Relazione progetto

Edillion

31-05-2016

Montini Leonardo

Tonucci Riccardo

Zamagni Enrico

Sommario

Sommario 2
Analisi 3
1.1 Requisiti 3
1.2 Analisi e modello del dominio 3
Design 5
2.1 Architettura 5
2.2 Design Dettagliato 5
Model – Montini Leonardo 5
View – Tonucci Riccardo 11
Controller – Zamagni Enrico 14
Sviluppo 17
3.1 Testing Automatizzato 17
3.2 Metodologia di lavoro 18
Commenti Finali 18
4.1 Autovalutazione 18
Guida Utente 20
Menu principale 20
Creazione dell'Hero 20
Selezione dello Stage 20
Schermata di combattimento 21
Caricamento salvataggio 21

Analisi

1.1 Requisiti

Il software mira alla realizzazione di un videogioco in stile RPG a tema medievale/fantasy.

Edillion è un gioco di ruolo con sistema di combattimento a turni tipico degli RPG[1] anni '90 (Final Fantasy[2], Dragon Quest[3]). Il giocatore dovrà combattere tramite scelta delle mosse e dei nemici da attaccare tramite interfaccia grafica fino al raggiungimento della vittoria o sconfitta.

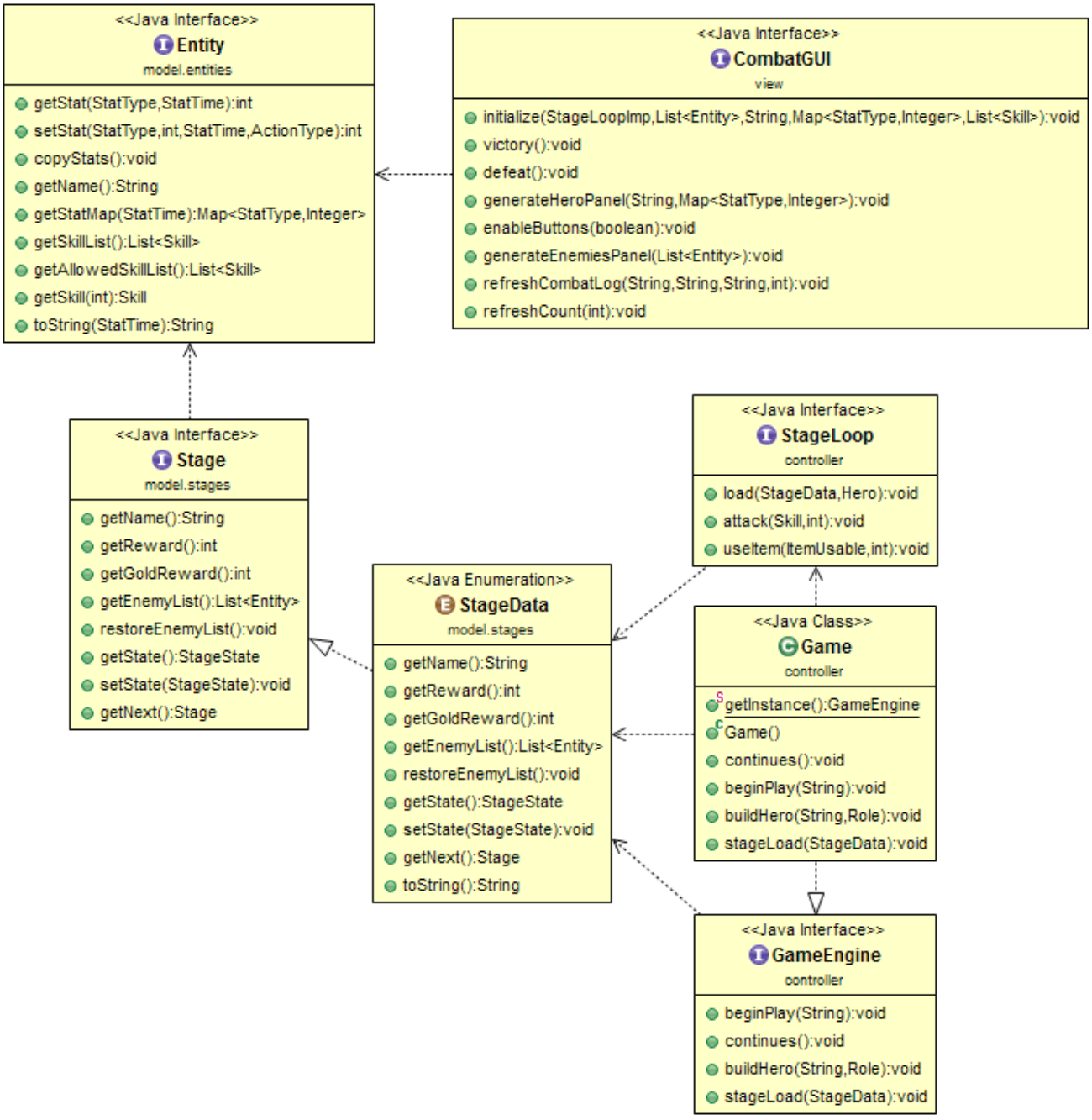
Il giocatore può scegliere nella creazione del personaggio la classe che intenderà giocare ognuna caratterizzata da diverse mosse.
La scelta dello stage varierà la difficoltà del gioco e i nemici che si ritroverà ad affrontare.
Attraverso il salvataggio il giocatore potrà riprendere una partita precedentemente giocata.
Il software è predisposto per eventuali aggiornamenti di mostri, ruoli, mosse e oggettistica.

1.2 Analisi e modello del dominio

Il videogioco consiste nel combattere all'interno di uno stage tramite scelta delle mosse con il quale il giocatore attacca i nemici e viceversa. Questo mette in relazione la varietà di entità, come mostri e ruoli dell'eroe, con le relative mosse da utilizzare in combattimento. Il giocatore verrà posto di fronte alla scelta dello stage che si vorrà affrontare.
Inizialmente all'avvio del gioco si creerà l'eroe decidendo nome e ruolo. L'unico stage da poter affrontare sarà il Tutorial il quale conseguimento, darà inizio allo sbloccaggio dei vari stage da affrontare. Tramite interfaccia sarà possibile selezionare stage già affrontati per aumentare esperienza e soldi necessari per il proseguimento del gioco. All'interno di ogni stage è possibile trovare diversi avversari, sia riferito al numero che alla loro tipologia. Il

[1] Role Playing Game o Gioco Di Ruolo (GdR): https://it.wikipedia.org/wiki/Gioco_di_ruolo
[2] Final Fantasy: https://it.wikipedia.org/wiki/Final_Fantasy
[3] Dragon Quest: https://it.wikipedia.org/wiki/Dragon_Quest

combattimento avviene a turni scanditi dalla velocità con la quale le varie entità possono eseguire i loro attacchi o attivare oggetti. Al termine del combattimento dopo aver ottenuto i premi si ritorna alla schermata di selezione stage ed il gioco prosegue.



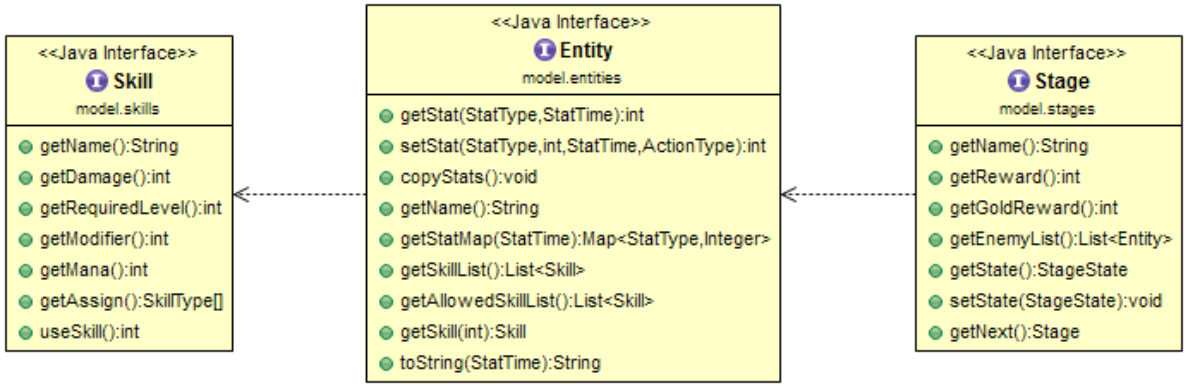
Design

2.1 Architettura

Per la gestione delle interazioni tra le varie componenti del software e la corretta suddivisione del lavoro tra i componenti del gruppo è stato utilizzato il design architetturale MVC (Model, View, Controller).
Il Controller inizializza il gioco caricando la View e ottenendo informazioni relative alle varie entità dal Model. Successivamente si effettuano varie interazioni tra le tre componenti mediante l'arbitraggio del Controller.

