29 FEBBRAIO 2016

RELAZIONE PROGETTO JOHNNY 2D

PROGRAMMAZIONE AD OGGETTI

MATTEO MINARDI, PAOLO VENTURI, GIACOMO PASINI

Sommario

Analisi

L'applicazione che si dovrà sviluppare sarà un videogame di genere Roguelike ispirato a "The Binding of Isaac" (https://it.wikipedia.org/wiki/The_Binding_of_Isaac), nel quale sarà presente un personaggio principale (eroe) il cui scopo è quello di sconfiggere il maggior numero di nemici all'interno di un'arena e acquisire il punteggio migliore. Gli elementi essenziali per un gioco di genere Roguelike sono: una generazione casuale dei livelli, lo scontro tra più nemici e la morte permanente in caso di sconfitta. Nella nostra versione lo stile di gioco non sarà a turni ma in tempo reale.

Requisiti

L'applicazione finale dovrà esporre le seguenti funzionalità di base:

- L'esperienza di gioco sarà ambientata in un'arena con un insieme di nemici generato man mano col passare del tempo e che l'eroe dovrà sconfiggere al fine di vincere la sfida.
- All'eroe sarà associato un punteggio ad identificare la bravura del giocatore. Tale punteggio potrà essere incrementato sia sconfiggendo i nemici sia raccogliendo appositi collezionabili.
- All'eroe sarà inoltre associata una barra di salute che ne determina il tempo di gioco a disposizione. Tale barra si ridurrà ogniqualvolta l'eroe subirà del danno dai nemici. All'esaurimento di tale barra la partita terminerà e sarà mostrato il punteggio ottenuto dal giocatore. Questi potrà quindi scegliere se continuare a giocare o terminare l'applicazione.
- L'eroe deve essere in grado di sparare proiettili e potersi muovere liberamente all'interno dell'arena di gioco delimitata da muri non oltrepassabili.

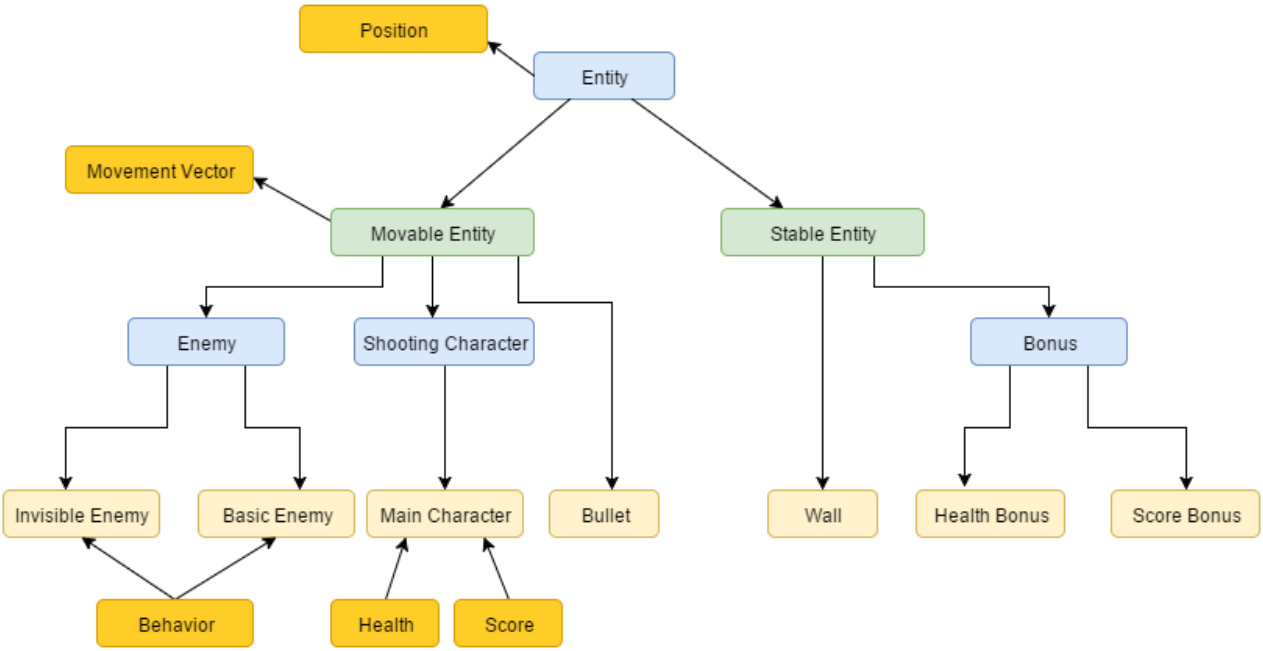
Analisi e modello del dominio

Questo gioco essenzialmente consiste nello sparare e fuggire da nemici rappresentati da mostri che si dirigono nella direzione del personaggio principale al fine di sconfiggerlo o comunque provare a farlo.

Il gioco si svolge, per ora, all'interno di una singola arena delimitata da muri che impediscono alle entità di uscire da essa.

Per rendere il gioco più avvincente è stata introdotta la possibilità di raccogliere da terra collezionabili che aiutano il giocatore a sopravvivere, a raggiungere un punteggio più elevato e quindi provare a migliorare il proprio record personale.
Sempre per lo stesso motivo la difficoltà cresce mano a mano che il tempo passa a causa dello spawn casuale di nuovi nemici che si muovono verso il personaggio.

Questa breve analisi di quello che è il gioco consente di giungere ad una serie di "Entità" necessarie durante la partita. Uno scheletro di modello sarà quindi composto da un personaggio principale controllabile che spara proiettili, nemici, bonus e muri.



Dopo il precedente ragionamento siamo giunti a dividere il modello essenzialmente in due rami fondamentali di entità:

- Le entità mobili: cioè quelle che possono esplorare l'arena di gioco e muoversi all'interno di essa.
- Le entità stabili: cioè quelle che lasciano la loro posizione immutata durante lo svolgersi del gioco.

Ovviamente i vari nemici, di ogni tipo essi siano, fanno parte delle entità che possono muoversi e, allo stesso modo, il personaggio principale, oltre ad essere in grado di sparare proiettili, è in grado di spostarsi seguendo direzioni diverse. Il personaggio è modellato da una salute che rappresenta la quantità di vita rimasta e un punteggio che indica quanti mostri sono stati uccisi e quanti punti sono stati ricavati da essi.

Al contrario i muri e i diversi tipi di bonus non possono muoversi, anzi, una volta creati, restano fissi nella loro posizione occupando il loro posto all'interno dell'arena.
Allo stesso modo la struttura che delimita il rettangolo giocabile viene modellata come una sorta di arena che è composta dai diversi pezzi di mura necessari a realizzarla.

Tutti gli elementi che modellano il gioco cooperano e convivono all'interno di un ambiente che contiene le varie entità controllandone le varie operazioni in modo da far procedere il gioco con il passare del tempo e il susseguirsi degli eventi.

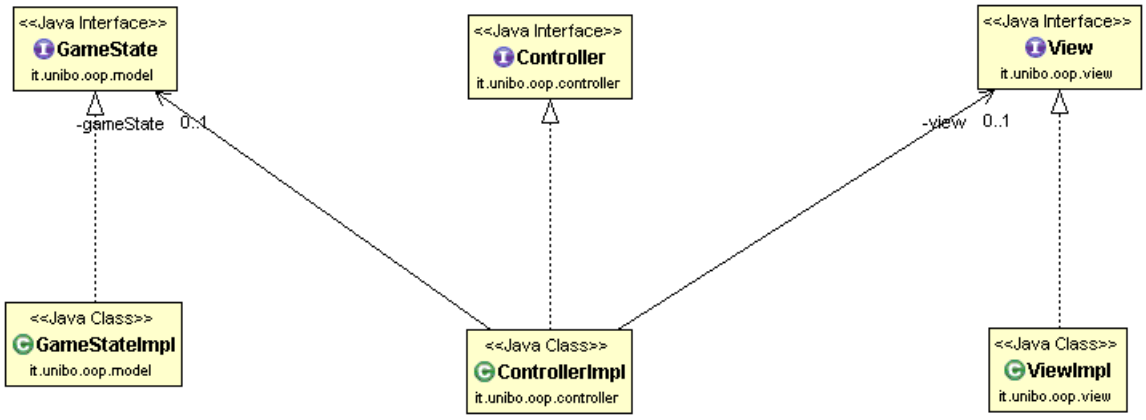
Il modello è stato volontariamente circoscritto ad una serie limitata di entità e funzionalità in vista del monte ore stabilito. Lo scheletro del modello permette la sua espansione in vista di futuri miglioramenti.
Le feature che non sono state implementate per questo progetto ma verranno sviluppate in un eventuale futuro sono:

- Diverse armi con diversi danni ad essi associati.
- Aggiunta della possibilità di avanzamento di livello con creazione casuale di arene tipo labirinto.
- Nuovi nemici diversi per ogni livello. (Eventuali boss finali)
- Nuovi collezionabili con effetti dannosi al personaggio principale.

Design

Architettura

Nel coordinamento dei componenti principali di cui è composta l'applicazione è stato utilizzato il pattern MVC. La classificazione in package rispecchia tale suddivisione.

