Relazione del progetto
Quest for Stuff

Per “Programmazione ad Oggetti”

Anno Accademico 2014-2015

Corso di Laurea in Ingegneria e Scienze Informatiche

Ravaioli Giacomo 0000652695

Ragazzini Samuele 0000655973

Portolani Filippo 0000634124

Indice

1 Analisi

1.1 Requisiti 3

1.2 Problema

2 Design

2.1 Architettura 4

2.2 Design dettagliato 6

2.2.1 Design del model 6

2.2.2 Design della view 7

2.2.3 Design del controller 8

3 Sviluppo

3.1 Testing 11

3.2 Divisione dei compiti 11

3.3 Note di sviluppo 12

4 Commenti finali

4.1 Conclusioni e lavori futuri 13

Capitolo 1

Analisi

1.1 Requisiti

Il software creato per questo progetto è un gioco “platform” 2d della tipologia “Hack'n'Slash” e quindi il suo scopo è puramente di intrattenimento. Il gioco è composto da una schermata principale in cui l'utente può decidere se cambiare le impostazioni, visualizzare i punteggi più alti oppure iniziare a giocare. Vi sarà poi una schermata che permette di scegliere il personaggio. L'avventura è composta da quattro livelli la cui struttura varia in maniera casuale. Questi sono a loro volta suddivisi in quattro aree di difficoltà: facile, medio, difficile ed estrema. All'interno del livello vi sarà lo “spawn” casuale di nemici che avranno abilità differenti a seconda della tipologia e che il giocatore dovrà uccidere o evitare. Sarà possibile, inoltre, visualizzare il numero delle vite rimaste, il tempo trascorso e il punteggio totale. Una volta terminati i livelli, si sbloccherà la stanza del boss finale. Infine il punteggio ottenuto verrà salvato e mostrato nella schermata degli “highscore”.

1.2 Problema

Il software dovrà essere in grado di gestire diversi aspetti strettamente legati alla creazione di un platform. Un primo importante problema è la gestione di aspetti quali la collisione, l'input dell'utente e il comportamento dei nemici. Un secondo problema da considerare è la natura random della creazione delle mappe e la conseguente scelta di quali parti utilizzare all'interno del livello così come avviene per lo spawn dei nemici. Il sistema deve, inoltre, prevedere l’inserimento futuro di nuovi personaggi con poteri speciali particolareggiati in maniera facilitata. Essendo il gioco sviluppato attraverso un susseguirsi di livelli è anche necessario mantenere le informazioni dei risultati ottenuti durante quest’ultimi in maniera da permettere tramite salvataggio su file un eventuale highscore.

Capitolo 2

Design

2.1 Architettura

Per sviluppare il progetto utilizziamo il pattern architetturale MVC. La parte di View è preposta alla visualizzazione a schermo del videogioco e di tutti gli elementi ad esso appartenenti. Della parte di Controller del pattern fanno parte tutti quegli handler che vanno a gestire gli eventi generati dall'interazione, in particolare: quelli dell'utente con il sistema e quelli del personaggio scelto con i nemici. Sono compresi inoltre anche quelli che devono creare e fare il rendering dei vari elementi del videogioco. La definizione di quest'ultimi, compresa la mappa, appartiene invece al Model del suddetto pattern.