2.2 Design Dettagliato

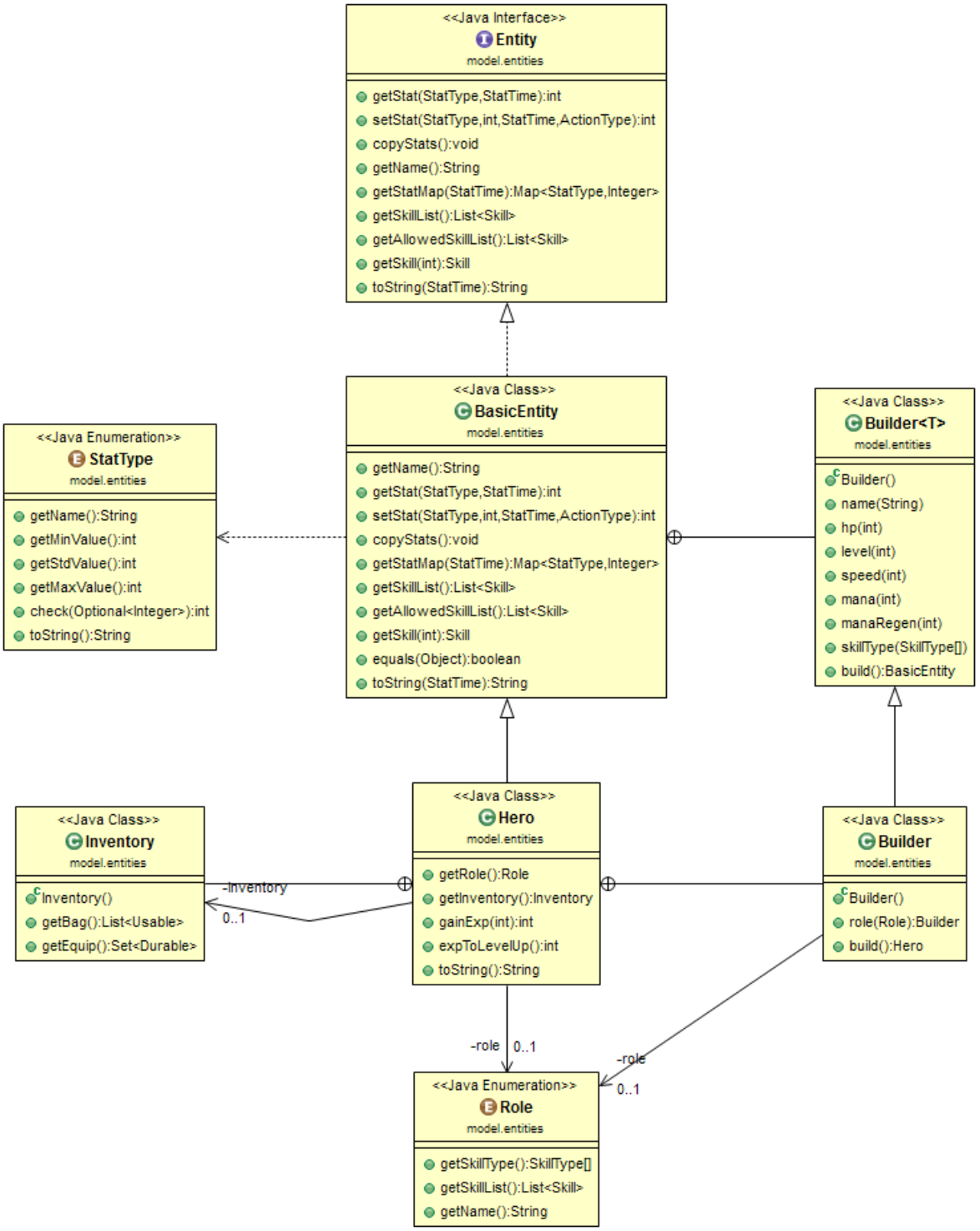
Model – Montini Leonardo



Enum

Per modellare le molteplici categorie di entità, statiche tra loro ma potenzialmente sempre soggette ad un arricchimento quantitativo, si è deciso di fare un massiccio uso del concetto di *Enum* offerto da Java, rendendo il codice sempre pronto ad eventuali aggiornamenti ed ampliamenti futuri del gioco.

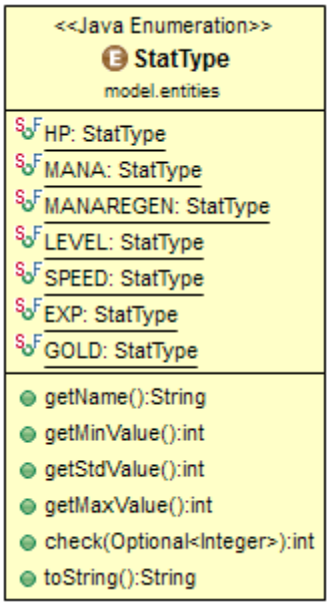
Entity



Entity è l'interfaccia principale per modellare un'entità, implementata da *BasicEntity* il cui compito è gestire le istanze dei nemici all'interno di uno stage.
Per quanto riguarda l'eroe, ossia il personaggio comandato dal giocatore, si è resa necessaria la creazione di un'ulteriore classe *Hero* poiché quest'ultima dovrà possedere caratteristiche aggiuntive le quali comporterebbero un errore concettuale nel caso venissero attribuite anche ad un semplice nemico.

Avendo l'eroe solamente caratteristiche aggiuntive rispetto ai mostri, si è deciso di sfruttare l'ereditarietà estendendo la classe *Hero* da *BasicEntity*.

StatType



Le statistiche principali delle entità sono generalmente rappresentate da valori interi, come punti vita, livello o punti esperienza.
Il problema di fondo era la gestione di ogni singola statistica, con tanto di getter e setter per ciascuna di esse e metodi aggiuntivi per aumentarne o decrementarne il valore.
Partendo da questi principi, si è deciso di gestire tutte le statistiche in maniera dinamica grazie all'enum *StatType.*
All'interno dell'entità saranno presenti due mappe per associare la statistica al relativo valore.
Questa scelta implementativa fornisce numerosi vantaggi, tra i quali:

- Massima estensibilità e predisposizione all'aggiunta di un'ulteriore tipologia, preoccupandosi solamente di aggiornare il costruttore.
- Struttura dati unica da passare alla *View* per la visualizzazione di tutte le statistiche.
- Pratica gestione di backup delle statistiche per ripristinare i valori di default ad inizio stage avendo una mappa "global" ed una "current".
- Un unico metodo *getStat* e *setStat* che permettono di effettuare tutte le operazioni necessarie.
- Controllo della correttezza dei valori inseriti dinamico e generalizzato con un unico metodo.

