Relazione per il progetto “Bomberman”

Giacomo Frisoni, Giulia Giombini, Sofia Rossi

15 maggio 2016

Indice

Analisi 3

1.1 Requisiti 3

1.2 Analisi e modello del dominio 4

Design 5

2.1 Architettura 5

2.2 Design Dettagliato 6

Sviluppo 28

3.1 Testing automatizzato 28

3.2 Metodologia di lavoro 28

3.3 Note di sviluppo 29

Commenti finali 31

4.1 Autovalutazione 31

Appendice 33

Guida utente 33

Analisi

1.1 Requisiti

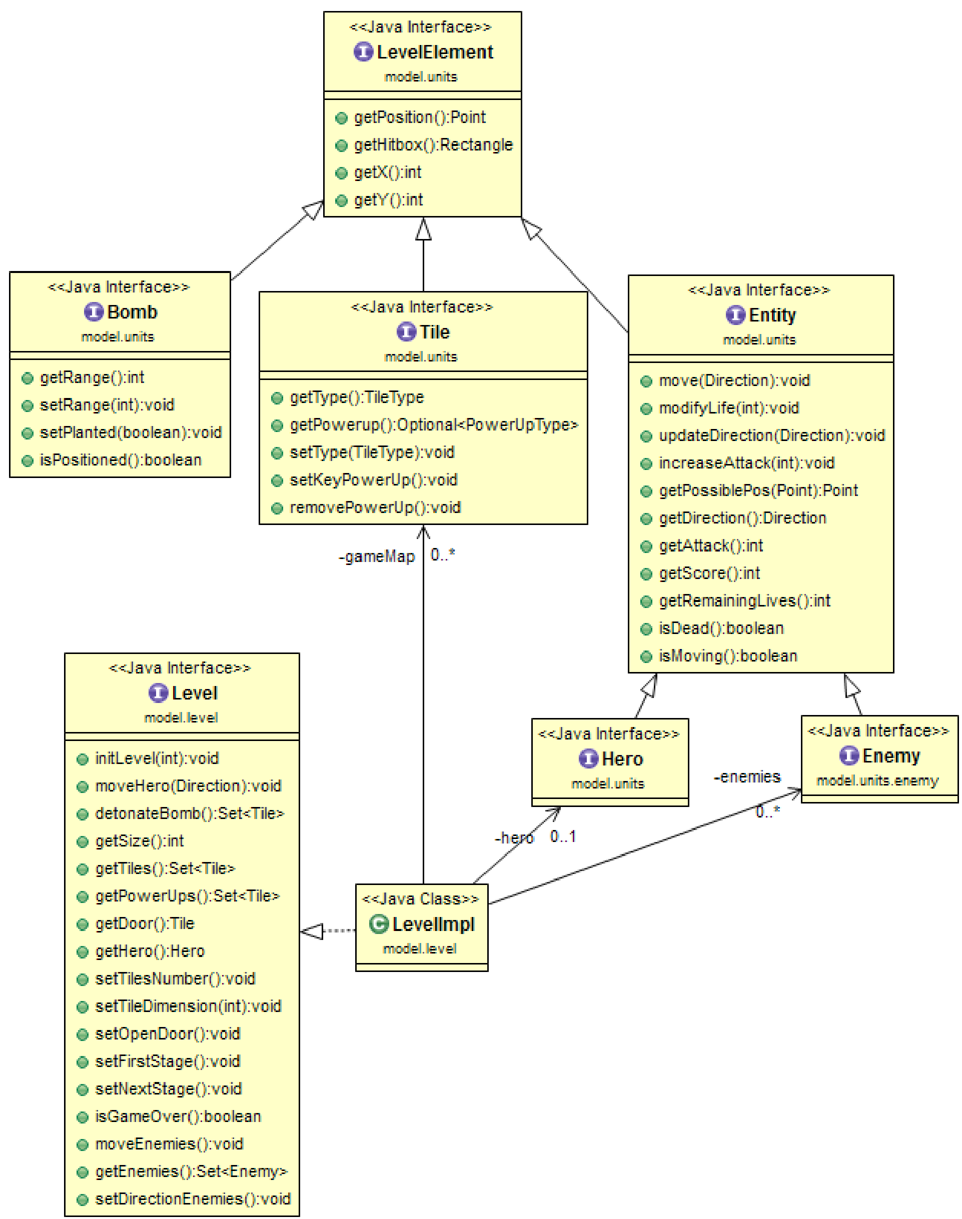
Il software mira alla realizzazione di una versione riadattata del noto videogame del genere Maze, “Bomberman[1]”, in occasione del 30° anniversario dell’uscita per NES.