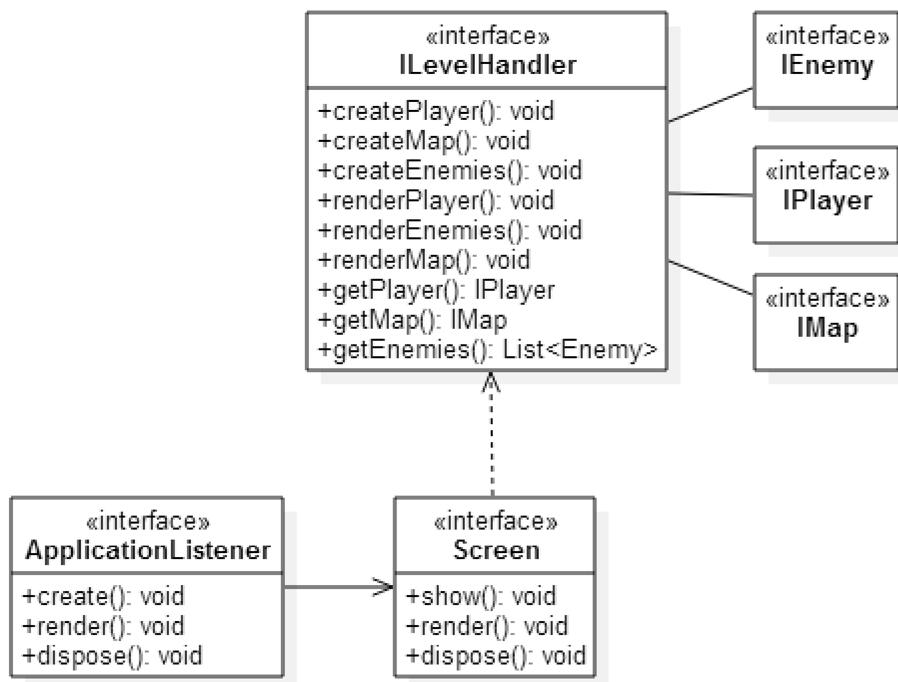

Figura 1 - Schema UML architetturale MVC

ApplicationListener è l'interfaccia principale del gioco. Questa può avere un solo screen attivo per volta. Lo screen dipende da ILevelHandler il quale porta ad una modifica della view in base a ciò che accade nel mondo. Il model è rappresentato da IMap, IEnemy e IPlayer ai quali il controller principale ILevelHandler accede in lettura e scrittura.

Di seguito riportiamo un altro UML riguardante l'interazione tra l'ApplicationListener e gli altri controller principali necessari alla realizzazione di un video-game.

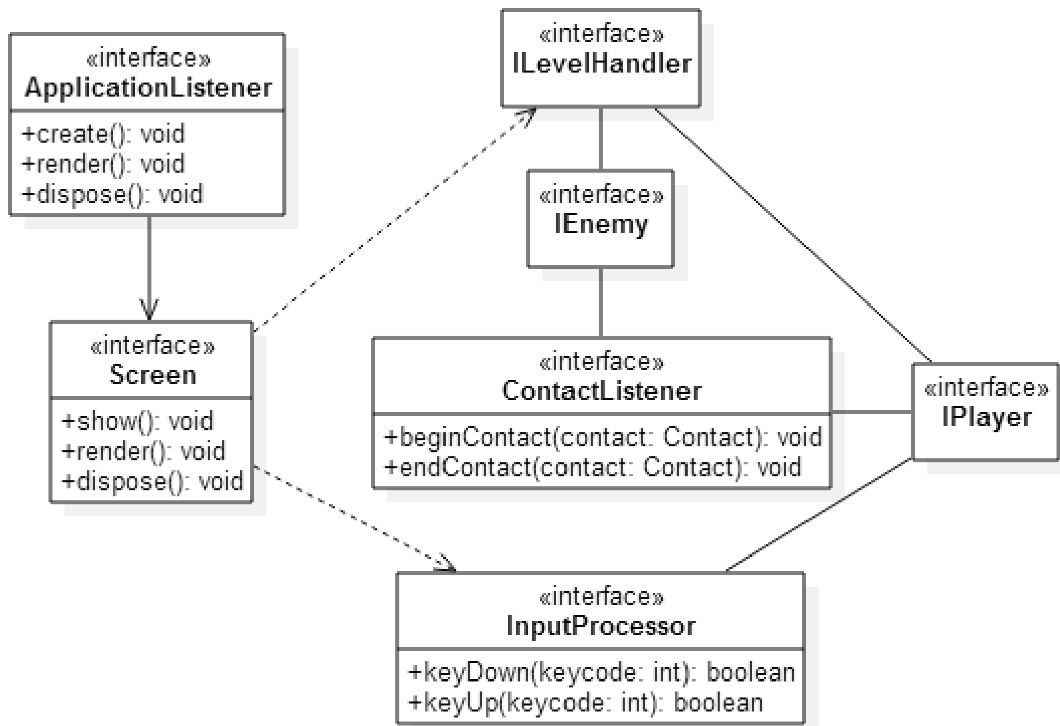

Figura 2 - Schema UML architetturale input e contact controller

Nel sistema sono presenti altri due interfacce di controller importanti come si vede nella figura sopra, InputProcessor e ContactListener. Il primo necessario per controllare le azioni dell'utente modificando la view mentre il secondo per registrare eventi di contact tra i vari oggetti della mappa compresi player e nemici.