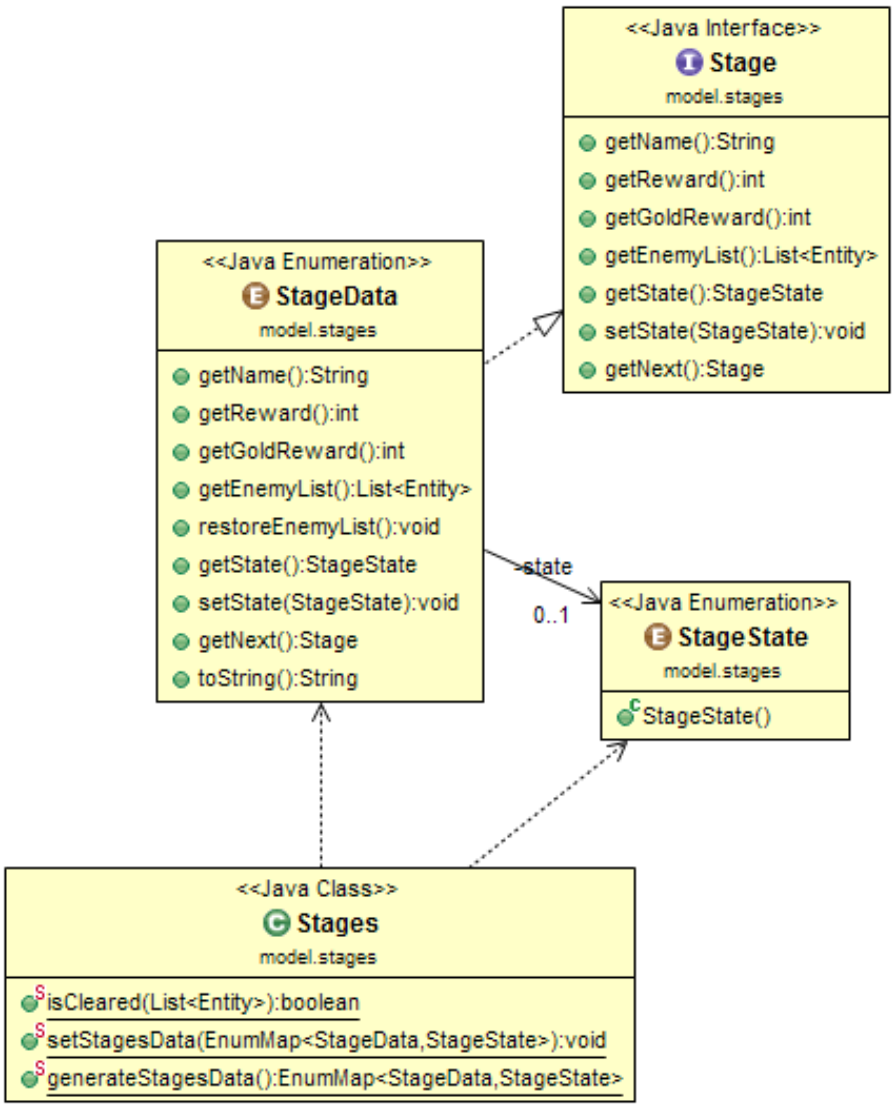
Importante precisare la scelta dello scope della mappa "global" a protected anziché private, questo per dare la possibilità alla classe Hero di aggiungere valori alla mappa stessa.

Builder

Per evitare costruttori con un elevato numero di campi, si è deciso di sfruttare il pattern builder, associando l'inserimento di ogni campo ad un metodo con il relativo nome.

La scelta di questo pattern ha reso necessario un approfondimento dello stesso per permettere la compatibilità del builder di *BasicEntity* attraverso l'ereditarietà su Hero.
Da una risposta della comunità di StackOverflow[4] è stata reperita una possibile soluzione di builder applicato all'ereditarietà, riadattato per modellare la situazione in questione.

Stage



Il vero e proprio combattimento avviene sulla struttura dello *Stage*.
La gestione delle diversità tra gli stage è affidata all'enum *StageData* che implementa l'interfaccia *Stage* contenente la dichiarazione dei metodi necessari per estrarne le informazioni necessarie.
Nella creazione di uno *Stage*, così come per l'assegnazione di alcune caratteristiche in casi trattati nelle righe successive, è stato scelto un costruttore capace di ricevere un numero variabile di argomenti, permettendo di inserire solamente i mostri in maniera sequenziale nella dichiarazione delle singole istanze dell'enum *StageData*, delegando il compito di

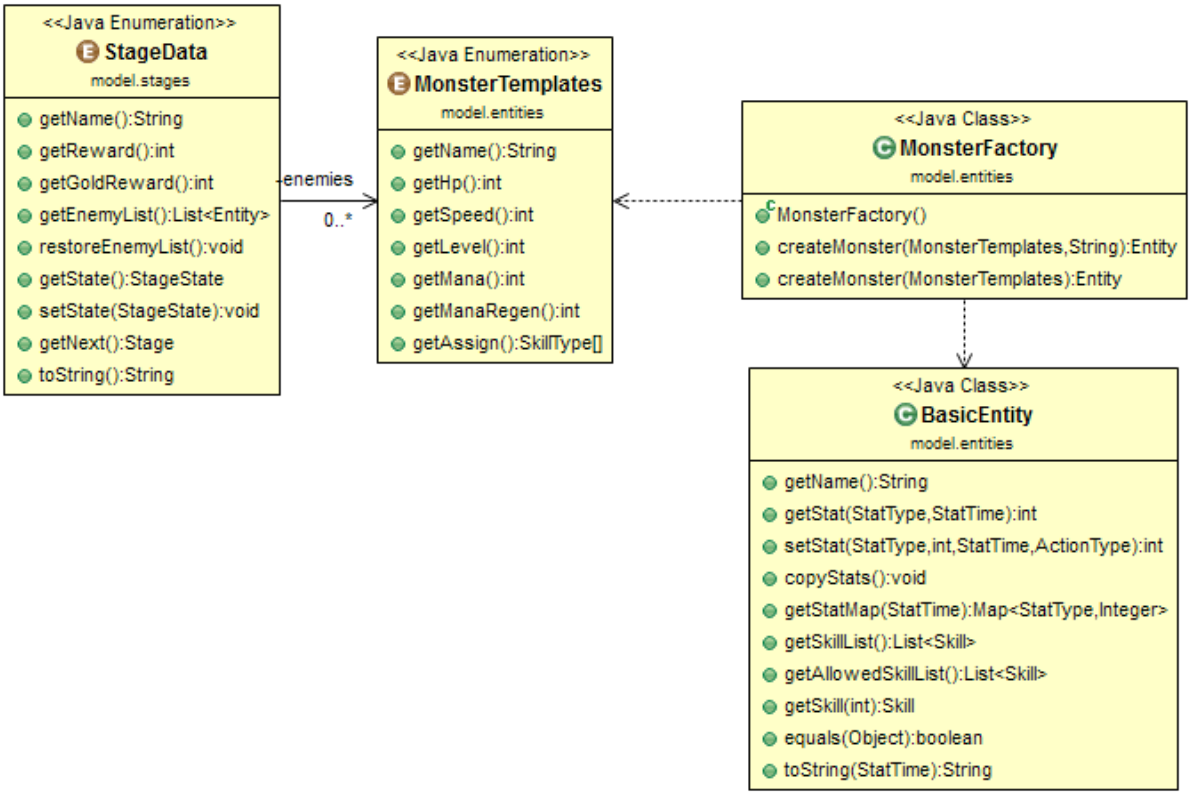
[4] Link: http://stackoverflow.com/questions/17164375/subclassing-a-java-builder-class

riorganizzarle in lista al costruttore stesso, evitando contorte e più difficilmente leggibili dichiarazioni.

Stages

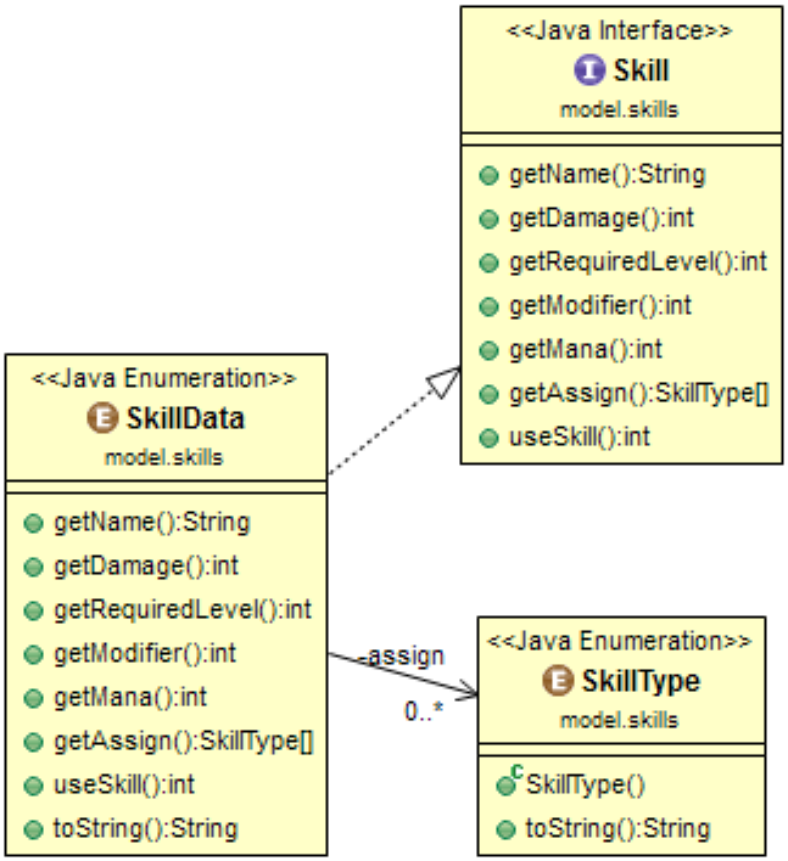
Per la gestione di alcuni aspetti comuni a tutti gli stage si è deciso di creare una classe a parte con tre metodi statici, per alleggerire il codice dell'enum *StageData*.

