate, ci si è avvalsi di varia documentazione web per comprendere appieno le funzionalità e l'uso dei servizi messi a disposizioni da JavaFX.
Le fonti sopra citate sono, per esempio:
-Documentazione Oracle

-StackOverFlow

-Vari blog..

Per il file loading è stata usata la libreria esterna Google Gson.
Si reindizza alla seguente pagina:
https://github.com/google/gson

Capitolo 4
Commenti Finali

4.1 Autovalutazione e lavori futuri

Il team si è trovato molto bene nel lavoro di gruppo, infatti i membri sono riusciti a collaborare e a interagire facilmente.

Secondo il nostro giudizio il progetto è complessivamente buono, considerato il tempo avuto a disposizione e le naturali difficoltà nel lavorare su un progetto software complesso.

Il nostro punto di forza è sicuramente l'estensibilità del nostro codice. Progettiamo infatti nei prossimi mesi di aggiungere features pensate durante lo sviluppo come per esempio l'introduzione dei power up, elementi di gioco con movimenti più complessi come piattaforme che cadono quando vengono toccate e casse reattive.
Gli strumenti per realizzarli sono già disponibili.

Ci siamo accorti che lavorando in gruppo si ha modo di imparare molto grazie alle conoscenze degli altri.

4.2 Appendice

4.2.1 Risorse Utilizzate

Musica:

-Tema Menù: Kalashnikov - Goran Bregovic.

-Effetti Gioco: OpenGameArt.org

Grafica: Vena Artistica di Giuseppe Pisano.

4.2.2 Storia

0011 giovani programmatori, afflitti ed amareggiati dal non trovare un'idea per un progetto universitario, uscirono a prendere una birra per appianare le turbe del loro spirito.

Ritrovatesi davanti a una Delirium Tremens, in questa notte grigia e uggiosa, il loro animo così grevemente sconvolto fu infine trucidato dall'acidità provocata dalla falsa promessa di quella dorata bevanda all'aroma di luppolo.
Forse fu proprio questa condizione al limite dell'umano, più vicina alla morte che alla vita, che portò Magno a partorire un'idea a dir poco folle:
SVILUPPARE UN GIOCO PLATFORM.

Questa idea, sembrata inizialmente geniale a quelle povere anime afflitte, si rivelò però fatale. Durante una lunga ed estenuante nottata di lavoro i nostri tre eroi persero conoscenza, trovandosi all'interno del loro grigio mondo delirante…

4.2.3 Guida Utente

I comandi del gioco sono:

-WASD per il movimento e salto;

-SPACE per lo sparo;

-ESC per la pausa.

Per usufruire appieno delle funionalità del gioco, quali l'audio, il sistema operativo delle essere supportato appieno da JavaFX