2.2 Design dettagliato

In questa sezione entriamo nel dettaglio della progettazione del sistema sviluppato. Per mantenere il processo di progettazione il più chiaro e lineare possibile abbiamo deciso di suddividere tale sezione in tre parti: design del model, design della view e design del controller. Questo in quanto membri del team si sono divisi equamente la parte di model mentre il controller è stato progettato in collaborazione. Di conseguenza non verranno di seguito specificato quale membro del team ha progettato cosa ma verrà descritto in dettaglio nella sezione successiva.

2.2.1 Design del model

Di seguito mostriamo gli UML di dettaglio relativi alla parte di model, che permettono di capire come sono stati risolti i problemi relativi a tale componente del MVC identificati in fase di analisi.

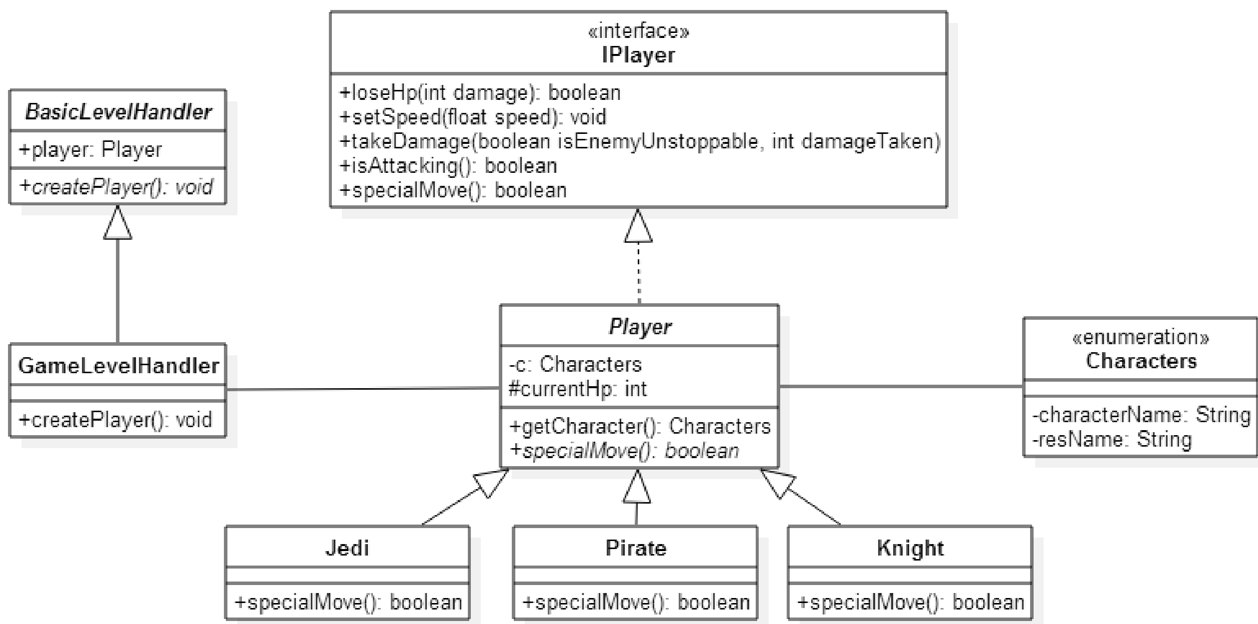

Figura 3 - Schema UML di dettaglio model personaggio

Il software deve permettere l'inserimento di nuovi personaggi al pool di quelli disponibili facilitando tale operazione. Abbiamo quindi pensato di sfruttare il pattern Factory Method.

IPlayer definisce l'interfaccia alla quale tutti i personaggi devono aderire. Essendo questi ultimi differenti solo nella definizione di valori di statistiche bonus e dell'attacco speciale estenderanno tutti una classe astratta Player implementazione dell'interfaccia sopra citata.