Bomberman è un rompicapo ambientato all’interno di un labirinto dove è presente il personaggio principale che l’utente, tramite comandi, deve controllare per accedere al livello successivo. Le funzionalità basilari di questo software risultano essere quindi la visualizzazione dell’eroe, dei nemici e del background di gioco, nonché l’interazione tra i vari elementi contenuti all’interno di uno stage.
Il genere Maze, inizialmente coniato nel 1980, identifica qualsiasi gioco in cui le varie azioni si svolgono all’interno di un labirinto. Al fine di evitare nemici, aggirare ostacoli (come le bombe) o riuscire a raggiungere un tempo record di superamento di livello, è necessaria una particolare attenzione e velocità del giocatore nel prendere le decisioni.

- Il giocatore dovrà avere i supporti necessari per muovere il protagonista all’interno della mappa. Inoltre dovrà essere in grado di piantare bombe finalizzate alla distruzione di blocchi e antagonisti;
- Le collisioni con gli altri elementi di gioco porteranno a diverse conseguenze. In presenza di un contatto eroe-nemico vi sarà una eventuale morte dell’eroe stesso, con la relativa fine del livello. Per quanto concerne gli elementi statici del gioco, quali power-up o blocchi, si applicheranno i relativi effetti sull’eroe o sul suo movimento;
- Il livello, generato casualmente, rappresenterà la partita corrente. Questo avrà una sua difficoltà, identificata dalla resistenza e dalla potenza dei nemici che ostacoleranno in modo graduale il superamento di stage successivi;
- Sarà fornita la possibilità di utilizzare una seconda modalità di gioco, denominata “DarkMode”, in cui l’utente non avrà la visibilità completa della mappa, ma solo di un’area delimitata sulla base della posizione in cui si troverà in un certo istante;
- Il giocatore potrà visualizzare l’andamento delle sue ultime partite attraverso un grafico. Questo sarà realizzato grazie ai vari punteggi e tempi di gioco registrati al termine di ogni partita.

[1] Pagina di Wikipedia relativa a Bomberman: https://it.wikipedia.org/wiki/Bomberman