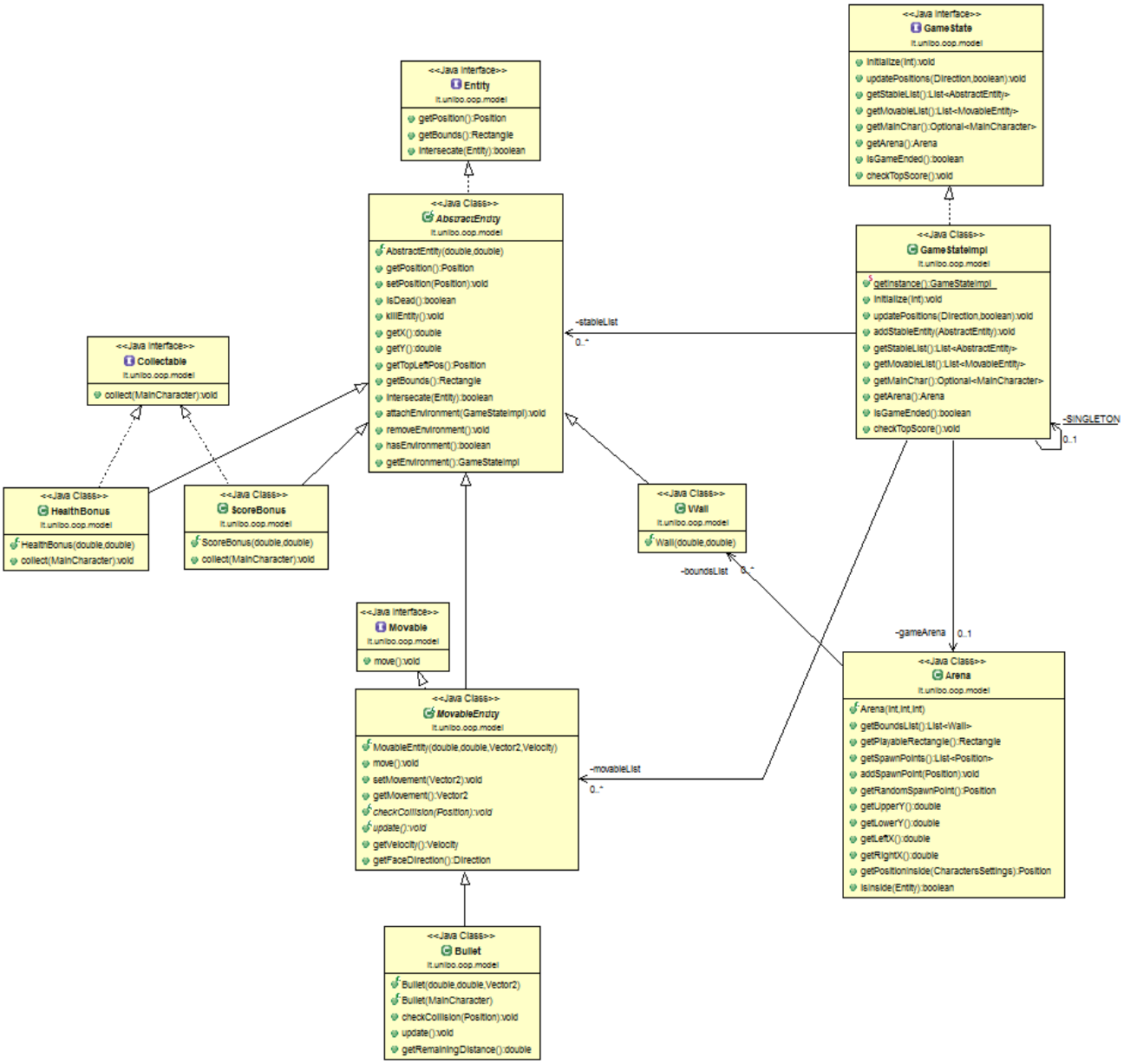


In tale suddivisione il controller acquisirà gli input dell'utente tramite la View e comanderà il model (GameState) il quale si aggiornerà di conseguenza. Successivamente il Controller comanderà alla View di eseguire il suo aggiornamento sulla base di come il GameState è cambiato. Il susseguirsi continuo di queste fasi costituisce il cosiddetto "gameloop", cuore dell'applicazione e del Controller. Le interfacce consentono un buon grado di disaccoppiamento, tale da garantire l’indipendenza delle componenti dall'implementazione delle altre.

Design dettagliato

Model (Matteo Minardi)

Base



Per quanto riguarda il Model praticamente la totalità degli elementi è modellata dalla classe *Entity* oppure composta da elementi di essa. *Entity* rappresenta una generale entità che ha al suo interno una posizione di riferimento a dove si trova e un rettangolo che delimita i bordi dell'elemento usato perlopiù per le collisioni. *Entity* ha una implementazione astratta generale chiamata *AbstractEntity* che racchiude tutte le funzionalità comuni alle entità stabili come: il GameState in cui si trovano, la gestione

delle posizioni e la creazione dei propri bounds con funzioni astratte. Per le collisioni infatti ogni entità avrà un rettangolo che lo racchiude al suo interno di dimensioni specifiche in base alla sua altezza e larghezza dichiarate in appositi metodi astratti. Questo tipo di implementazione mi ha permesso un vasto utilizzo del **Template Method** pattern che permette alle classi figlie di specificare informazioni a priori sconosciute al padre. Mi ha permesso di evitare riscritture di codice inutile rendendo il modello semplice da scrivere e intuitivo da capire. Le classi astratte hanno il compito di alleggerire il carico di codice delle classi figlie e racchiudere concetti comuni ai vari gruppi di entità.

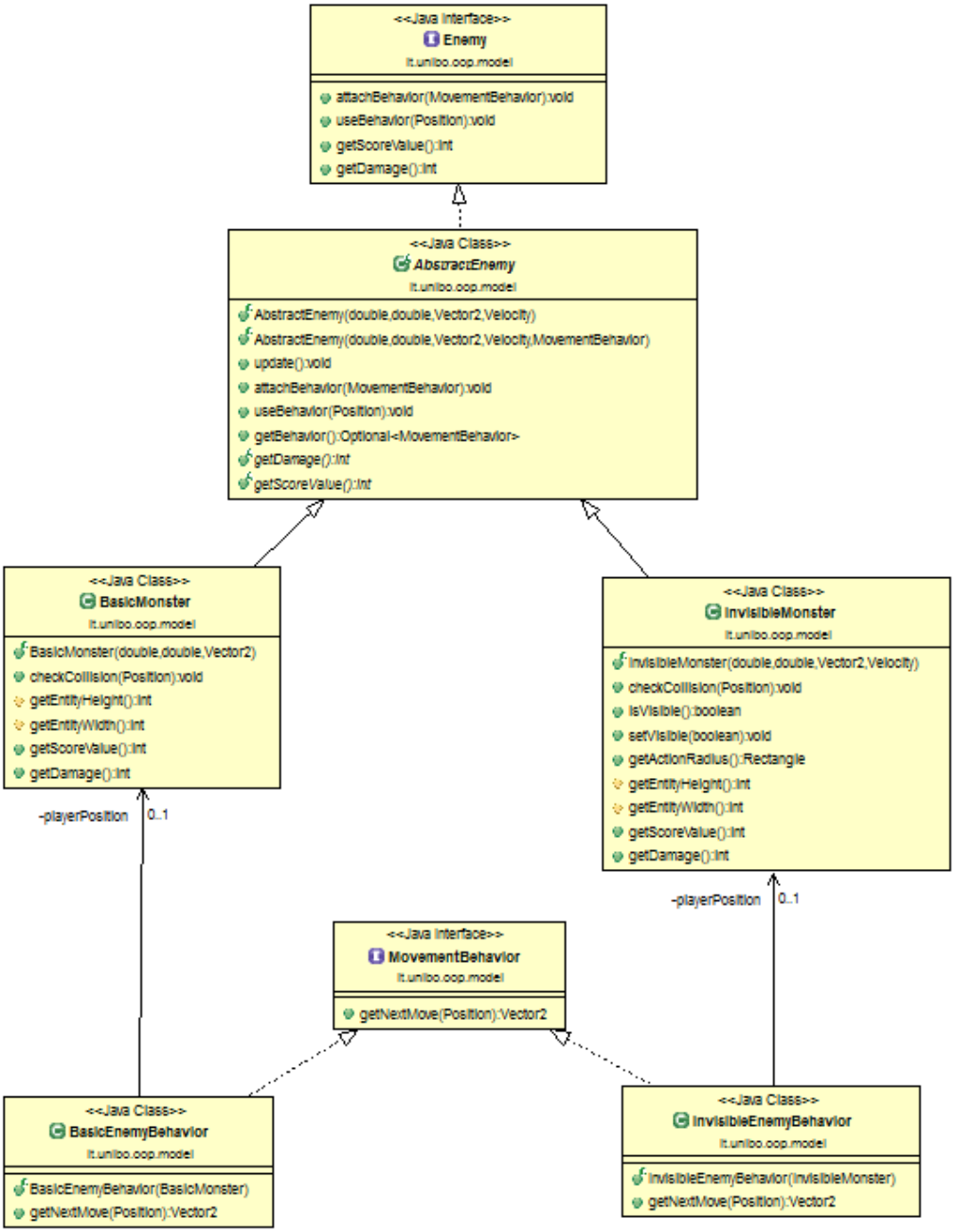
Una delle classi più importanti inoltre è il *GameStateImpl* che definisce l'ambiente di gioco dove avvengono tutti i movimenti, le collisioni e l'acquisizione di input importanti per l'andamento del gioco. Al suo interno contiene essenzialmente due liste:

- La lista di entità *AbstractEntity* (le stabili): che non vengono più mosse dopo la loro aggiunta nell'inizializzazione. Vengono però rimosse in caso di avvenimenti particolari come collisioni con oggetti collezionabili (*Collectable*). Di questa lista fanno parte anche le mura che però vengono create in una classe apposita (*Arena*) che ha il compito di generarla in modo opportuno tenendo conto di alcuni parametri fondamentali come la grandezza dell'area di gioco e la dimensione dell'HUD per l'utente.
- La lista di entità *MovableEntity*: vengono invece aggiornate durante il progredire del gioco con un metodo che permette di modificare il prossimo movimento e consente di controllare se sono avvenute collisioni particolari.