Una volta che il player ha deciso il personaggio con cui giocare tramite un'apposita schermata il GameLevelHandler, controller delegato alla gestione di un livello della partita, andrà a scegliere quale specializzazione di Player instanziare.

Per quanto riguarda i nemici quest'ultimi hanno meno specializzazioni, si distinguono infatti per la sprite e conseguentemente per l'animazione. Inoltre il gioco prevede uno o più boss di conseguenza si è deciso di modellare il nemico tramite l'interfaccia IEnemy che definisce i metodi ai quali tutti i nemici e il boss devono aderire e definire quest'ultimo tramite l'estensione di un normale nemico. Di seguito lo schema UML di dettaglio del model del nemico.

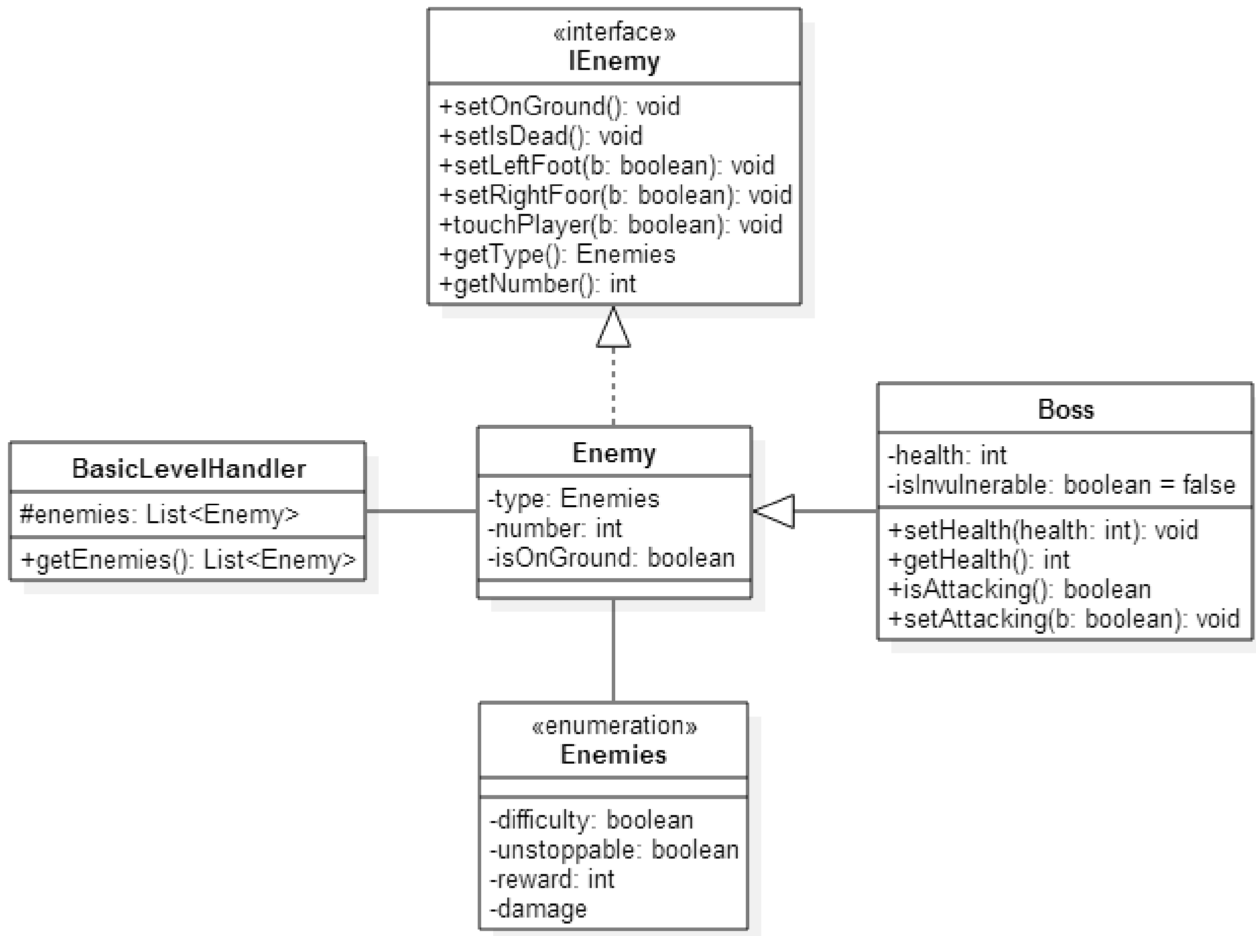

Figura 4 - Schema UML di dettaglio model nemico

La parte di modello relativa alla mappa è stata sviluppa in modo analogo a quella soprastante pertanto non stiamo a mostrare l'UML relativo in quanto del tutto simile. Specifichiamo tuttavia che ogni mappa aderisce all'interfaccia IMap implementata dalla superclasse BasicMap. Inoltre la difficoltà maggiore nella progettazione della mappa si è ritrovata nella sua creazione per evitare la perdita di frame.

2.2.2 Design della view

In questa sezione trattiamo il design di dettaglio della view. La struttura del videogame impone di avere diverse schermate, una per ogni schermata di gioco che si vuole mostrare, accomunate da diversi metodi. Di seguito mostriamo l'UML relativo a tale