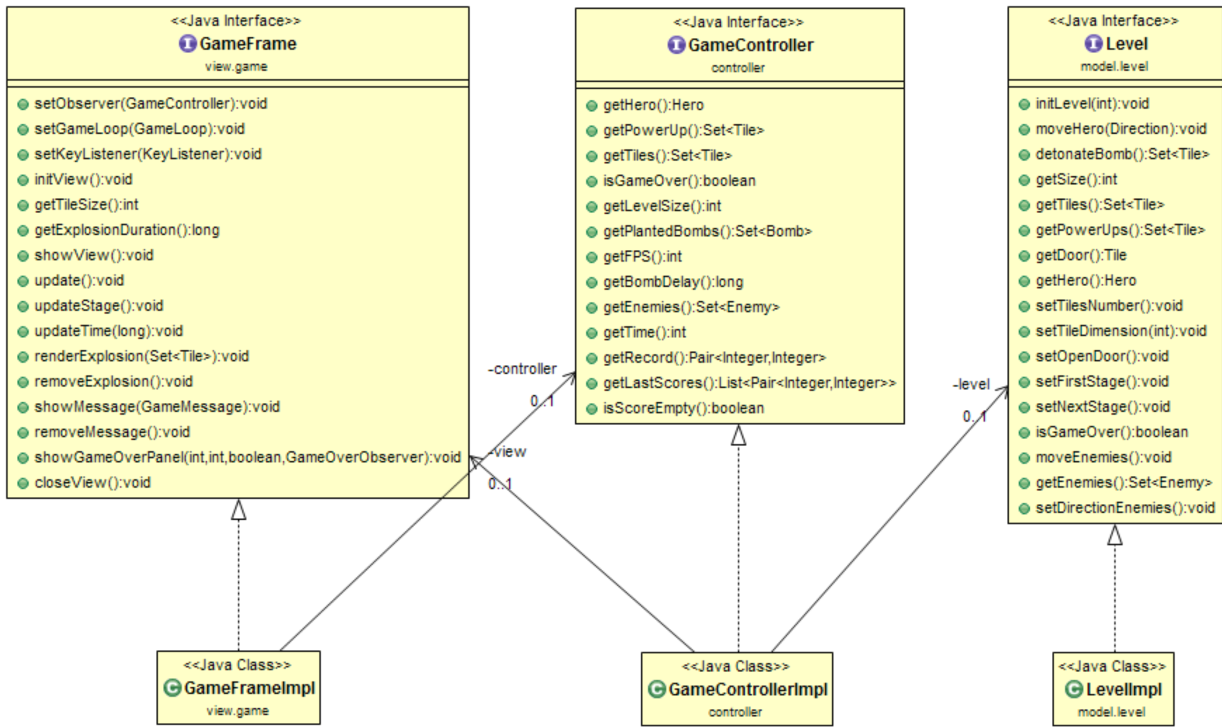
1.2 Analisi e modello del dominio



Il programma dovrà essere in grado di gestire una serie di livelli. Tale insieme di livelli costituisce una partita. Ogni livello è formato da un certo quantitativo di tile, rappresentanti la mappa di gioco su cui delle entità possono muoversi.
Le entità sono di due tipologie differenti: eroe e nemici. Un altro elemento di fondamentale importanza presente all'interno di una partita è rappresentato dalla bomba, che l'eroe può rilasciare durante il suo cammino. Gli elementi costitutivi il problema sono sintetizzati in figura.
La difficoltà primaria sarà quella di riuscire a gestire le collisioni fra i vari elementi di gioco e i fenomeni che ne conseguiranno. Questo richiederà un design attento.
L'interfaccia grafica dovrà essere flessibile e in grado di adattarsi a risoluzioni differenti. Inoltre, la gestione delle animazioni delle varie entità dovrà esser ben progettata.
Un altro aspetto cruciale riguarda la corretta gestione dei thread associati alle bombe che potranno essere attive contemporaneamente su schermo.

2.1 Architettura

Per gestire l'interazione fra le componenti principali del software si è scelto di adottare il pattern architetturale MVC (basato sulla organizzazione del progetto in tre parti distinte: Model, View e Controller).
Il seguente diagramma UML rappresenta l'architettura del software ad alto livello:



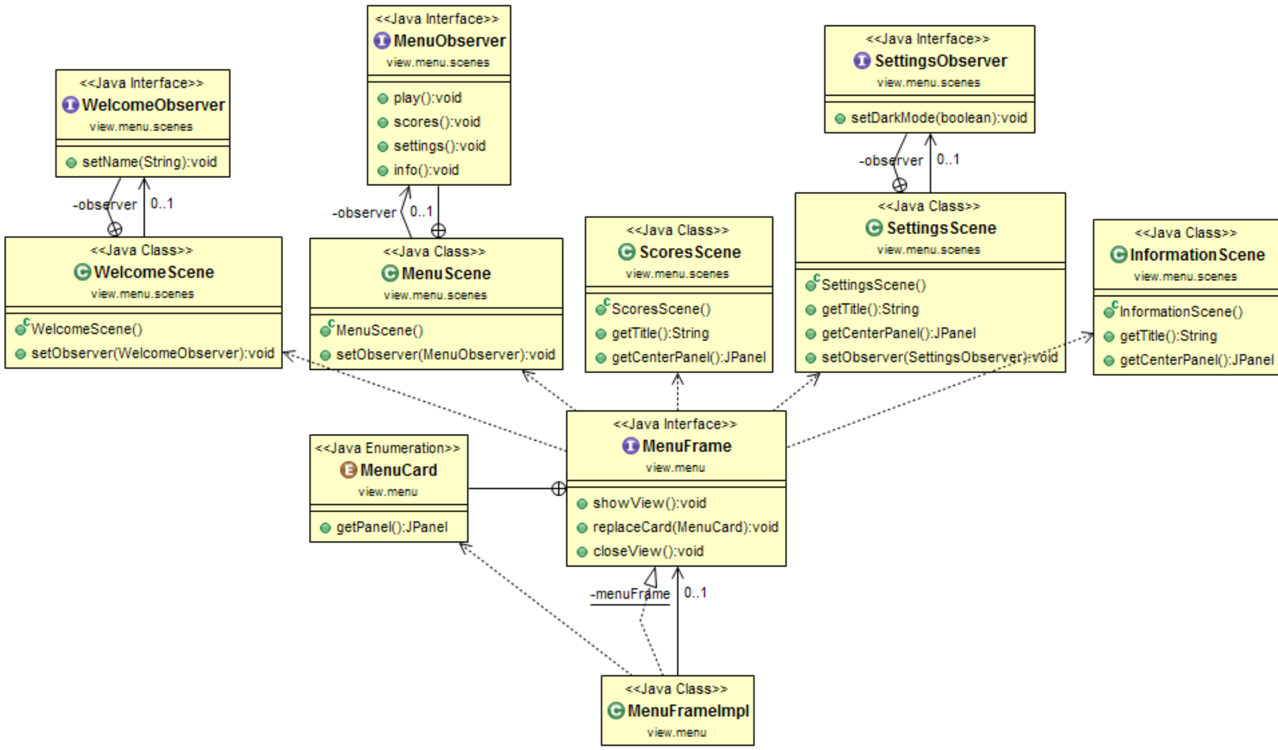
Model, View e Controller si compongono di altre classi che per motivi di chiarezza non sono state incluse in questo primo diagramma.
All'inizio di una partita, viene creato un Controller incaricato di istanziare il Model e la View, inizializzandoli in maniera opportuna.
La View fornisce una rappresentazione grafica del Model e, grazie all'uso di questo design architetturale, è studiata in modo tale da poter essere facilmente sostituibile in blocco (senza determinare sostanziali cambiamenti nelle altre parti).
Se in futuro si decidesse di realizzare una modifica di questo tipo, infatti, sarebbe sufficiente creare una nuova classe che implementi l'interfaccia *GameFrame*, aggiornando i relativi riferimenti.
Le componenti MVC interagiscono tra loro unicamente attraverso i metodi pubblici delle interfacce (implementate dalle rispettive classi).