MonsterFactory



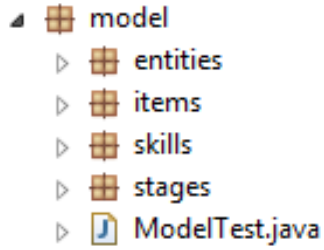
Per la creazione e l'istanziazione dei mostri (*BasicEntity*) all'interno di uno stage si è deciso di utilizzare il pattern Factory.
In questo modo è prevista la ripetizione di uno stage più volte generando nuove istanze di nemici ad ogni combattimento in una singola esecuzione dell'applicazione.
Per aggiungere un fattore casuale aggiuntivo la factory accetta in input oltre ad un *MonsterTemplate* da cui generare l'istanza *BasicEntity* corrispondente, una stringa da aggiungere come suffisso al nome del mostro appena creato.
Nell'eventualità non si volesse usufruire di questa particolarità esiste un secondo metodo della factory richiedente in input solamente il template del mostro, richiamando il primo metodo con una stringa vuota.

Skill



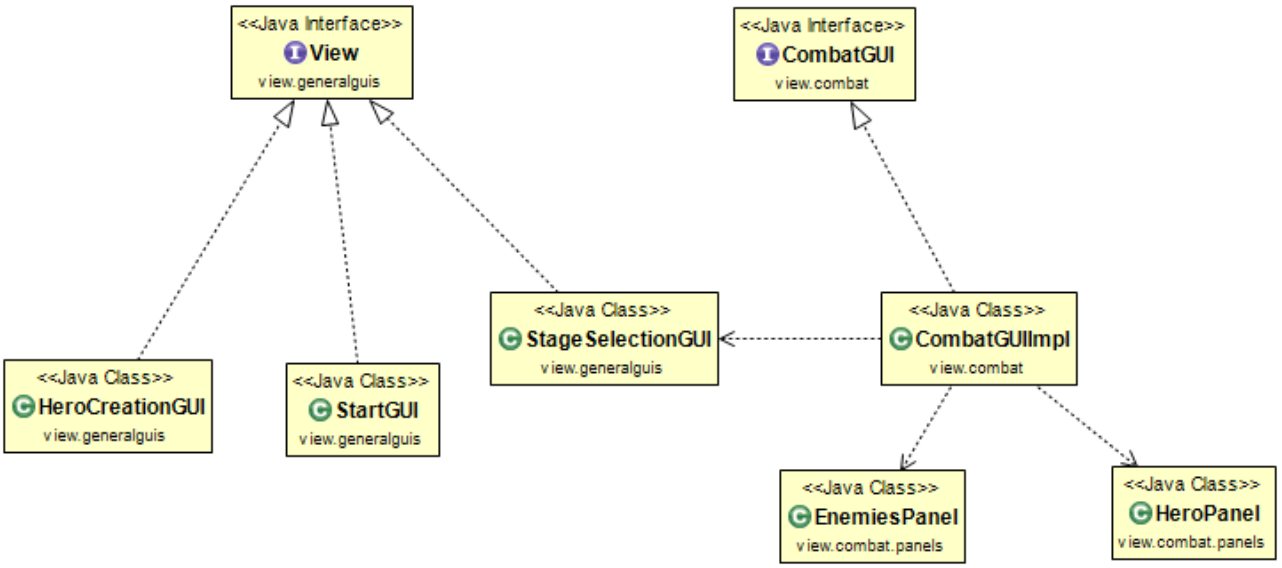
Anche per la gestione degli attacchi (*Skill*) si è deciso di combinare l'utilizzo di Enum per la gestione dei dati relativi a ciascuna skill e varArgs per assegnare una skill a più categorie di mosse (*SkillType*).
Questa seconda scelta è stata intrapresa per permettere ad un'*Entity* di avere mosse di tipologie diverse e per assegnare ad ogni mossa una o più tipologie.
Con i pochi dati attualmente presenti nel gioco l'utilità di questo aspetto viene messa poco in luce, ma nell'eventualità di un'espansione futura dove potranno essere presenti molte più classi e mosse tornerà sicuramente utile permettere ad alcune particolari skill di appartenere a più categorie diverse.
Per la costruzione della lista delle mosse è stato implementato un metodo statico *computeSkillList* che accetta un numero variabile di *skillType* e genera la lista effettiva delle *skill* senza duplicati, inserendo solamente le skill delle categorie passate in input.

Package



La suddivisione in package è stata effettuata con criterio concettuale, raggruppando le classi con maggior interazione ed affinità tra loro, rendendo la navigazione all'interno dell'albero dei package agevole ed intuitiva, permettendo una rapida ricerca.

View – Tonucci Riccardo



Per l'implementazione della View è stato scelto di utilizzare la libreria Swing fornita da Java. La View si occuperà di rappresentare i dati del Model all'interno del gioco che il Controller le fornisce.

E' stato deciso di suddividere la View in tre principali blocchi:
Il primo blocco gestisce tutte le viste che sono utilizzate dall'avvio del software al gioco effettivo;
Il secondo blocco gestisce la rappresentazione dei salvataggi del gioco nella View;
Il terzo blocco rappresenta il gioco vero e proprio con tutti gli elementi necessari all'interattività con l'utente.

View e GUI generali

E' stato deciso di implementare ogni GUI generale tramite una singola interfaccia denominata *View*.
Questa interfaccia contiene un singolo metodo che permette a ogni GUI di essere inizializzata, in maniera tale da generare i componenti e impostare le caratteristiche necessarie all'avvio e posizionamento della GUI stessa.
In questo modo in futuro sarà possibile inserire altre GUI semplicemente creando una classe e implementando il metodo *initialize*.

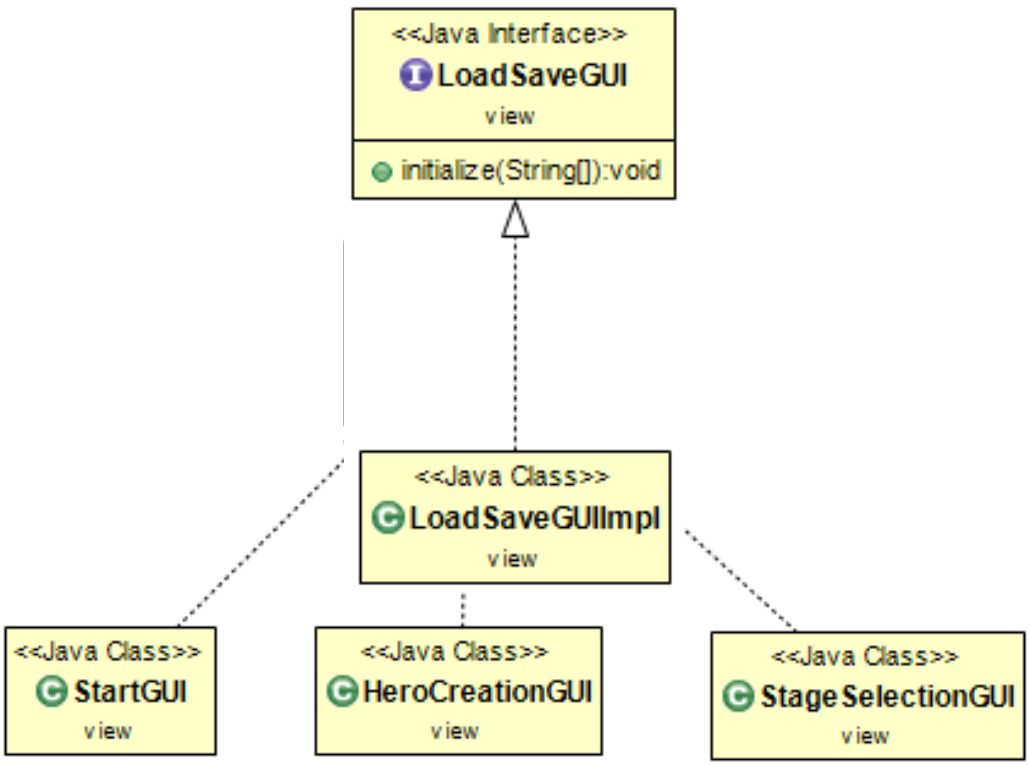
All'avvio del gioco veniamo posti davanti alla StartGUI nella quale è possibile avviare una nuova partita, caricare una partita precedentemente salvata o visualizzare i crediti.
Se decidiamo di avviare una nuova partita o caricare un salvataggio, scegliamo di interagire con il Controller che dopo aver ottenuto informazioni dal Model, creerà istanze di altre GUI a seconda del tasto premuto.