Questa struttura è unica all'interno del gioco e viene creata soltanto una volta quindi abbiamo effettuato la scelta di considerarlo come **Singleton** utilizzando un noto pattern che rispecchia i canoni delle nostre esigenze. Ogni entità avrà al suo interno un oggetto che rappresenta il *GameState* in cui si trova e si comporterà in conseguenza allo stato di esso. Tutte le collisioni e i movimenti saranno la conseguenza dell'ambiente di gioco corrente, ogni oggetto infatti, prima di muoversi, controlla se causa qualche collisione con altri elementi delle liste sopra citate e in quel caso, a seconda di cosa è avvenuto, esegue una ben specifica azione accuratamente scelta.

Come detto prima, la lista Movable rappresenta una enorme fetta delle entità coinvolte tra le quali: MainCharacter, AbstractEnemy e Bullet che verranno descritte sotto le prossime immagini.

Nemici



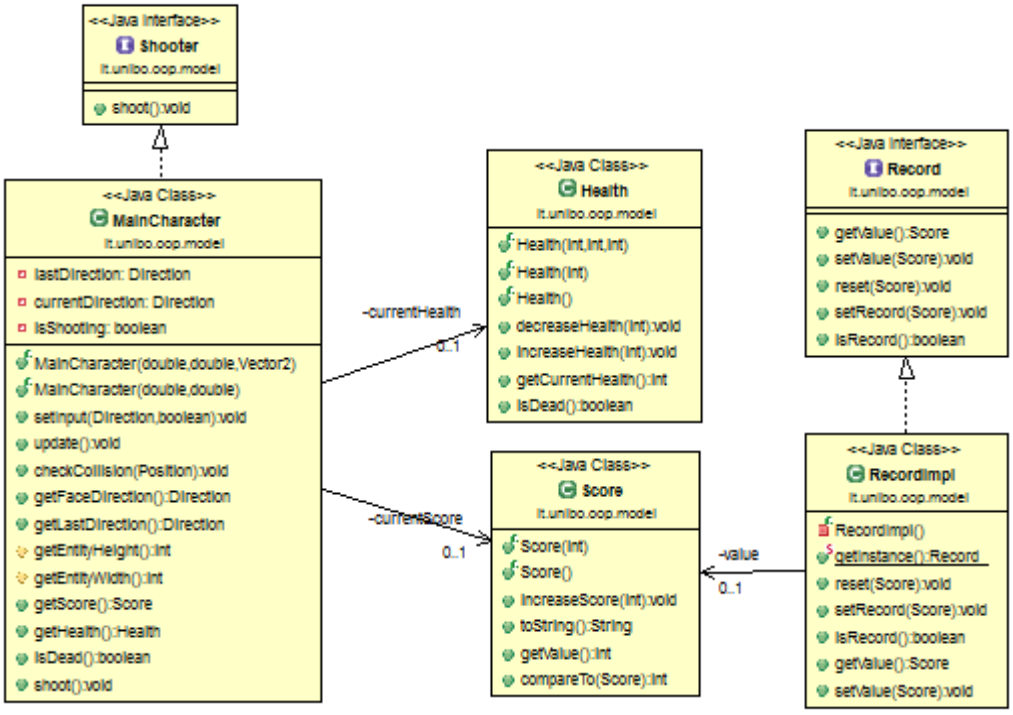
In breve, i nemici, estendono *MovableEntity* e al loro interno hanno un comportamento ben specifico che detta loro ciò che devono compiere al prossimo movimento. Questo comportamento, detto *MovementBehavior* consente loro di calcolare il prossimo vettore che andrà sommato alla posizione corrente per ottenere così la posizione in cui si troverà nell'aggiornamento successivo. Questo pattern, detto **Strategy,** incapsula all'interno di un oggetto un particolare comportamento basato su semplici algoritmi di intelligenza

artificiale. Sono stati implementati solamente due mostri, senza una vera e propria salute e con due diversi comportamenti associati.

Il primo (*BasicMonster*) ha un movimento lento e ha un comportamento lineare che continua a seguire il personaggio principale ovunque esso vada. Semplicemente evita di collidere con altri nemici fermandosi se vede che crea una collisione e viene sconfitto se un *Bullet* lo raggiunge o si trova ad affrontare il personaggio principale in un corpo a corpo.

Il secondo (*InvisibleMonster*) è un mostro molto veloce che fa molti danni in caso di collisione e non si vede finché non entri nel suo raggio d'azione. Il suo comportamento analizza la posizione del personaggio principale, decide se rivelarlo o no e soprattutto se condurlo addosso a lui o alla sua tana iniziale. È possibile vederlo solo se si è all'interno del raggio di visione altrimenti non viene visto. Quando non vede nessun personaggio principale il comportamento lo riconduce alla posizione iniziale.

Personaggio Principale



Il personaggio principale (*MainCharacter*) è considerato come una MovableEntity che ha la capacità di sparare (*Shooter*) e ha alcune feature che lo caratterizzano come la

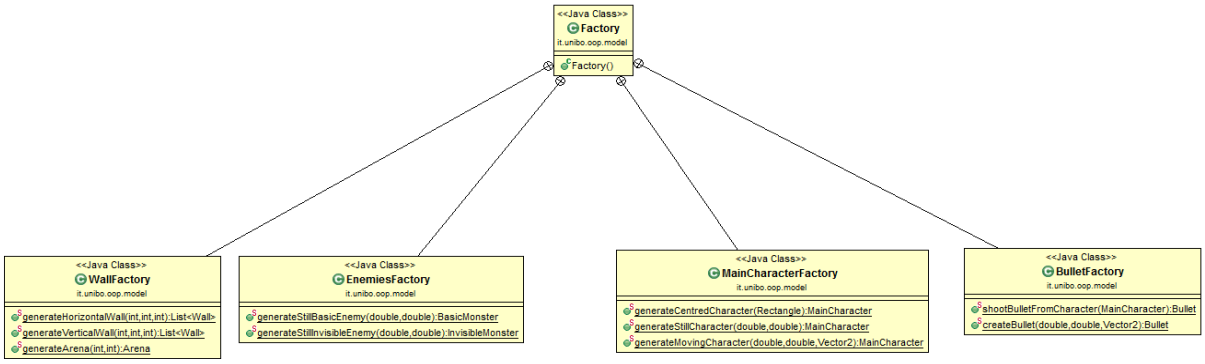
salute (*Health*) e il punteggio (*Score*).

Il personaggio principale oltre ad essere più forte e più veloce dei nemici è in grado di sparare *Bullet* diretti nella direzione in cui si sta andando. Le collisioni vengono gestite in modo che non possa oltrepassare il muro esterno definito da *Arena* e che se raccolga un *Collectable* gli venga attribuita la ricompensa. I mostri cercano di assaltarlo togliendogli vita in caso ci riescano e una volta terminata la vita contenuta in *Health* il gioco finisce con una sconfitta.

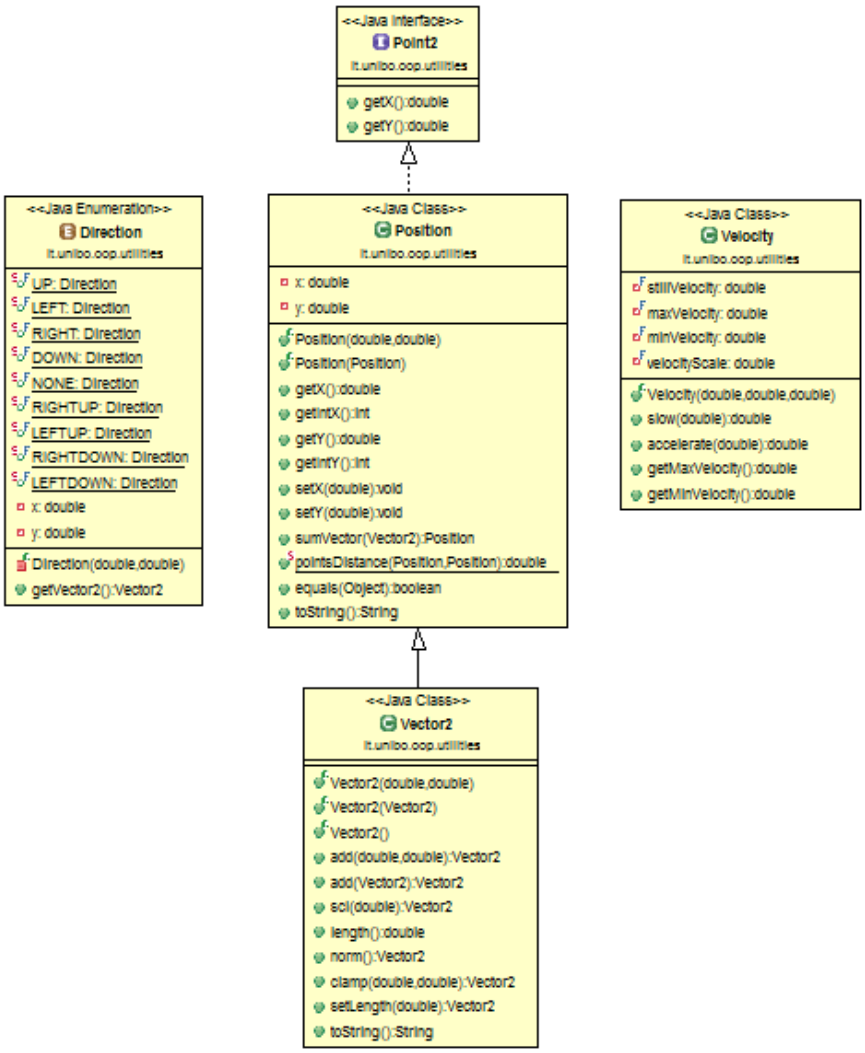
Per facilitare la gestione del record da parte delle principali componenti del pattern MVC, si è deciso di utilizzare una classe ad hoc RecordImpl che, tramite implementazione del pattern **Singleton**, può essere acceduta in totale libertà dalle suddette parti. Tale classe wrappa uno *Score* serializzabile che potrà poi essere salvato su file dal Controller, Stampato a schermo dalla View e aggiornato dal Model.