2.2 Design Dettagliato

View – *Giacomo Frisoni*

La View si occupa di gestire l'interazione con l'utente e di mostrare a video una rappresentazione grafica dello stato del Model durante una partita. Per realizzarla è stato scelto di utilizzare la libreria Swing al fine di poter mettere in atto tutte le conoscenze apprese durante il corso.
La View è stata sviluppata interamente sulla base del pattern architetturale MVC.

All'avvio dell'applicazione viene visualizzato un menù dedicato alla navigazione delle viste, che consente l'accesso ai vari contenuti del gioco.
Al fine di rendere la gestione delle viste interne al menù il più possibile flessibile ed espandibile, è stato scelto di trattare ogni schermata in maniera indipendente dalle altre e di considerare il loro "replacement" come un aggiornamento del pannello contenuto all'interno del frame principale (*MenuFrameImpl*).



L'interfaccia *MenuFrame* presenta la definizione dell'enumerazione *MenuCard*. Quest'ultima rappresenta l'insieme dei pannelli che possono essere sostituiti e, dunque, mostrati all'interno della finestra. Nello specifico, le possibili scene del menù che possono essere consultate sono:

- *WelcomeScene*, mostrata nel momento in cui non risultano essere ancora presenti file di salvataggio legati all'applicazione
- *MenuScene*, schermata di home a partire dalla quale avviene la navigazione
- *ScoresScene*, dedicata alla visualizzazione delle principali statistiche inerenti i punteggi raggiunti dall'utente nel gioco

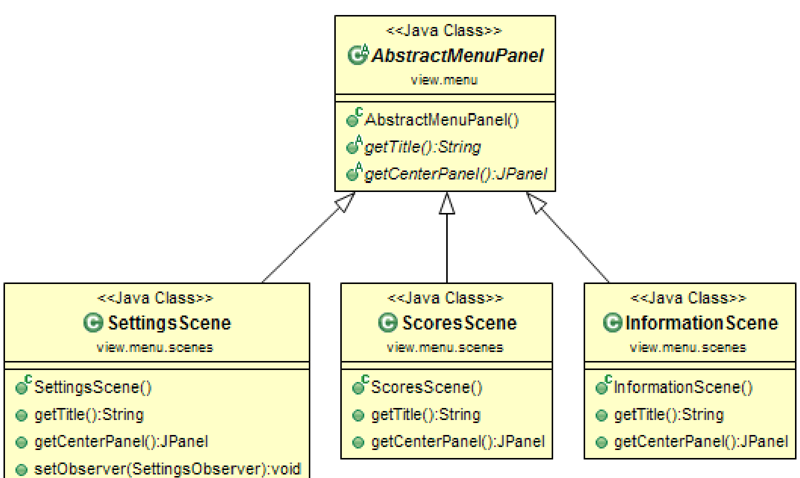
- *SettingsScene,* per l'accesso alle varie impostazioni
- *InformationScene,* per la lettura dei vari comandi, dei power-up disponibili e delle informazioni inerenti gli autori.

La scelta di realizzare un'enumerazione contenente le singole istanze delle varie schermate è stata perseguita per differenti ragioni. Un approccio di questo tipo, infatti, evita la creazione non necessaria di un nuovo pannello ogni qualvolta si presenti la necessità di mostrarlo e, dunque, permette di non andare incontro alla ridefinizione dei medesimi oggetti più volte. In aggiunta, oltre ad essere semanticamente chiaro, evita la ripetizione di codice e si dimostra essere particolarmente flessibile in vista di future modifiche.

Se dovesse rivelarsi necessaria l'aggiunta di una nuova schermata di menù, infatti, sarebbe sufficiente definire un nuovo elemento all'interno dell'enumerazione specificante l'istanza del pannello desiderato, per poterlo poi impiegare.