La scelta di utilizzare dei pulsanti per interagire con il Controller è stata fatta in modo tale da predisporre il software per eventuali aggiornamenti futuri o funzionalità aggiuntive.

La *HeroCreationGUI* una volta istanziata dal Controller permette di creare il proprio personaggio, scegliendo nome e ruolo dello stesso

L'ultima classe che ho ritenuto necessario implementare è la *StageSelectionGUI*, con essa è possibile selezionare lo stage di battaglia che si vuole affrontare.

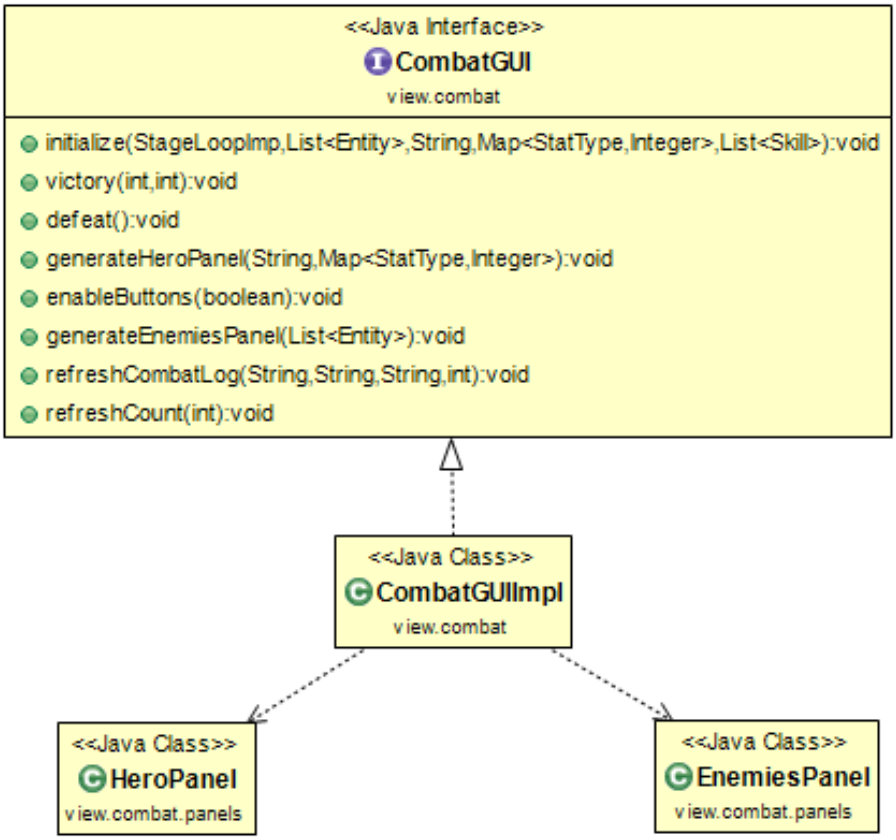
LoadSaveGUI



L'interfaccia *LoadSaveGUI* implementa solamente la classe *LoadSaveGUIImpl*, questa sceltà è stata fatta poiché ritengo che il caricamento del salvataggio di una partita precedente sia una funzionalità del gioco non strettamente collegata alle altre GUI.

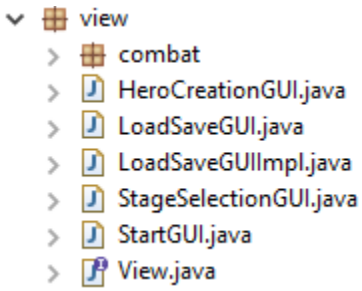
Tramite questa GUI è quindi possibile selezionare e fornire al Controller il salvataggio di una partita precedente giocata.

CombatGUI



L'interfaccia *CombatGUI* implementa la classe omonima.
Tramite questa classe il Controller può richiamare i vari metodi messi a disposizione per generare o aggiornare i componenti della View.
Inoltre alcuni metodi sono anche usati dal costruttore della *CombatGUI* stessa la prima volta che essa viene generata dal Controller.
Le due classi *HeroPanel* e *EnemiesPanel* gestiscono rispettivamente il pannello delle statistiche dell'eroe e dei nemici. Ho preferito suddividere la *CombatGUI* in questi due ulteriori pannelli per poi aggiornarli tramite metodi omonimi. Così facendo con aggiornamenti futuri sarà possibile aggiungere statistiche aggiuntive alle entità del software.