Factory e Utilities

Infine in queste ultime immagini sono presenti le strutture dati alla base delle entità utili per modellare le funzionalità necessarie.



Per prima cosa viene usato il pattern **Factory** che serve per la creazione di nuovi oggetti in modo intuitivo senza il bisogno di ricorrere a costruttori poco chiari e con diversi parametri.



Le strutture Position e Vector caratterizzano rispettivamente la posizione e il vettore movimento scalato opportunamente dalla Velocity di ogni Entità.

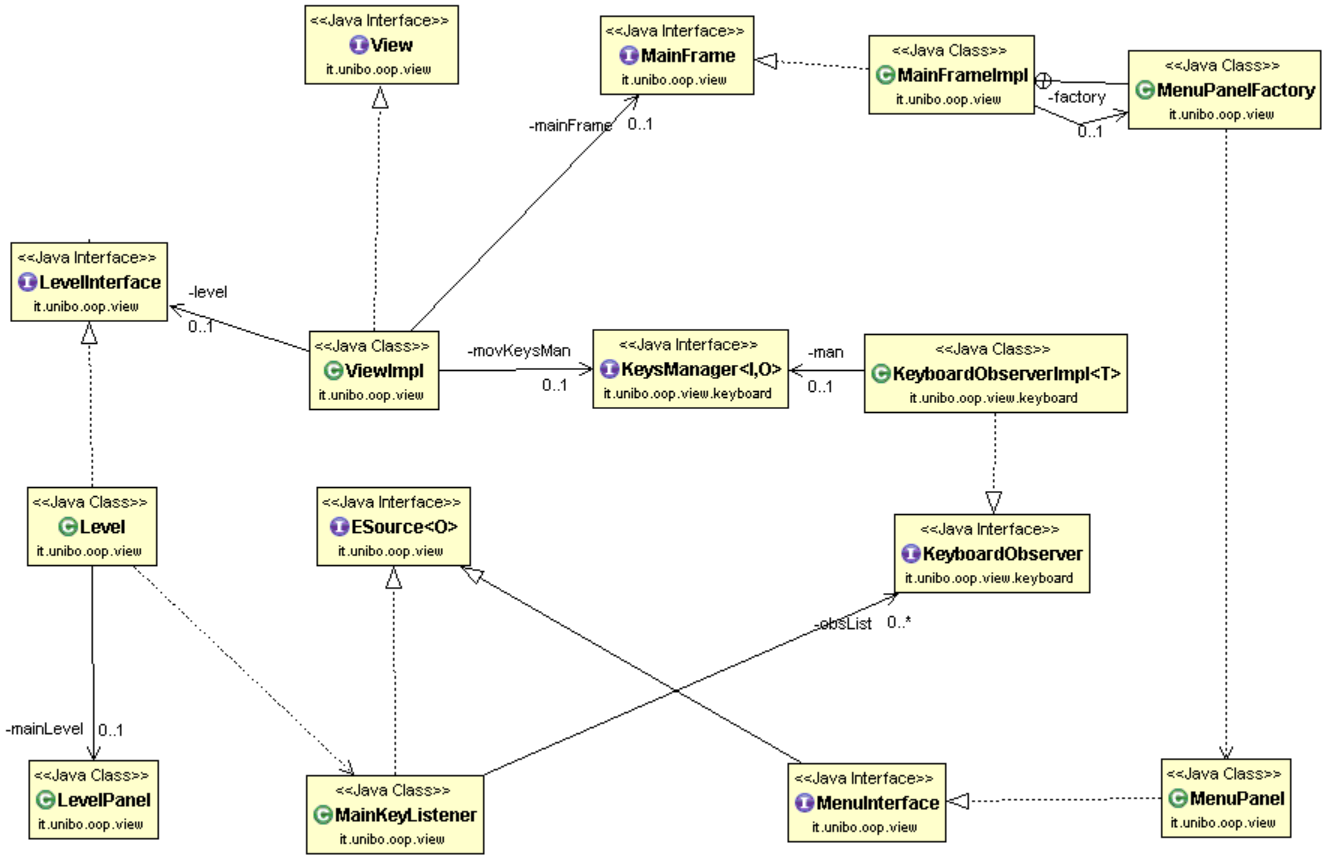
È stato inserito tra le utilities anche un semplice player per gestire l'audio del gioco, con funzioni per riprodurre (sia una sola volta che in loop) e stoppare le musiche di sottofondo per menù e livelli. Il player aderisce al pattern singleton.

Questo breve riassunto è stato appositamente diviso in 4 aree semantiche diverse strettamente collegate tra loro da relazioni. Il design realizzato è il risultato di una analisi passo passo del modello teorizzato. Questo capitolo ha lo scopo di illustrare ad un certo livello di astrazione (né troppo specifico, né troppo superficiale) il dettaglio dell'implementazione o comunque i ragionamenti effettuati per arrivare al codice

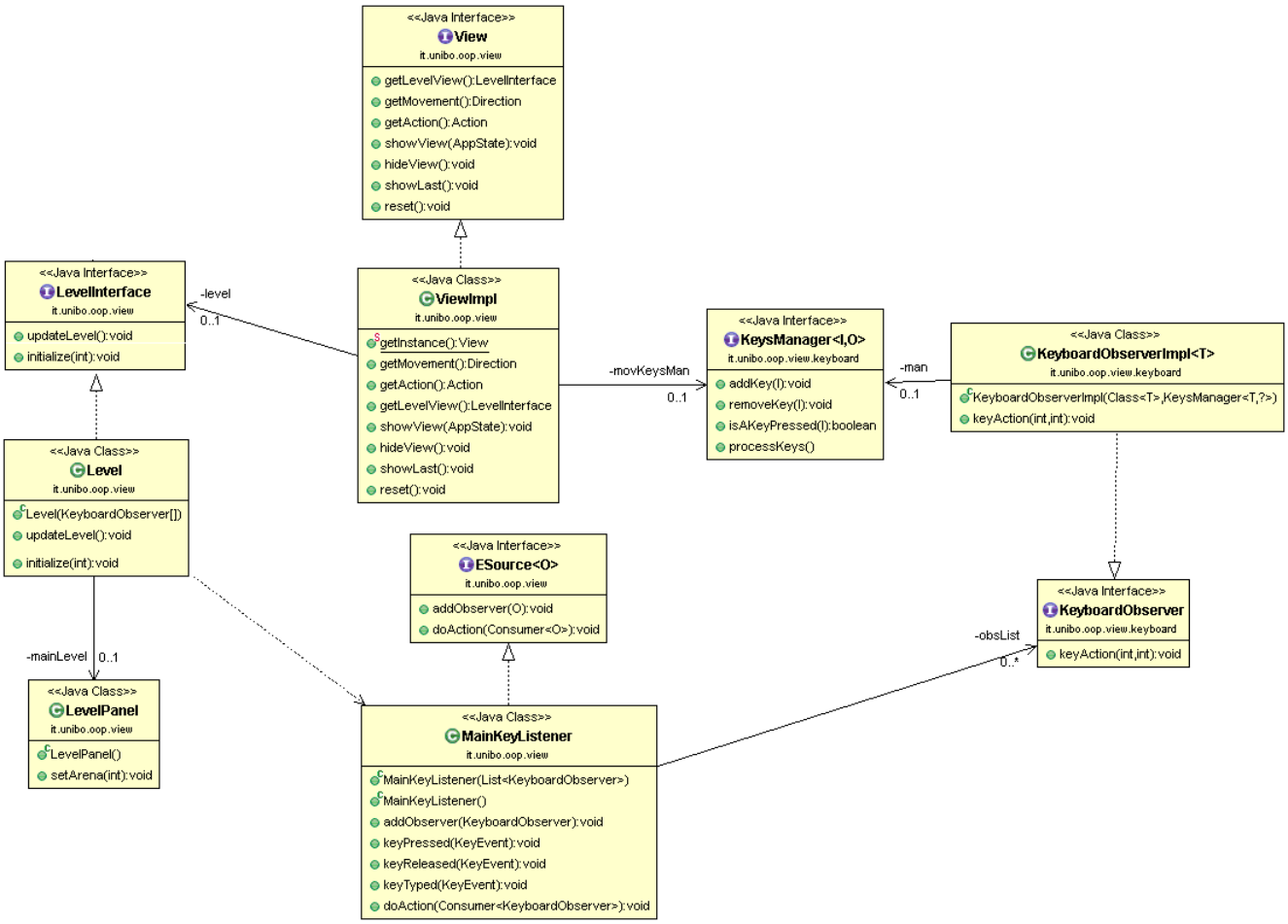
riguardante il Model. Si è fatto ampio utilizzo di pattern ben noti per rendere il codice ordinato e intuitivo.