In questo modo, il Controller dedicato al menù (*MenuController*) è in grado di realizzare la sostituzione di una schermata semplicemente richiamando il metodo *replaceCard(),* definito all'interno dell'interfaccia *MenuFrame,* e indicando la vista che si intende mostrare direttamente mediante il relativo oggetto di *MenuCard.*

Al fine di evitare ogni forma di duplicazione di codice nell'implementazione dei vari pannelli è stato anche scelto di ricorrere all'uso del pattern Template Method.

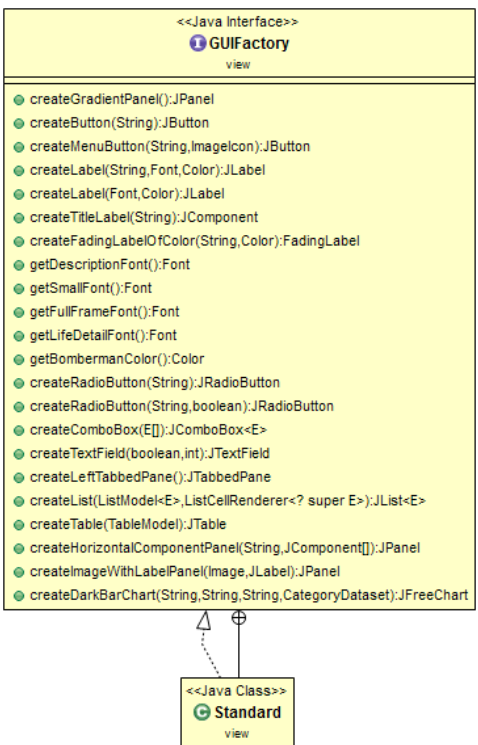


Dato che ogni vista di navigazione interna al menù è caratterizzata da un titolo, da un pannello principale rappresentante il contenuto in analisi e da un pulsante per il ritorno alla home, si è raccolto questo "comportamento" comune in una classe astratta, denominata *AbstractMenuPanel.*

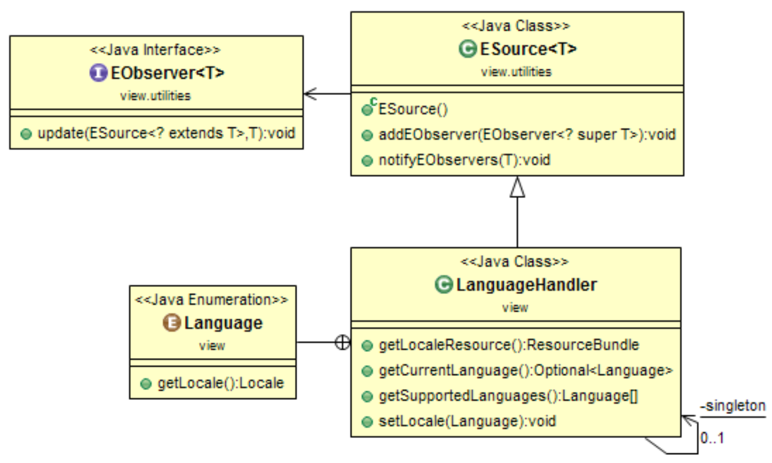
Di conseguenza, al momento della definizione di una nuova schermata risulta sufficiente specificare il titolo associato a quest'ultima e il pannello centrale (body) che si intende visualizzare.

Questa scelta progettuale ricade sempre nell'intento di favorire l'espandibilità dell'applicazione per future ed eventuali modifiche.

Per gestire al meglio la creazione dei componenti di GUI all'interno delle varie viste, si è scelto di adottare l'Abstract Factory Pattern.