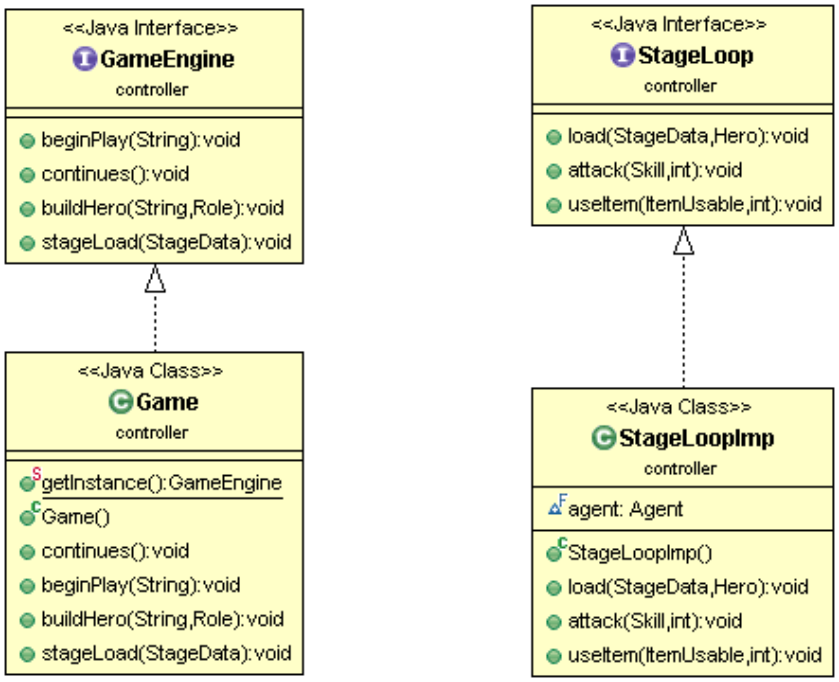
Package



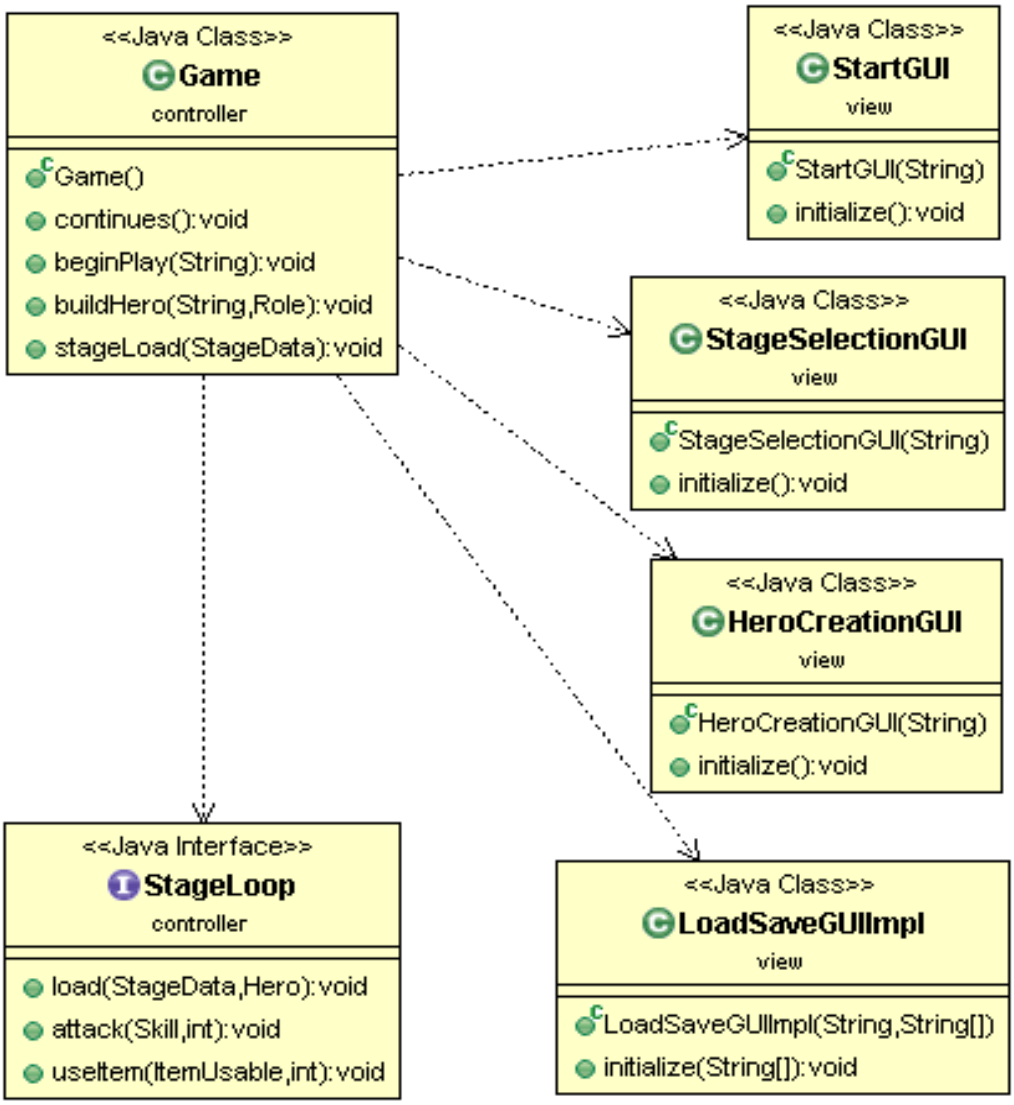
La realizzazione del package è stata effettuata in modo tale da consentire una divisione tra le GUI principali e quella del combattimento, cosìcchè si può facilmente accedere alle varie classi e interfacce. Inoltre sono stati suddivisi anche i due suddetti pannelli all'interno del sotto-package *panels*.

Controller – Zamagni Enrico

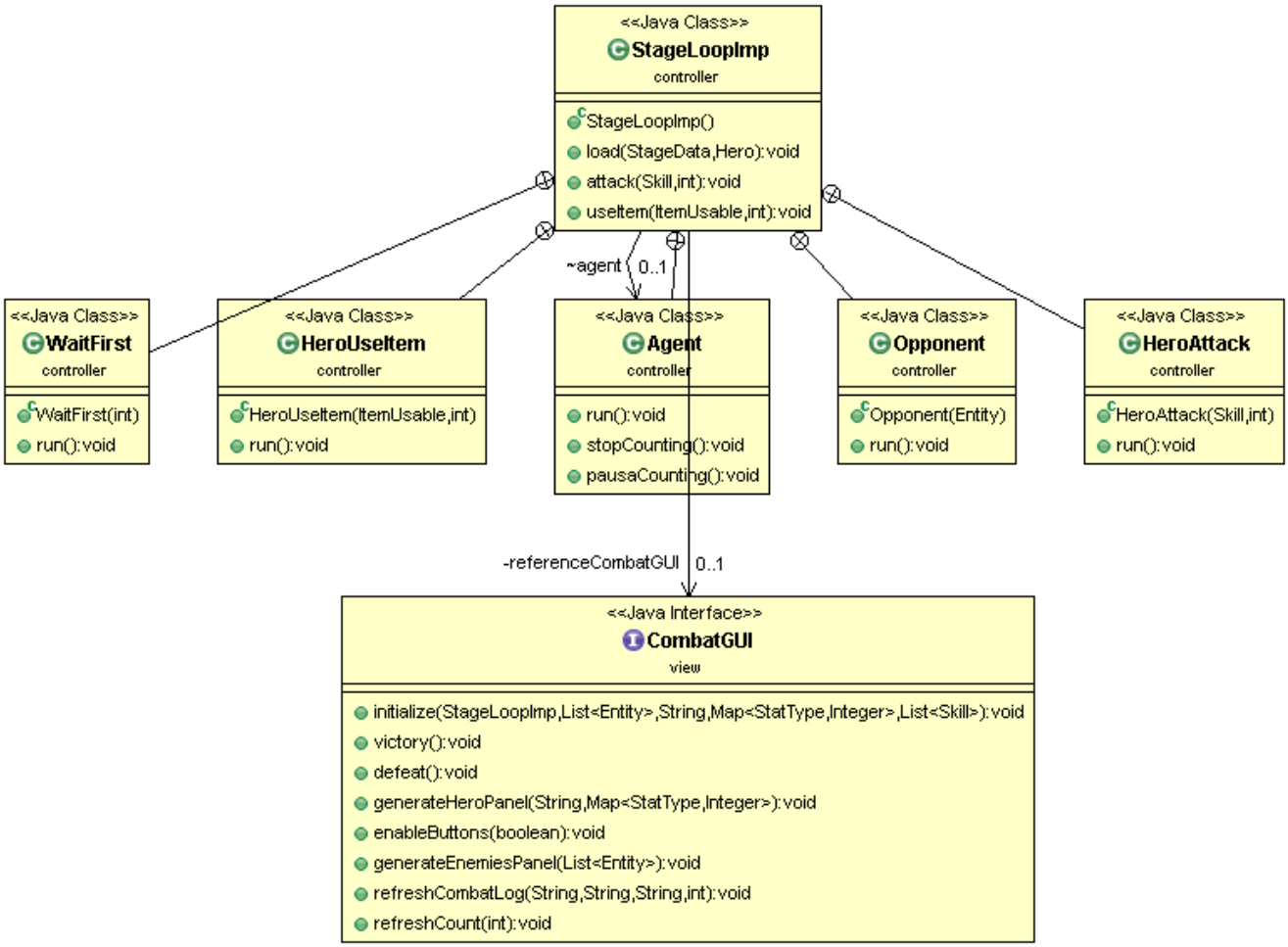
Il controller è la coscienza del gioco in quanto utilizza il *Model* come stampini e la *View* come interlocutore con il giocatore.
Ho deciso di creare due grandi interfacce attraverso il quale manovrare tutto il gioco e comunicare con la View.
Dalle interfacce discendono una classe ciascuna. La prima è la classe *Game* la quale si occupa di tutta quella parte non inerente al combattimento e la seconda *StageLoopImp* che si occupa solo di quello.



Per semplificare la comunicazione con la *View* e far sì che si creasse una sola Istanza per la gestione del caricamento dati salvati e degli stage, ho deciso di utilizzare il pattern Singleton all'interno della classe *Game*. Essa comunica e si indirizza allo *stageLoop* attraverso il susseguirsi di chiamate alla *View*.