View (Giacomo Pasini, Paolo Venturi)

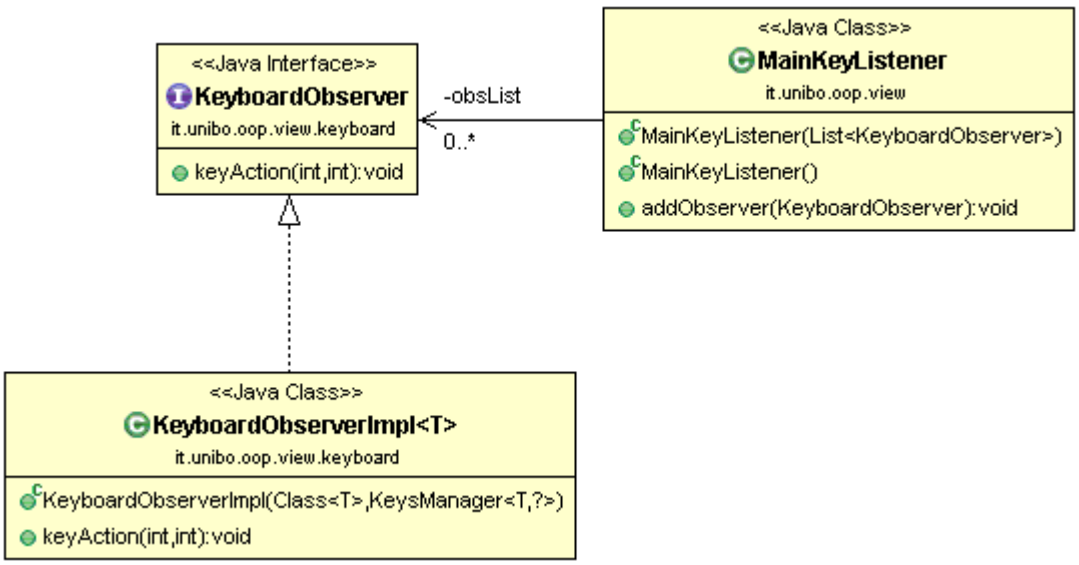
La parte relativa alla View è stata divisa in due core differenti: la gestione della scena di gioco (Level) e la gestione dei menu (MainFrame).



Level



Qui si trova la gestione della grafica del gioco, nonché della gestione degli eventi da tastiera. Difatti, al Level è associato un KeyListener custom in grado di notificare più Observer per la tastiera. Tramite questo vengono catturati ed inviati a due KeysManager (classi addette alla gestione e manipolazione degli eventi da tastiera) gli eventi generati durante il gioco e, tramite appositi metodi della View, rese disponibili all'esterno la direzione e l'azione risultanti dalla conversione degli "eventi meccanici" (pressione del tasto) in "eventi logici" (azione da compiere sul livello di gioco). Trai i pattern utilizzati e visibili dall'UML di cui sopra, vi è il pattern **strategy** nell'interfaccia ESource, utilizzato anche in altre parti del progetto e il pattern **Observer**.



All'interno della finestra principale viene inserito il LevelPanel, dove viene disegnata la parte grafica principale del gioco; questo viene aggiornato con la frequenza scelta nel GameLoop. In esso vengono disegnati tutti gli elementi grafici, ovvero il background dell'arena, le sprite del personaggio e dei vari nemici, le apparizioni dei collezionabili, le icone e le statistiche. Ad ogni nuovo frame il controller chiama il metodo di repaint del pannello e questo viene ridisegnato apportando le varie modifiche, ovvero i cambiamenti di posizione e l'aggiornamento delle statistiche. Il background dell'arena viene scelto casualmente all'inizio del gioco dal controller tra quelli che sono stati caricati durante la costruzione del pannello.

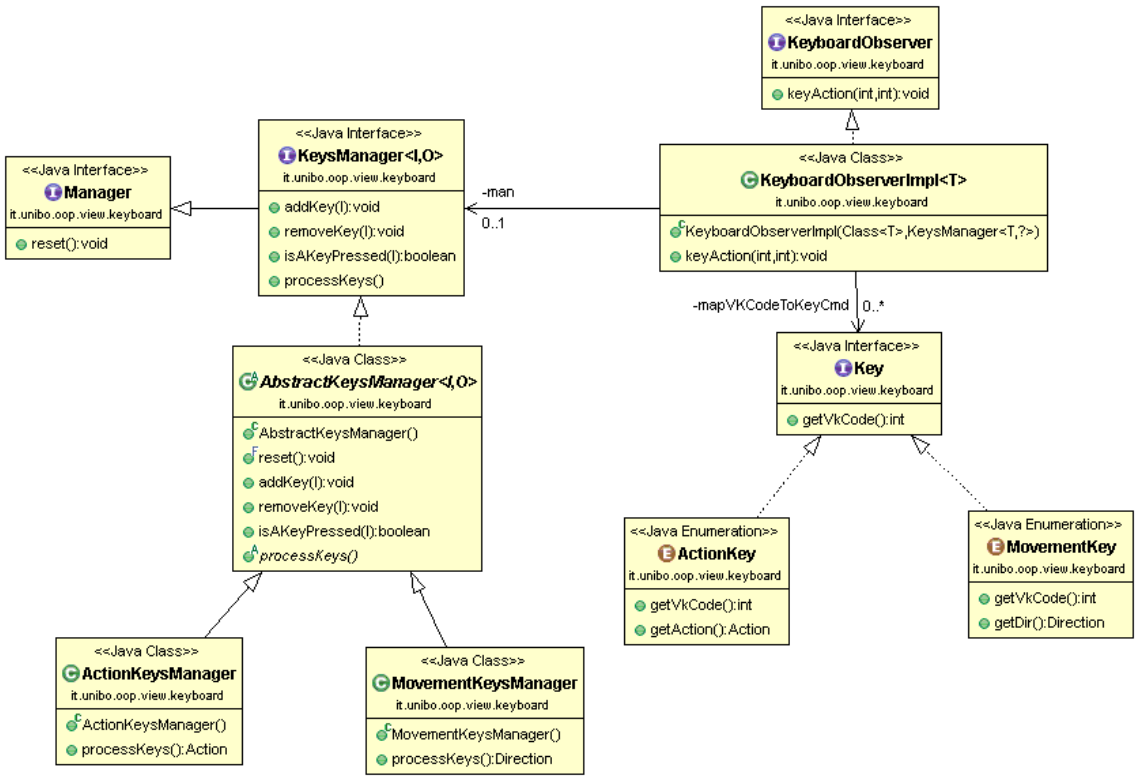
Per rendere automatica l'aggiunta e la modifica delle sprite dei personaggi si è ritenuto comodo creare una classe SpriteSheet che si occupa di caricare gli sprite sheet (ovvero le immagini che contengono ognuna le varie posizioni di un set di sprite), che vengono generati utilizzando un sito apposito. Un metodo della classe si occupa poi di tagliare di conseguenza l'immagine per ottenere ogni singola sprite, che viene associata alla direzione in cui è rivolta (UP, DOWN, LEFT, RIGHT), per essere successivamente disegnata di conseguenza ad ogni aggiornamento della schermata in base all'informazione sulla posizione contenuta nel model.

Focus sulla gestione della keyboard

Al fine di migliorare l'organizzazione delle classi e la leggibilità si è deciso di racchiudere le classi relative alla gestione della tastiera in un sottopackage apposito della view.

Per quanto riguarda la gestione dei tasti e la loro mappatura su specifiche azioni del

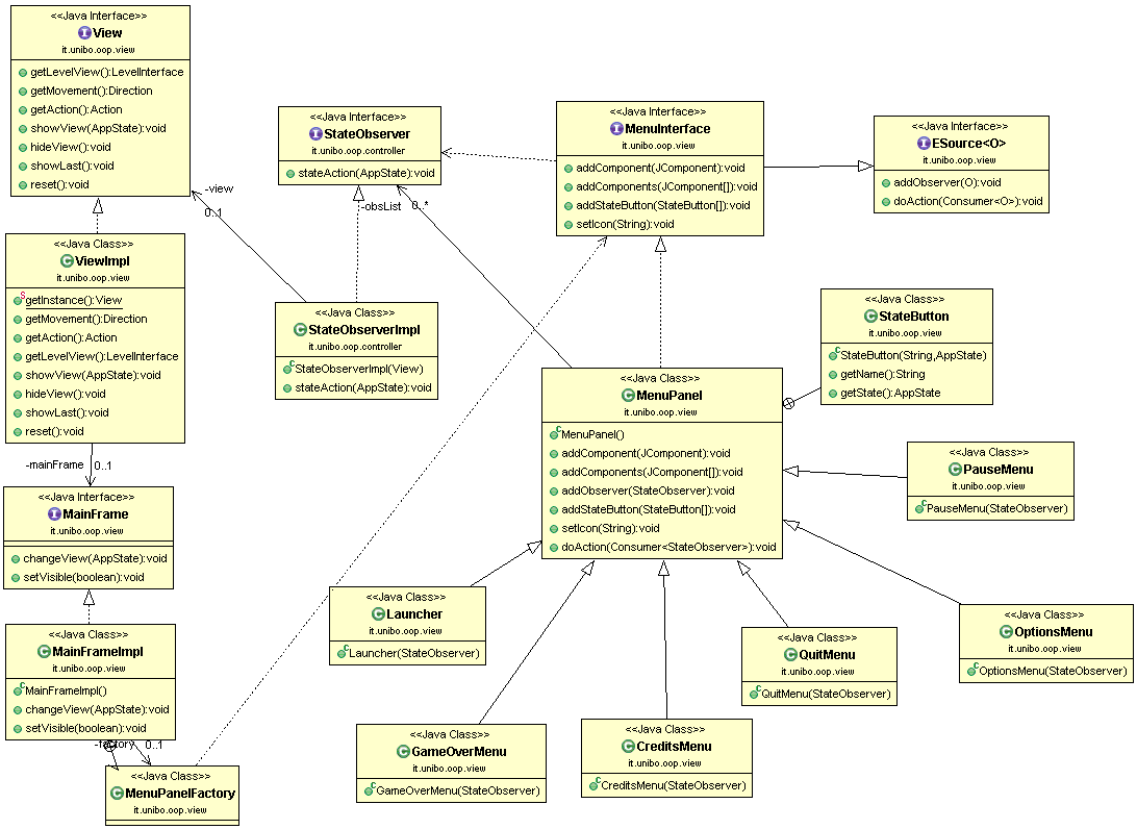
gioco, si è deciso di suddividere in due manager appositi le due categorie individuate: i movimenti di base del personaggio e le azioni che questo può compiere. Ció deriva principalmente dal fatto che mentre il primo difficilmente subirá modifiche, dato che gestisce giá una rosa di ben 8 possibili direzioni di movimento (piú che sufficienti per questo genere di applicazioni), il secondo, al contrario, é piu suscettibile di modifiche e ampliamenti in quanto, per ora, gestisce solamente due azioni (sparare e pausa); con una separazione simile, se si volessero aggiungere nuove azioni lo si potrá fare senza minimamente toccare il primo. Per garantire l'integrazione di future feature, Il set di eventi meccanico/logici gestito è stabilito a priori in apposite enumerazioni modificabili e ampliabili liberamente, senza bisogno di modificare la struttura delle classi di gestione. É quindi possibile cambiare la mappatura delle keys logiche e meccaniche semplicemente modificandone il valore nelle suddette enumerazioni. Infine, come da UML, ci si é avvalsi di una classe astratta per fattorizzare il codice comune ed evitare inutili copia-incolla. Entrambi i gestori della tastiera sono resi disponibili al controller per mezzo della entry-view.