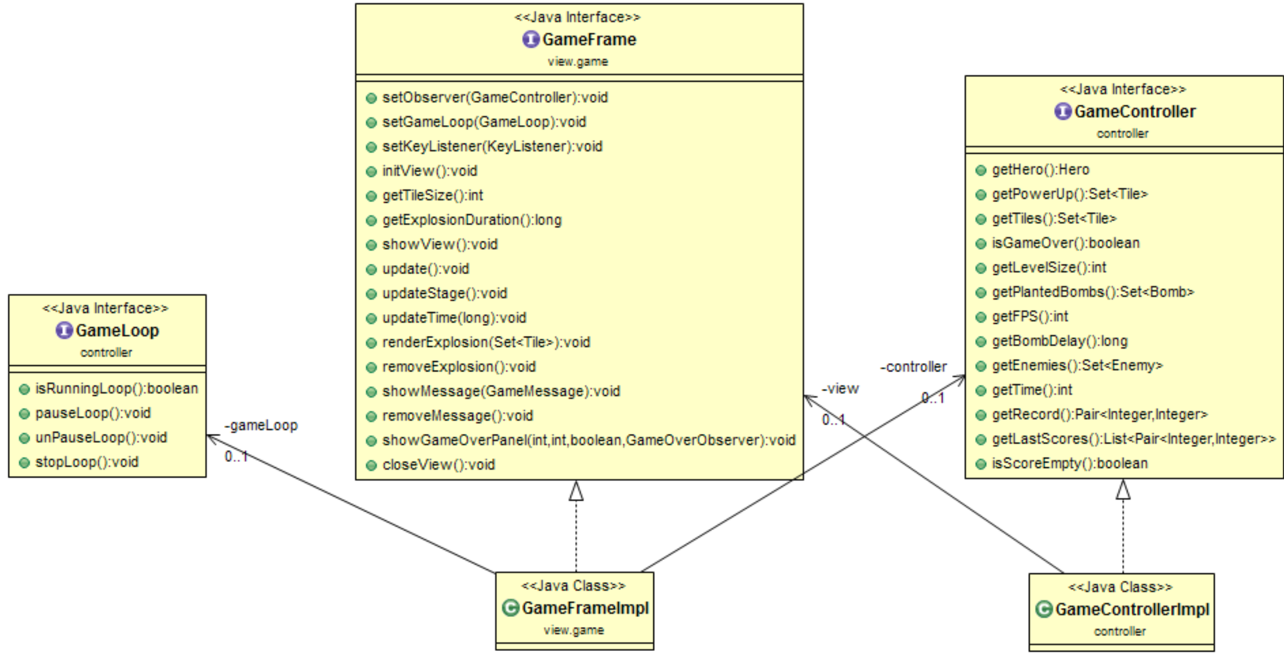


L'interfaccia *GUIFactory* fornisce i metodi factory finalizzati alla creazione e al ritorno degli oggetti. Essa viene poi specializzata, lasciando alle sottoclassi il compito di decidere quali classi istanziare e come.
Nell'applicazione si è fatto uso della sola implementazione *Standard*, ma se un domani si decidesse di modificare la resa grafica delle interfacce utente legate al gioco o di inserirne altre in aggiunta a quella già esistente sarebbe sufficiente modificare i relativi metodi in *Standard* o definire una nuova istanza di *GUIFactory*.
Un altro aspetto su cui ci si è voluti concentrare riguarda il supporto multilingua. È opportuno, infatti, definire la GUI una sola volta e cambiare dinamicamente le parti testuali a seconda della lingua di sistema o della preferenza all'interno dell'applicazione stessa.