parte del MVC. Vista la possibilità della creazione di numerosi screen abbiamo deciso di utilizzare come pattern un Template Method. Un interfaccia definisce i metodi che andranno implementati da tutti gli screen mentre una classe astratta "MyScreen" implementa i metodi comuni a tutti gli screen che la andranno ad estendere. Nello schema a seguire sono state inserite le specializzazioni principali della classe astratta MyScreen. Ve ne sono diverse aggiuntive per le varie sezioni del gioco (HighscoreScreen, PickScreen etc.) tuttavia abbiamo deciso di non mostrarle in tale schema in quanto si comportano analogamente a quelle presenti. La scelta della screen da utilizzare viene fatta in base alla fase del gioco in cui ci si trova ed

è scatenata da eventi quali il contatto tra un player e un oggetto o dall'input dell'utente.

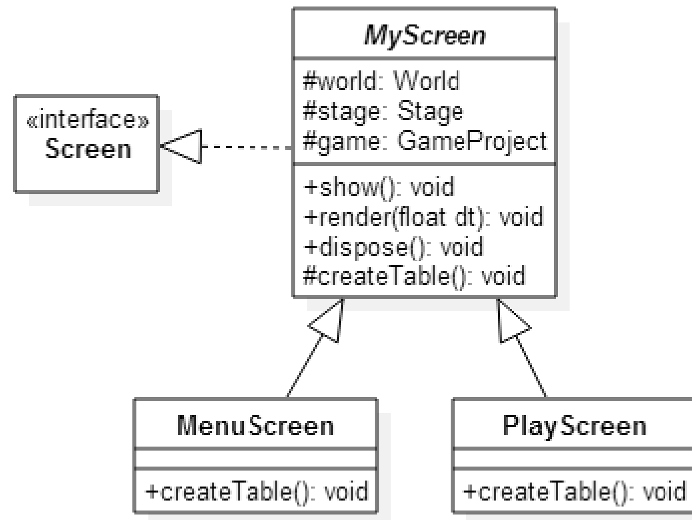

Figura 5 - Schema UML di dettaglio della View

2.2.3 Design del controller

La parte di controller come scritto nella sezione precedente comprende le classi necessarie per gestire l'input dell'utente, i contatti e l'intelligenza artificiale dei nemici nonché i meccanismi per mantenere informazioni durante tutta la durata della partita sul personaggio giocato e il player.

Per le classi relative alla gestione dei livelli è stato utilizzato il Template Method, sfruttando le estensioni della classe astratta BasicLevelHandler, che a sua volta implementa l'interfaccia ILevelHandler come nell'UML di seguito:

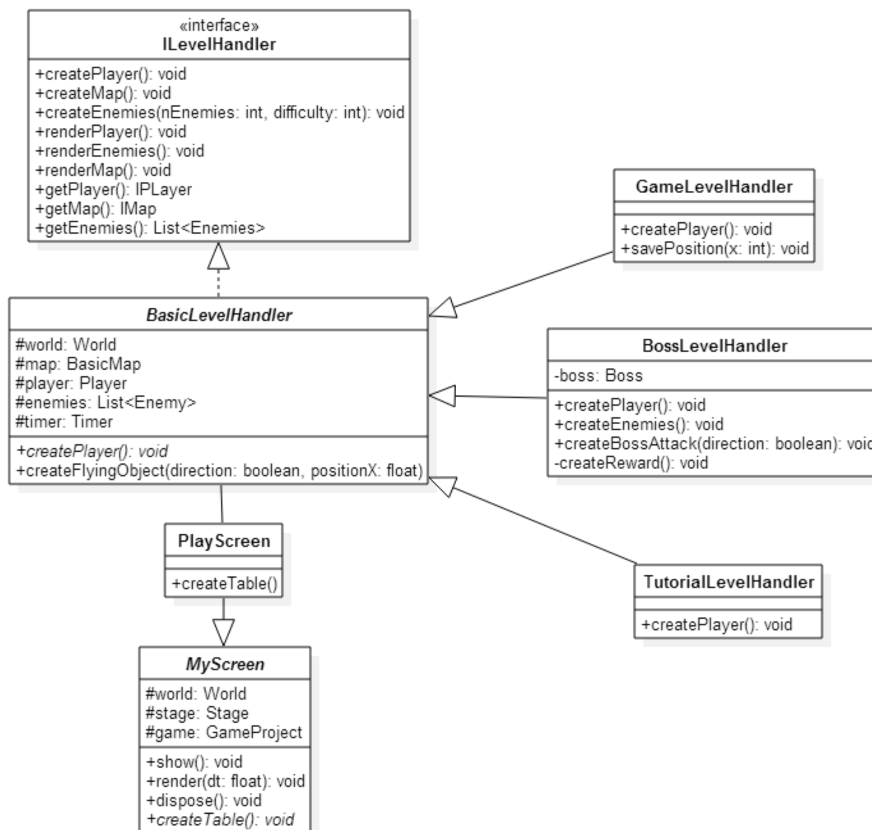

Figura 6 - Schema UML di dettaglio del level controller

In base alle condizioni nella classe PlayScreen viene inizializzata la corretta istanza di sottoclasse.

Per quanto riguarda l'intelligenza artificiale dei nemici, essa viene gestita dalle classi che implementano l'interfaccia IArtificialIntelligence, come mostrato nello schema:

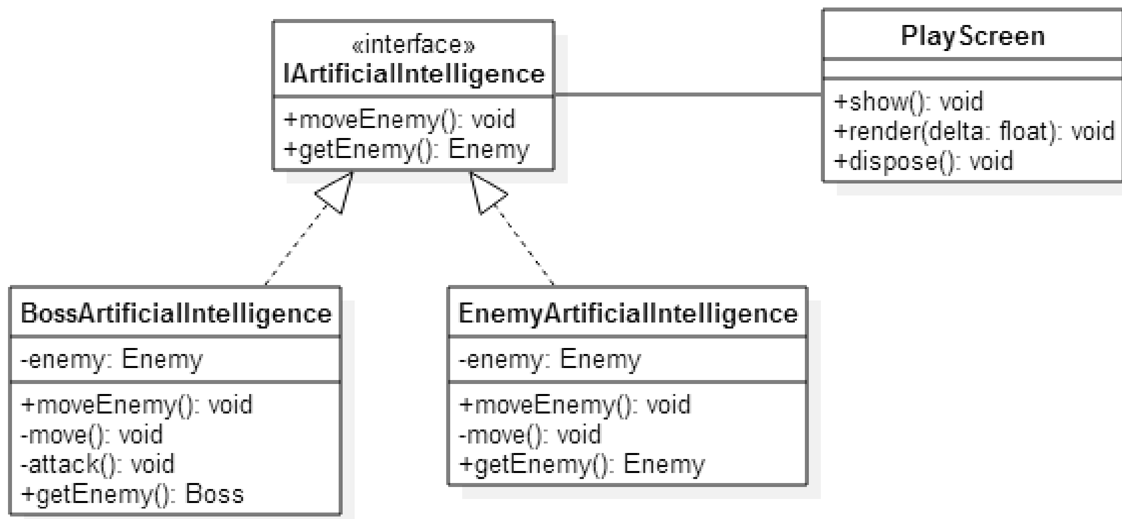

Figura 7 - Schema UML di dettaglio AI controller

Questo permette l'aggiunta di tipologie di nemici (boss o nemici base) differenti attraverso l'estensione delle classi che aderiscono all'interfaccia.