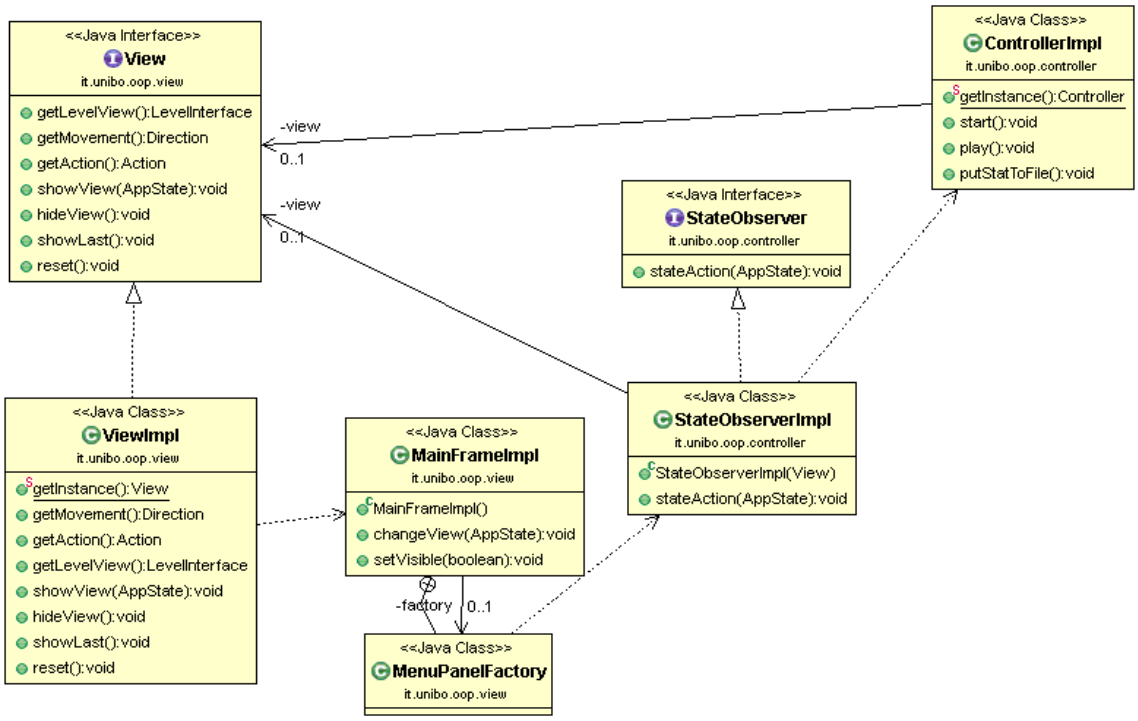
Gestione dei menu (MainFrame)

Vista la natura dell'applicazione e il gran numero di stati (menu e fasi di gioco) individuati, si è deciso di utilizzare un unico frame (MainFrame) che, al cambio di stato dell'applicazione, modifica il proprio pannello principale. Per facilitare questa transizione e la creazione/sostituzione dei vari pannelli si è deciso di utilizzare una inner

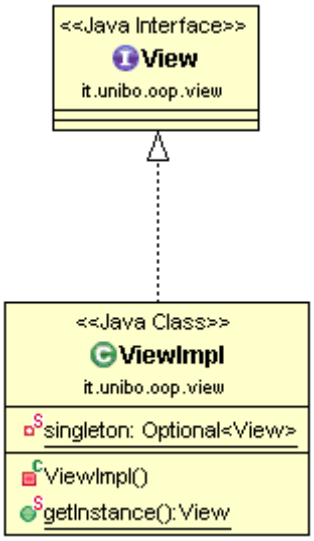
class come **simple factory**. Ciò consente, oltretutto, l'aggiunta di nuovi pannelli/stati senza dover toccare altre classi della View. Sotto si può notare come il codice comune dei vari menu, nonché la customizzazione dei vari componenti dei menu, sia fattorizzato nella classe padre MenuPanel.



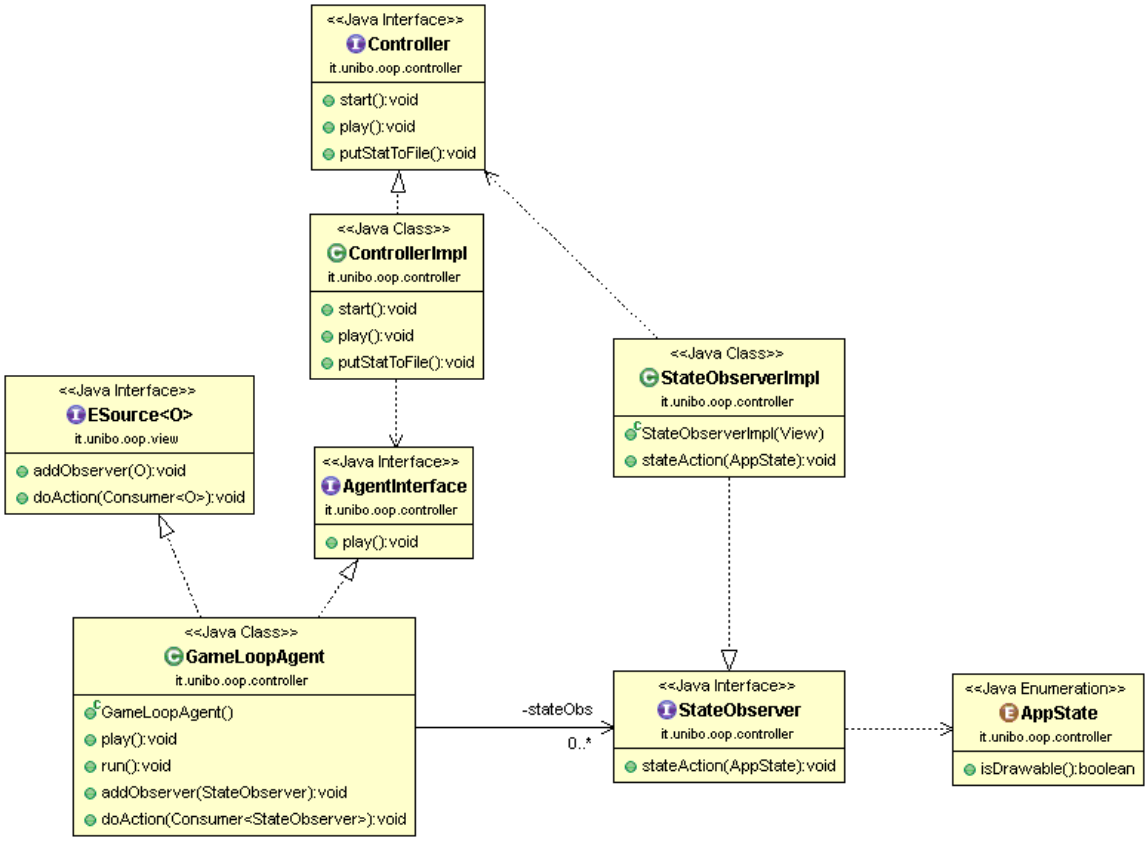
La sostituzione del pannello avviene sempre tramite la mediazione di un observer dedicato (StateObserver) che fa da tramite tra tutti i buttons per il passaggio di stato (StateButton) e la View, la quale comanderà poi il MainFrame di cambiare pannello in relazione allo stato comunicato all'observer. Oltre a questo, l'observer consente anche l'invio di comandi da View a Controller e viceversa. NB: ulteriori dettagli sul ***pattern observer*** utilizzato e sull'interazione tra Controller e View sono trattati più nel dettaglio nelle sezioni dedicate successivamente.