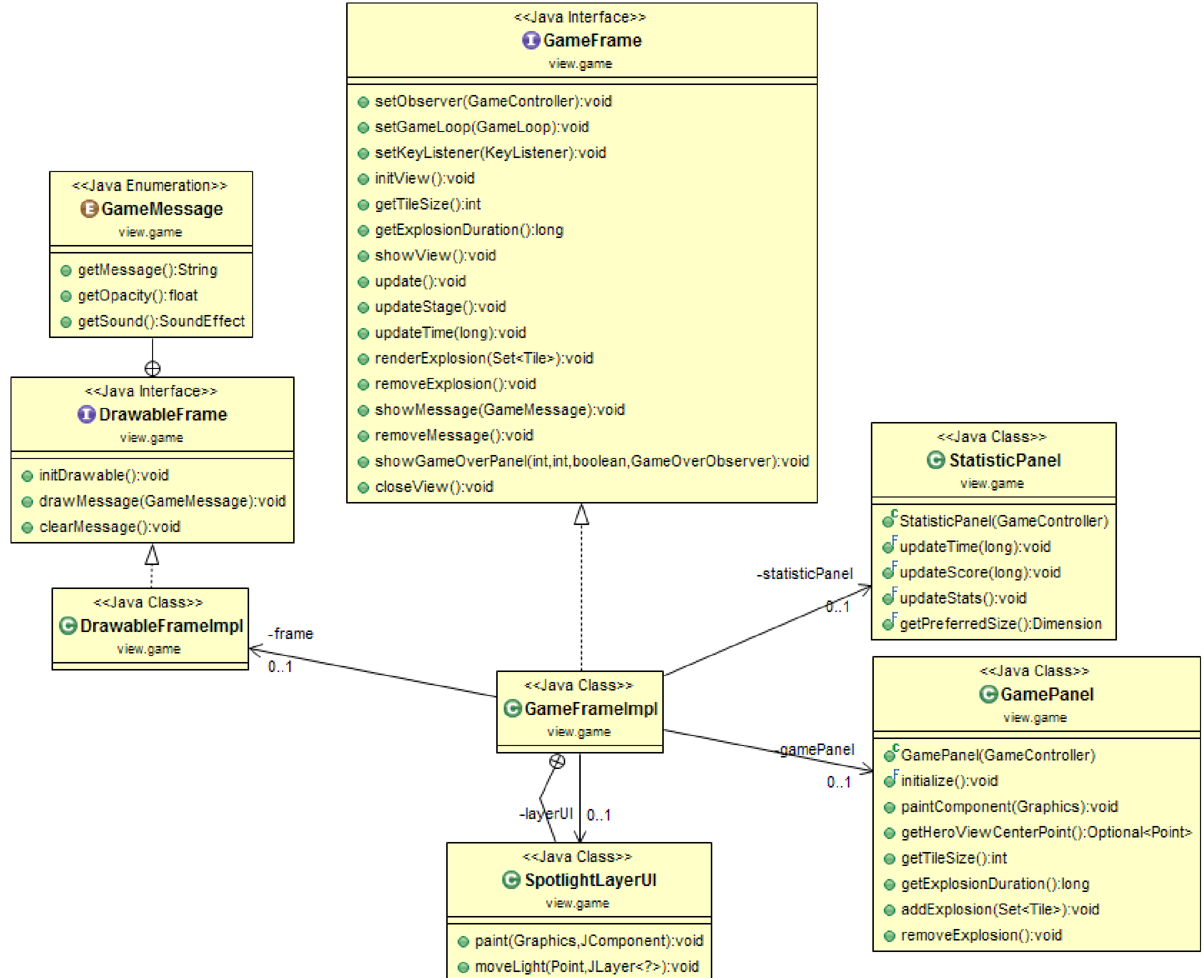
A tal fine si è deciso di creare un unico gestore: *LanguageHandler*.
Quest'ultimo, costruito mediante pattern Singleton (data la mancata necessita di possedere più istanze), definisce le lingue supportate dall'applicazione (mediante l'enumerazione *Language*) e permette la loro impostazione.
LanguageHandler fa uso di ResourceBundle e, dunque, si basa sull'architettura fornita da Java per l'internazionalizzazione. Le stringhe associate a ciascuna lingua sono opportunamente organizzate all'interno di file di supporto (properties files).
All'interno dell'applicazione, infatti, non sono presenti "stringhe magiche" o costanti dichiarate appositamente per scopi testuali, ma solo riferimenti (mediante valori di chiave) a tali file.
Anche questa scelta garantisce un'elevata flessibilità in quanto il supporto di una nuova lingua richiede semplicemente l'aggiunta del relativo file properties (coi nuovi valori associati alle chiavi) e l'inserimento dell'apposito oggetto all'interno dell'enumerazione.
Al fine di poter ricaricare il contenuto delle istanze relative le varie viste precedentemente descritte, al fronte di un cambio di preferenza per quanto concerne la lingua, è stato scelto di ricorrere al pattern Observer.
L'impostazione di una nuova lingua, infatti, fa sì che il *LanguageHandler* notifichi i vari pannelli ascoltatori di modo che questi possano aggiornare il loro contenuto testuale o comunque eseguire alcune istruzioni al sopraggiungere dell'avviso.

Di seguito, si analizza la parte progettuale dedicata alla gestione del gioco vero e proprio.



La View non è a conoscenza di alcun dettaglio implementativo relativo al Model e ogni informazione inerente esso è ottenuta attraverso il Controller e, in particolare, il *GameController*.
GameFrameImpl è la classe principale per gli aspetti di vista: essa presenta al suo interno la definizione del frame e dei pannelli in esso contenuti, dedicati al rendering del gioco. L'interfaccia *GameFrame*, implementata da *GameFrameImpl*, contiene i soli metodi che il *GameController* può richiamare per gestire la rappresentazione grafica sulla base dello stato del Model.

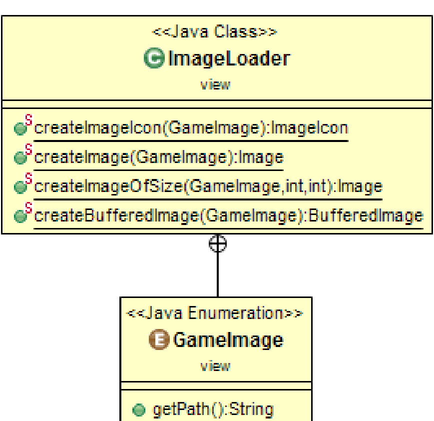
Allo stesso modo, i metodi definiti all’interno dell’interfaccia *GameController* sono gli unici su cui la View si può appoggiare per ottenere le informazioni necessarie per la componente di disegno.
Inoltre, la View è in grado di agire sul game loop gestito dal Controller mediante un’altra omonima interfaccia. In questo modo essa può decidere di sospendere o di riprendere il ciclo rappresentante il motore di gioco a fronte di eventi come, per esempio, la perdita o la riacquisizione del focus sulla finestra.