Un importante problema che si è posto in fase di analisi è stato il dover mantenere per tutta la durata della partita le informazioni relative al giocatore, così come quelle relative al personaggio utilizzato.

Per poter fare ciò abbiamo deciso di creare la classe Engine acceduta attraverso i metodi getter preposti, nella quale vengono mantenuti i dati importanti durante lo svolgimento della partita. Per tale classe come si può intuire dallo schema sottostante abbiamo utilizzato il pattern Singleton.

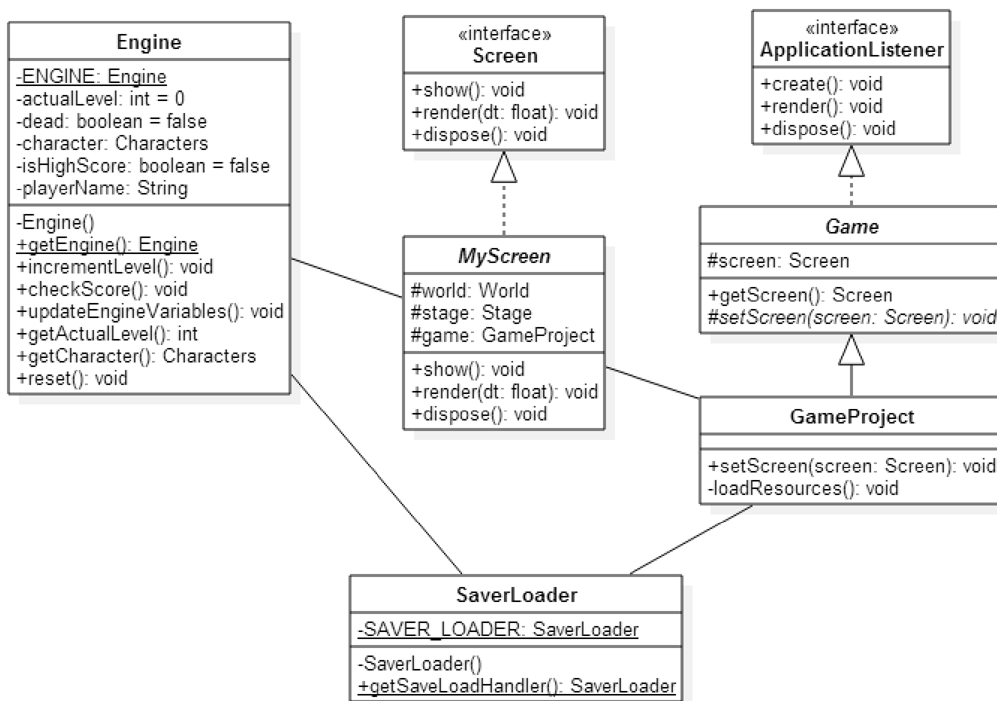

Figura 8 - Schema UML di dettaglio engine

La classe Engine è inoltre dotata di un metodo per poter riportare ai valori di default tutte le variabile necessarie per il compimento di una nuova partita.

Una strategia del tutto analoga abbiamo deciso di utilizzarla per altre due classi. La prima è quella responsabile del salvataggio/caricamento su/da file degli highscore realizzati dai giocatori del videogame. La seconda è il “ResourcesManager” che è la classe utilizzata per caricare all’avvio del gioco tutte le risorse che vengono quindi salvate in apposite strutture dati in modo da poterle utilizzare ogni volta in caso di necessità senza aggiungere ritardi al video-game causando perdita di frames.

Capitolo 3

Sviluppo

3.1 Testing

Vista la natura dell'applicazione non è stato possibile adoperare testing automatizzato per il controllo di errori o per verificarne il corretto funzionamento. Abbiamo provveduto tutti insieme a svolgere test manuali sull'applicazione unitamente ad un piccolo alpha-testing svolto da parte di nostri conoscenti.