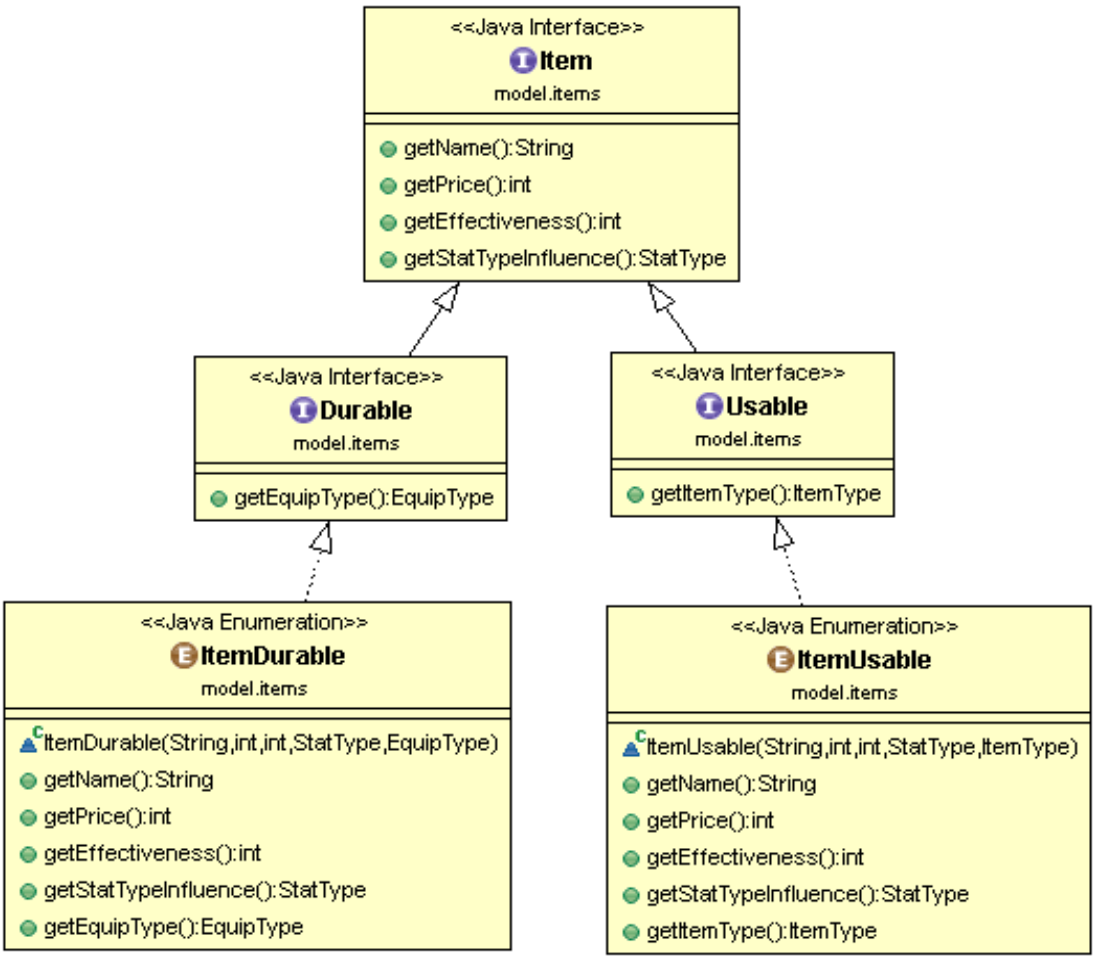
Il cuore del gioco è nello *StageLoop*, il combattimento.
Il fatto di voler dare dinamicità allo scontro a fatto si che mi indirizzassi sull'utilizzo di thread. Per poter gestire lo scorrere del tempo infatti mi sono ispirato alla lezione in laboratorio sulle *reactivegui*. Il fatto di poter utilizzare un contatore esterno che scandisse il tempo e alternasse gli attacchi dei mostri è stato di fondamentale importanza, anche perché per poter bloccare il tempo durante la scelta da parte del giocatore su che mossa fare, avrei dovuto tenere traccia dei vari thread mostri e bloccarli in contemporanea per poi rifarli partire senza perdere il sincronismo. Mentre per i mostri ho assegnato un thread a ciascuno, per l'eroe il thread creato è di gestione della singola azione. Inizialmente non avevo assegnato un thread all'eroe ma quando è iniziata l'interazione con la *View* è stato palese che fosse necessario per poter modificare le statistiche in tempo reale.



Per il salvataggio dai dati ho serializzato l'eroe e le informazioni degli stage, di modo che alla ripresa successiva del salvataggio si torna precisamente allo stesso stato precedente alla chiusura del gioco.
Infine ho preimpostato l'utilizzo dell'inventario, anche se ancora non utilizzabile. Utilizza un thread molto simile a quello che gestisce l'attacco con la differenza nella selezione del bersaglio in base alla tipologia dell'item.

ModelOggettistica

Per la creazione dell'inventario mi sono basato sulla tipologia adottata nel resto del model ritenendola facilmente espandibile in futuro ed elastica. Utilizzando le enum e un susseguirsi di interfacce ho gestito l'inventario.



Sviluppo

3.1 Testing Automatizzato

La maggior parte degli aspetti di modellazione e controllo tra le funzioni base delle entità è stato gestito tramite la libreria *JUnit*.
Nello specifico, tra gli aspetti principali, il testing è stato mirato al corretto funzionamento della creazione dell'eroe con relativa gestione dei controlli dei valori in input e conseguenti eccezioni.
Particolare attenzione sul metodo *gainExp* con una porzione di test interamente dedicata, a causa dei numerosi effetti a cascata provocati dalla sua chiamata (aggiornare il livello, aumentare le statistiche e resettare i punti esperienza).
Si è deciso di automatizzare anche vari aspetti relativi agli stage ed alla gestione degli oggetti ed equipaggiamenti.

3.2 Metodologia di lavoro

Montini Leonardo

Nella suddivisione dei compiti ho scelto di gestire la parte di model, secondo l'architettura MVC.
Nello specifico mi sono occupato della creazione ed implementazione delle classi nel package *model* in tutte le sue classi, ad eccezione del sotto-package *items* a cui ha lavorato Zamagni.
Ho curato particolarmente tutti gli aspetti dinamici ed estensibili in modo da rendere pratico un eventuale nuova versione del gioco con nuovi nemici (*MonsterTemplate*), mosse (*SkillData*), missioni (*StageData*), classi di eroi (*Role*), nomi casuali (*RandomName*) e con qualche leggero aggiornamento anche statistiche (*SkillData*).
Avendo iniziato un po' prima ho messo le basi per far lavorare il resto del team, aggiungendo eventuali funzioni su richiesta dei colleghi ove necessario.
Il DVCS Mercurial, mi è tornato utile diverse volte in situazioni di codice non funzionante per tornare alla versione precedente e funzionante da cui ripartire.

Tonucci Riccardo

Ho sviluppato in autonomia tutta le interfacce e le classi della View.
Ho costruito le GUI in maniera tale che in futuri aggiornamenti siano facilmente modificabili.
Mi sono accordato con i miei colleghi prima di iniziare a scrivere codice per quanto riguarda il parere soggettivo sulla vista. Così facendo ci siamo accordati su dove andavano posizionati i componenti della View e su come andavano fatte le varie viste.

Zamagni Enrico

Ho realizzato le classi e le interfacce presenti nel package controller e quelle nel package model.items. Ho implementato all'interno del model.ModelTest il test testInventory. Mi sono dovuto confrontare più volte con gli altri membri del gruppo per comprendere e modellare la logica del gioco.
All'inizio, per poter procedere indipendentemente, mi sono concordato con Tonucci per le interfacce di comunicazione tra Controller e View.

Commenti Finali

4.1 Autovalutazione

Montini Leonardo

Fin dall'analisi della mia parte ho ragionato su quale fosse il metodo migliore per gestire l'estensibilità del codice, mantenendo slegate le mie modifiche sul lavoro dei colleghi, studiando una possibile soluzione ricaduta nell'uso ripetuto della struttura *Enum*.

Ho trovato fin dai primi test molto pratico l'utilizzo di *Enum*, tanto da modificare durante il percorso alcuni aspetti fino ad arrivare a gestire ogni entità in gioco con un meccanismo simile, testato e ben funzionante.
Anche l'utilizzo dei *VarArgs* si è rivelato pratico ed adatto allo scopo per il quale sono stati utilizzati.
Molto comoda è stata la combinazione Mercurial - Bitbucket, che sto attualmente usando anche per il progetto di un altro corso con il medesimo scopo di gestione del lavoro di più persone su un unico progetto.
Ho tentato di dare il massimo per spronare il team nel contribuire al progetto, collaborazione purtroppo riscontrata solamente con l'avvicinarsi della deadline, garantendo troppo poco tempo per la realizzazione di quanto previsto.
Sicuramente se si fosse iniziato con maggior anticipo la qualità del prodotto finale sarebbe stata decisamente migliore, farò frutto in ogni caso di questa esperienza andata non proprio come previsto.