Infine, occorre notare come anche la View implementi il ***pattern singleton***, per un accesso più "semplice e libero" da parte delle classi utilizzatrici.



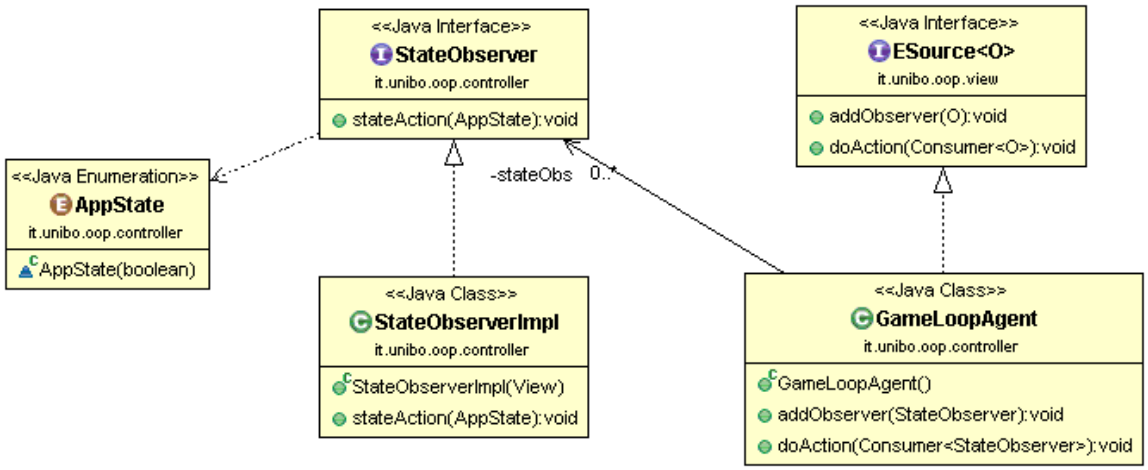
Controller (Paolo Venturi)



Al suo interno, il Controller è formato da alcune sottoclassi atte a gestire le varie fasi di vita dell'applicazione. Mentre ControllerImpl gestisce le inizializzazioni varie, Il vero cuore del Controller, il cosiddetto "game loop", è implementato nella classe GameLoopAgent. Questa ha come scopo quello di alternare in modo continuo la scansione degli input dalla View con le notifiche al Model e il conseguente ri-aggiornamento della stessa. Per semplificare la gestione delle notifiche alla View, nonché il susseguirsi delle diverse fasi dell'applicazione, si è utilizzato un Observer apposito che gestisse il passaggio da uno stato (AppState) ad un altro. Tramite questo è possibile comunicare sia al Controller che alla View in quale stato si dovrà migrare, e, relativamente a quest'ultima, quale menu dovrà essere mostrato.

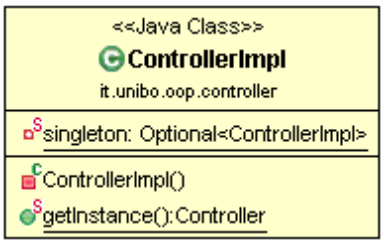
Come appena detto, e come si può notare nell'UML, è stato quindi utilizzato il pattern **Observer** per notificare il Controller dello stato in cui passare (che viene modificato in risposta ad eventi della View) e un pattern **Strategy** per il metodo doAction dell'interfaccia ESource<O> che, accettando come argomento un Consumer<O>,

consente di poter utilizzare, in altri parti del progetto, pattern observer analoghi ma che differiscono solo per il tipo/numero degli argomenti che richiedono. In sostanza è una generalizzazione del metodo *notifyObservers(Object)* visto a lezione.



L'utilizzo dell'observer e della enumerazione associata (AppState), contenente gli stati in cui l'applicazione può trovarsi, consentono un'integrazione semplice di future feature (come ad esempio l'aggiunta di nuovi menu o nuovi stati di gioco) alla parte di controllo senza dover stravolgere la struttura di questa.

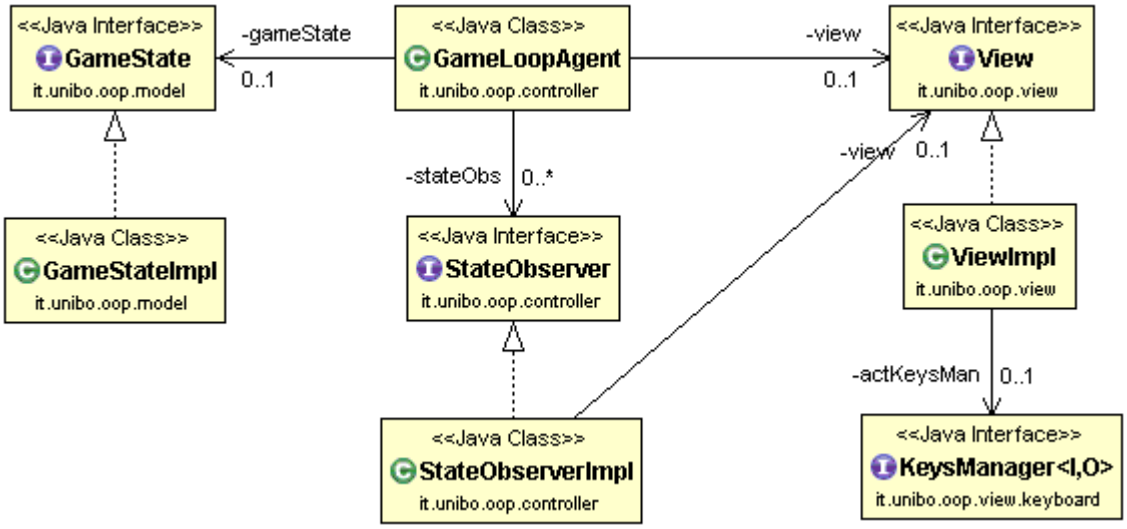
Infine, la classe ControllerImpl implementa il pattern **Singleton** in modo da consentire alle classi clienti un'interazione più semplice e libera dai vincoli di passaggio ai vari costruttori di queste del riferimento alla classe di controllo, analogamente a quanto avviene per Model e View.



Interazioni tra Controller e View/Model

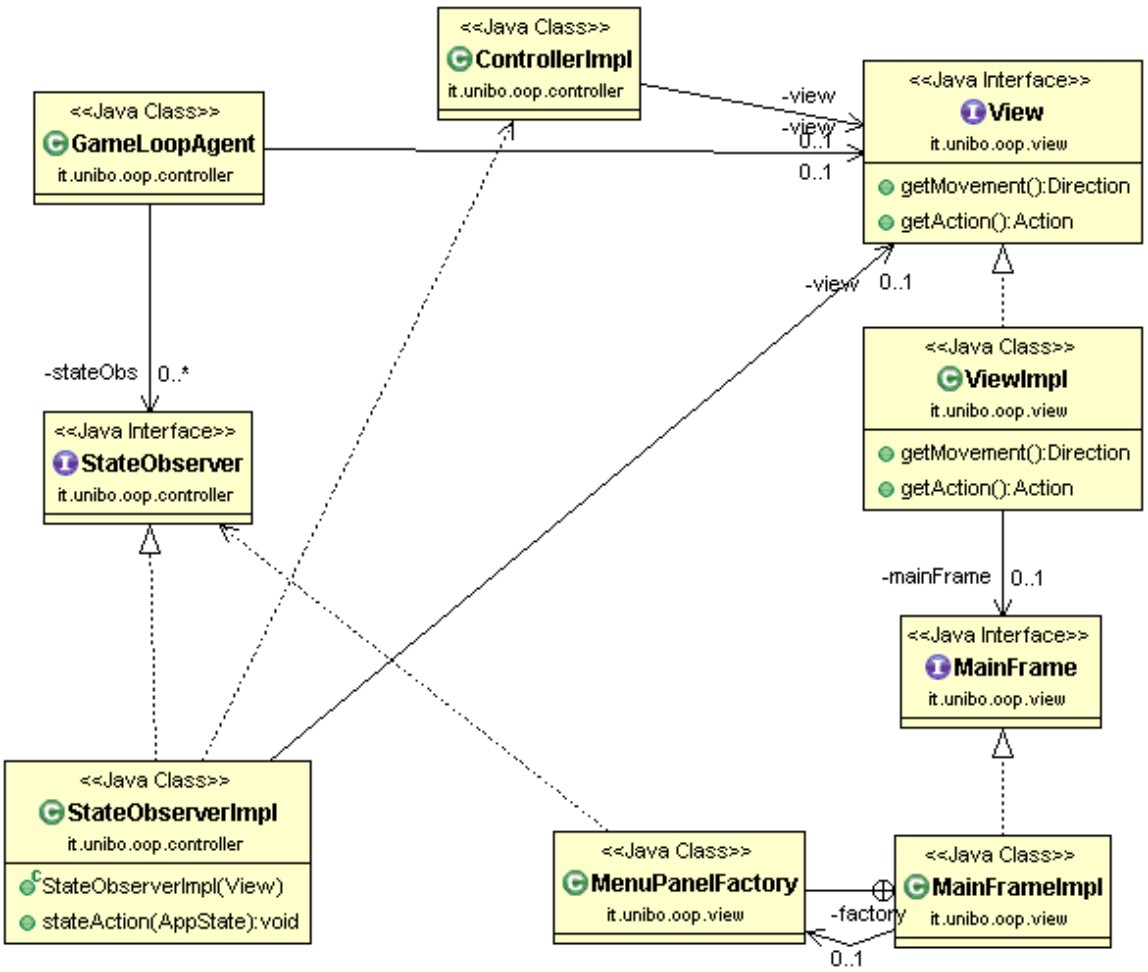
Per quanto riguarda le relazioni con la View e la gestione di questa, il controller ha a che fare con due tipologie di eventi: gli eventi da tastiera, generati durante il gaming vero e proprio, e gli eventi dai menu di gioco.