3.2 Divisione dei compiti

All'interno del team il lavoro è stato suddiviso dopo esserci dati delle linee guida su come procedere e delle dead-line entro le quali portare a termine porzioni specifiche dei nostri compiti per poterle poi integrare.

Il lavoro è stato suddiviso come segue:

1) Samuele Ragazzini si è occupato della parte di model relativa ai nemici, al boss e la parte di controller necessaria al loro movimento ed attacco (quest'ultimo per quanto riguarda il boss) ovvero le AI.
2) Filippo Portolani si è occupato della parte di model relativa al player, dell'AnimationHandler (controller necessario alla corretta visualizzazione delle animazioni dei personaggi), dell'InputPlayerHandler (controller dedito alla gestione dell'input da parte dell'utente) e unitamente a Giacomo Ravaioli della parte di model relativa alle mappe.
3) Giacomo Ravaioli si è occupato della view, quindi di tutte quelle classi specializzazioni della classe astratta MyScreen, come sopra insieme a Filippo Portolani della parte di model relativa alle mappe e della maggior parte delle enumeration e delle classi nel package "utilities" necessarie per lo svolgimento di compiti specifici all'interno dell'intero progetto.

La parte di progetto relativa al main controllor (ILevelHandler) e le sue specializzazioni, ovvero le classi presenti nel package "controller.level" sono state svolte da tutti e tre i membri del gruppo in quanto unione delle tre sotto-parti nelle quali il progetto era stato inizialmente suddiviso. Ci teniamo ad informare che la suddivisione sopra scritta comunque potrebbe aver ricevuto qualche modifica in quanto in diversi casi abbiamo avuto scambi di idee che hanno portato eventualmente a preferire una idea di sviluppo rispetto ad un'altra.

Il processo di integrazione come scritto sopra è stato effettuato a dead-line decise da tutti i membri entro cui ognuno sarebbe dovuto andare avanti nella sua parte di progetto.

Durante tutto il processo di sviluppo del video game è stato utilizzato come DVCS Bitbucket unitamente a Mercurial. Si è cercato di lavorare in branch separati o sul branch di default effettuando merge, pull o synchronize al momento delle dead-line.

3.3 Note di sviluppo

Per lo sviluppo di questo progetto abbiamo deciso di utilizzare la libreria esterna “libGDX”.

Non avendo infatti conoscenze sulla creazione di videogame ci siamo appoggiati a tale libreria che ci ha fornito le interfacce per la creazione di handler basilari quali InputProcessor e ContactHandler, del gioco stesso (ApplicationListener) e delle varie schermate (Screen).

Ovviamente il processo per utilizzare al meglio tale libreria si è sviluppato anche attraverso la consultazione di blog e video dedicati a tale libreria nonché e soprattutto alla libGDX API.

Vogliamo inoltre specificare che non possediamo i diritti delle musiche utilizzate all’interno del progetto. Queste sono state utilizzate solo per fini didattici. Non vi è alcuna intenzione da parte dei membri di farne un uso illecito, guadagnarci soldi o spacciarle per proprie.

Infine, vogliamo sottolineare che al momento dell’import tramite Eclipse con opzione “Clone existing mercurial repository” alla fine del processo di import è presente un bug che non permette di vedere nel workspace nel quale si è fatta l’operazione il progetto. È necessario quindi, dopo questo primo import, svolgerne un secondo selezionando “Existing projects into workspace” andando a scegliere il progetto nella cartella del workspace.

Capitolo 4

Commenti finali

4.1 Conclusioni e lavori futuri

In generale l'esperienza di sviluppare un video game pur semplice è stata appagante tuttavia l'utilizzo del DVCS Bitbucket è stato a tratti problematico vista l'inesperienza all'utilizzo di quest'ultimo e per uno dei membri del team è risultata piuttosto difficoltosa a causa di svariati problemi. Motivo per cui nel susseguirsi di commit possono essere presenti commit fatti nell'arco di secondi o minuti, o l'assenza di uno dei membri del team tra questi.

Ovviamente il lavoro può essere migliorato ed integrato. Si potrebbe pensare ad esempio all'aggiunta di una modalità 1v1 nella quale due player giocano nella stessa mappa o si sfidano in una creata apposta oppure farne una versione mobile.