Tonucci Riccardo

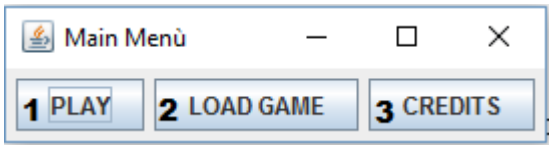
Nonostante sia stata una delle mie prime esperienze in ambito di lavoro in team, sono piuttosto soddisfatto del lavoro eseguito.
Ho trovato molto utile e fondamentale l'utilizzo di Mercurial e BitBucket anche non avendolo mai utilizzato prima.
Penso che la mia parte di progetto possa essere migliorata e aggiornata in futuro magari con l'aggiunta di nuovi contenuti.
Ho anche capito che è molto difficile sviluppare la grafica di un videogioco nonostante l'utilizzo di librerie come Swing facilitino abbastanza il compito.
Sono molto contento che mi sia stata concessa un occasione del genere per capire come si lavora in team.

Zamagni Enrico

Sono abbastanza soddisfatto del lavoro che ho compiuto. Ho fatto purtroppo più volte confusione il chè mi ha portato via più tempo del previsto e il non essere scesi più di tanto nel dettaglio nella logica di gioco, ha fatto si che mi si accumulasse il lavora a tratti. Estremamente utile ed istruttivo il dover collaborare in un gruppo che accentua l'importanza dell'analisi iniziale e l'importanza della comunicazione tra i membri. A volte scomodo ed a volte utile per chiarimenti o idee implementative. Mi sarebbe piaciuto anche utilizzare la parte che ho implementato dell'oggettistica apparentemente funzionante nei test. Rimarrà perciò la possibilità di ampliare il progetto rendendola parte integrante del gioco.

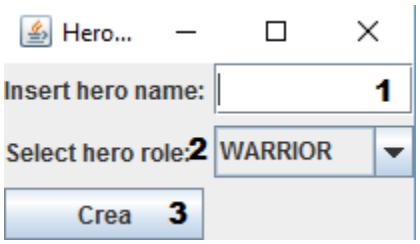
Guida Utente

Menu principale



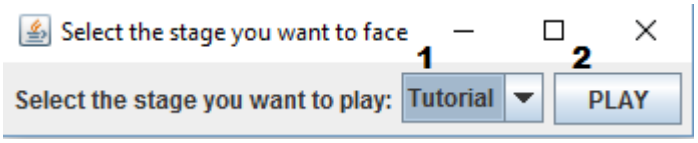
1. Tasto Play per iniziare una nuova partita
2. Tasto Load Game per caricare un salvataggio di una partita precedente
3. Tasto Credits mostra i crediti

Creazione dell'Hero



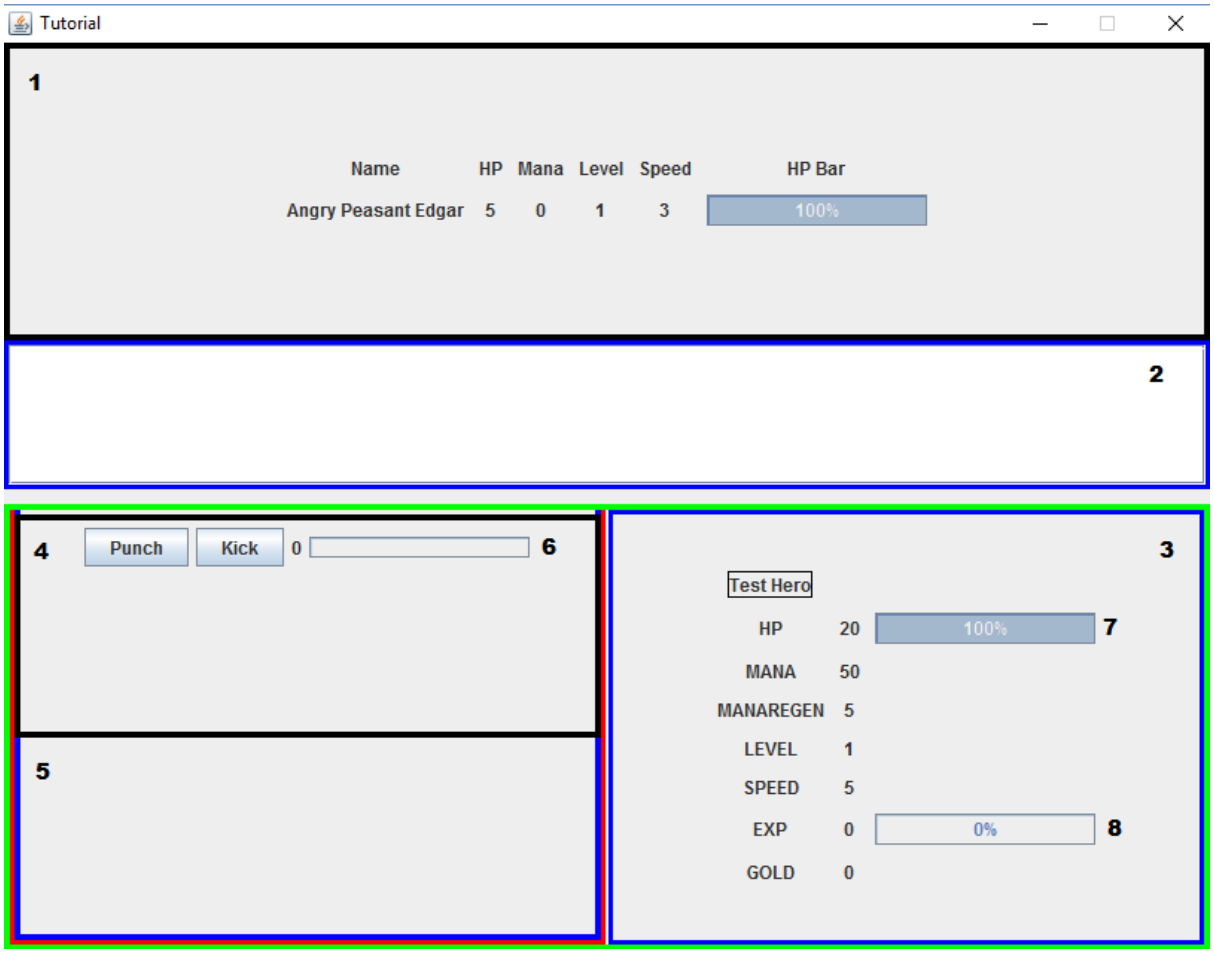
1. Inserire il nome del personaggio
2. Selezionare il ruolo del personaggio
3. Premere il tasto crea per creare il personaggio

Selezione dello Stage



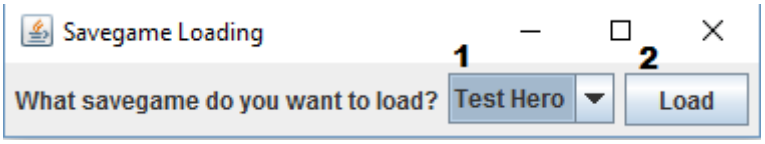
1. Selezionare lo stage che si vuole giocare, all'inizio l'unico sbloccato sarà il tutorial, vincendo il tutorial si sbloccheranno gli stage successivi
2. Premere il tasto play per giocare lo stage selezionato

Schermata di combattimento



1. Pannello delle statistiche del nemico
2. Pannello delle informazioni del combattimento
3. Pannello delle statistiche dell'eroe
4. Pannello delle abilità dell'eroe
5. Pannello degli oggetti (non ancora implementato)
6. Barra del turno, quando la barra arriva a 0, è il turno dell'eroe
7. Barra dei punti vita dell'eroe, quando la barra arriva a 0 si perde lo stage
8. Barra dell'esperienza dell'eroe

Caricamento salvataggio



1. Selezionare il nome del salvataggio che si vuole caricare
2. Premere il tasto Load per caricare il salvataggio