- Eventi da tastiera:



A questa tipologia di eventi corrispondono principalmente le informazioni da passare al Model e che determinano lo stato del personaggio (come movimento e azione), ed, eccezionalmente, comandi di "passaggio di stato" p.e. lo stato di pausa dell'applicazione. Per entrambi i casi, comunque, come si può vedere nell'UML di cui sopra, la gestione avviene tramite un accesso diretto da parte della classe implementante il gameloop alla View per ottenere le suddette informazioni. Una volta ottenute, se sono informazioni esclusive per il model allora vendono direttamente passate a questo, viceversa, se sono informazioni relative allo stato del gioco (p.e. game-pause) queste vengono inviate all'observer, cui spetterà il compito di notificare del cambiamento di stato i relativi clienti. Da notare come lo stesso observer sia utilizzato comodamente anche per il passaggio allo stato di "game over".

- Eventi da menu:



Per ciò che concerne questo secondo tipo di eventi, che non interessano direttamente il gameloop, la View (o meglio i menu di cui si compone la View e generati dalla relativa factory) accedono prima all'observer, conseguentemente, a seconda dell'evento, si ha un passaggio di stato relativo alla sola View (cambio di menu), oppure un cambio di stato che coinvolge anche il Controller (p.e. i comandi play, replay).

Come da UML, che mostra in generale le sopra citate interazioni tra le principali classi di C e V, un'ultima tipologia di interazione che si può avere è tra la entry class del Controller e la View. Questa interazione avviene esclusivamente nelle fasi di re/inizializzazione della View e nelle fasi di gioco, realizzando di fatto una divisione delle responsabilità di gestione della stessa da parte delle due principali classi che ne regolano il controllo (ControllerImpl e relativo Agent).

Sviluppo

Testing automatizzato

Nel nostro progetto il testing automatizzato è stato utilizzato principalmente per le funzioni basilari del Model tipo: posizioni, vettori, rettangoli di bounds e poche altre caratteristiche alla base di tutto il funzionamento del progetto. Un minimale test all'interno di qualsiasi progetto ti fa capire cosa non va e cosa rispecchia le aspettative. Quindi è importante costruirlo e saperlo costruire secondo le norme JUnit.

Suddivisione del lavoro

Paolo Venturi

La parte di principale interesse è stata quella relativa al Controller e alla gestione degli eventi tra questo e la view. Complessivamente sono state dedicate a tale parte circa 4/5 ore al giorno di progettazione, studio, e sviluppo per un totale di 3 settimane. Purtroppo a causa di fraintendimenti ed errori in fase di progettazione e ripartizione dei ruoli, ha avuto luogo un sovradimensionamento del mio ruolo che ha portato ad occuparmi di una buona percentuale della view. Ciò nonostante abbiamo fatto uso massiccio del DVCS e ci siamo coordinati al fine di riuscire ad ottenere una demo funzionante entro i termini previsti. Per ciò che concerne l'integrazione delle parti, si è deciso in comune quali metodi sarebbero stati i ponti tra le varie parti del progetto. Come da mail, la mia parte ha interessato progettazione e sviluppo di:

- Controller;
- Organizzazione e gestione della view e dei menu, degli eventi (Keyboard) e dell'interazione tra controller e view

Ho inoltre contribuito in piccola parte nello sviluppo di alcuni elementi del Model, relativi alla gestione del Record del giocatore.

Matteo Minardi

Il mio ruolo consisteva nello studio e nella realizzazione del modello strutturale del programma. Sono stato aiutato e ho ricevuto consigli dal resto del mio gruppo ma principalmente la parte progettuale è stata svolta in autonomia con l'aiuto di vecchi progetti da cui ho preso spunto e ho fatto riferimento per raccogliere le idee su come ragionare e come impostare il tutto essendo alle prime esperienze in questo campo.

Ho cominciato a ragionare sul lavoro appena il progetto è stato approvato e ho gettato le basi per la struttura. Per quanto riguarda l'aspetto implementativo ho dedicato un mese con una quantità di minimo 4 ore giornaliere al progetto con relativi giorni di pausa.

Giacomo Pasini

Il mio ruolo principale consisteva nello sviluppo dei contenuti visivi del gioco, ovvero la gestione della schermata di gioco principale, di tutte le immagini e della loro rappresentazione su schermo. Ho anche curato il suono sviluppando il riproduttore musicale. Ho cominciato a documentarmi su come programmare gli aspetti grafici appena il progetto è stato approvato: lo studio della gestione di immagini e audio ha costituito infatti la maggior parte del tempo da me impiegato nel progetto, che si aggira in media sulle 2 ore al giorno per circa un mese, con qualche giorno di pausa. Come detto da Venturi, purtroppo la sua gestione dei restanti aspetti della view è finita col risultare un lavoro un po' più grande di quanto si pensasse.

Commenti finali

Autovalutazione e lavori futuri

Matteo Minardi:

Sono particolarmente soddisfatto dal modello strutturale ottenuto, anche se opinabile sotto qualche aspetto, ho pienamente superato le aspettative nei miei confronti sia a livello di sviluppo sia a livello collaborativo come team di sviluppo. Ho dedicato parecchie ore alla progettazione su carta e alla discussione delle varie strade da intraprendere e di tutte le possibilità implementative scegliendo quella, secondo me, migliore. Ho cercato di scrivere del buon codice, cercando di evitare ridondanze e copia incolla. Una grande pecca del nostro lavoro è stata la suddivisione dei ruoli e del bilanciamento degli incarichi. Essendo il nostro primo progetto credo rientri nella normalità il formarsi di una certa disomogeneità a livello di conoscenze e a livello di lavoro all'interno del gruppo. Il goal finale è stato raggiunto senza nessuna intenzione di strafare o realizzare un gioco complicato. Sono contento che l'idea del gioco abbia coinvolto me e i miei colleghi motivandoci e dandoci infine una piccola soddisfazione.
Per il futuro penso sia meglio abbandonare questo progetto e dedicarsi con più calma a qualcosa di più elaborato e completo con l'utilizzo di librerie dedicate e magari in altri linguaggi.

Paolo Venturi:

Sono particolarmente soddisfatto del risultato ottenuto, anche se avrei preferito una più equa ripartizione dei ruoli. Ciò nonostante ho avuto la possibilità di cimentarmi in aspetti di progettazione, sviluppo e studio di alcuni degli elementi centrali in un'applicazione del genere e che mi hanno sicuramente interessato e soddisfatto. La buona collaborazione e cooperazione col resto del team ha sicuramente aiutato nello sviluppo e nel raggiungimento degli obiettivi che ci eravamo prefissati. La principale speranza per il futuro è un miglioramento nella stesura del codice e nella progettazione. Spero, infatti, che il codice prodotto e la progettazione delle parti siano soddisfacenti e relativamente a questo avrei preferito una migliore esperienza.

Sarebbe bello, per il futuro, utilizzare questo progetto come punto di riferimento per lo sviluppo di ulteriori applicazioni di genere analogo o migliorare direttamente questo. Ovviamente, essendo solo all'inizio, sono ansioso di mettermi alla prova e sperimentare anche ulteriori campi dell'informatica.

Giacomo Pasini:

Sono soddisfatto del risultato ottenuto, soprattutto dato che questo progetto mi ha dato modo di imparare molti aspetti del linguaggio (immagini e suoni) che non avevo ancora mai considerato, oltre a tutti gli aspetti di progettazione e sviluppo che si possono imparare solo lavorando in gruppo. Mi ritengo soddisfatto anche dell'interazione con gli altri membri del progetto, che è stata semplificata soprattutto dall'uso massiccio di DVCS.

Vorrei riuscire a migliorare ulteriormente questo progetto sotto ogni aspetto, magari rendendolo più complesso e completo. Sono sicuro che questo sia comunque un ottimo punto di partenza verso tanti altri progetti futuri, da usare come base per comprendere e risolvere eventuali errori ma soprattutto per quanto riguarda la progettazione e la coordinazione di gruppo.

Guida utente

All’avvio dell’applicazione viene visualizzato il launcher iniziale contente le possibili scelte effettuabili dall’utente:

- Play: questa funzione consente di entrare direttamente nel vivo del gioco creando un livello casuale con all’interno un personaggio e una serie di mostri da sconfiggere per completare l’obiettivo.
- Options: consente di gestire alcune funzionalità aggiuntive come l’attivazione della musica, guardare i crediti e ripristinare i record personali.
- Quit: permette all’utente di chiudere l’applicazione in modo corretto.

Una volta entrati nel gioco è possibile muoversi all’interno dell’arena con i tasti WASD e sparare proiettili con la Barra Spaziatrice. Il gioco sarà terminato quando l’utente termina la vita o sconfigge tutti i mostri presenti nell’arena e a quel punto sarà possibile scegliere se uscire o giocare un'altra partita. È presente il menu di pausa accessibile in qualsiasi momento del gioco premendo il tasto Esc in cui è possibile riprendere quando si desidera, ricominciare il livello, gestire le opzioni o uscire definitivamente.

Fonti

http://stackoverflow.com/

http://obviam.net/index.php/the-mvc-pattern-tutorial-building-games/

https://it.wikipedia.org/wiki/Pagina_principale

https://www.codeandweb.com/what-is-a-sprite-sheet

https://bitbucket.org/danysk/oop14-giunchi-gabriele-mazzesi-lorenzo-ricci-mattia-minigore