Il frame definito all’interno della classe *GameFrameImpl* estende da *DrawableFrameImpl*.
DrawableFrameImpl è una classe rappresentante, com’è possibile intuire dalla denominazione, un particolare tipo di frame su cui è possibile disegnare dei messaggi testuali.
L’elenco dei messaggi che possono essere visualizzati dalla suddetta classe è determinato dall’enumerazione *GameMessage*.
Considerando il fatto che ogni avviso (legato alla pausa, alla perdita di focus della schermata, al passaggio di livello ecc.) è contraddistinto da una struttura comune, si è ritenuto opportuno agire in questa maniera.
Se in futuro si volessero supportare ulteriori messaggi di gioco, infatti, sarebbe sufficiente specificare un nuovo elemento all’interno di *GameMessage*, evitando così ogni ripetizione di codice. Questo approccio costituisce anche una soluzione valida ed elegante per poter assegnare ad

ogni messaggio non solo una componente testuale, ma anche uno specifico livello di opacità dello sfondo e un suono legato alla sua visualizzazione.
Il frame si compone essenzialmente di due elementi fondamentali: *StatisticPanel* e *GamePanel*.
Il primo rappresenta il pannello contenente le varie informazioni di gioco (il tempo trascorso, il punteggio attuale e le statistiche correnti del giocatore sulla base dei potenziamenti raccolti); il secondo, invece, è il pannello su cui avviene il rendering del videogioco vero e proprio.
All'inizio di una partita, è il *GamePanel* a calcolare la dimensione ideale che una tile deve possedere, considerando la dimensione in numero di blocchi per lato della mappa, la risoluzione dello schermo e le proporzioni desiderate. Questa informazione viene poi usata dal Controller per inizializzare il Model al fine di una corretta gestione delle collisioni.
In questo modo l'applicazione si dimostra flessibile e in grado di adattarsi al form factor e, come accennato in precedenza, alla risoluzione di qualsiasi monitor. Anche per dar prova di questa flessibilità, si è scelto di cambiare la dimensione del livello ad ogni stage in maniera casuale.
Il *GameController*, durante il gioco, domanda l'update della parte grafica mediante lo specifico metodo fornito dall'interfaccia *GameFrame*. Quest'ultimo, oltre ad aggiornare le statistiche, ordina il repaint del *GamePanel* di modo che esso possa rappresentare lo stato attuale alla chiamata degli elementi e delle entità presenti nel livello di gioco. È sempre il *GameController* a domandare alla View l'aggiornamento d'informazioni quali il numero di secondi trascorsi dall'inizio della partita. Così facendo, è solo il Controller a stabilire lo stato di avanzamento del gioco.
La classe *SpotlightLayerUI*, infine, viene impiegata (qualora sia attiva la DarkMode) per il disegno di un layer di opacità e di una illuminazione radiale finalizzata alla creazione di un ambiente oscuro e di un unico effetto torcia in corrispondenza della posizione corrente del giocatore.

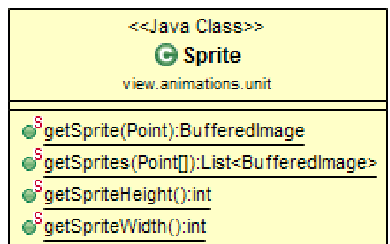
Nello sviluppo della componente grafica si è prestata particolare attenzione anche al caricamento delle varie risorse, assicurandosi che questo avvenga solo quando strettamente necessario (riducendo il numero di accessi in lettura).



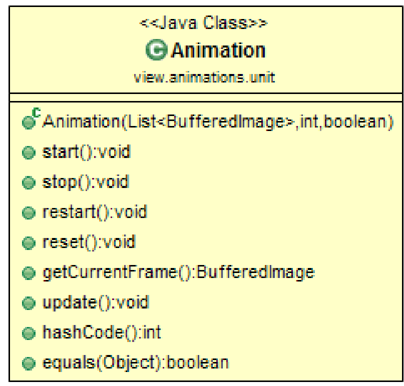
ImageLoader è una classe di utilità incaricata di mettere a disposizione i metodi necessari per accedere alle varie immagini dell'applicazione. Tutte le immagini di gioco che è possibile impiegare per la costruzione della GUI sono specificate all'interno dell'enumerazione *GameImage*.
Questa scelta è stata realizzata con l'intento di mantenere i percorsi legati alle suddette risorse in un unico punto del codice, in maniera tale da facilitare future modifiche o inserimenti.

Nello specifico *ImageLoader* permette di ottenere una risorsa in diversi modi per far fronte alle varie esigenze che si possono venire a presentare.

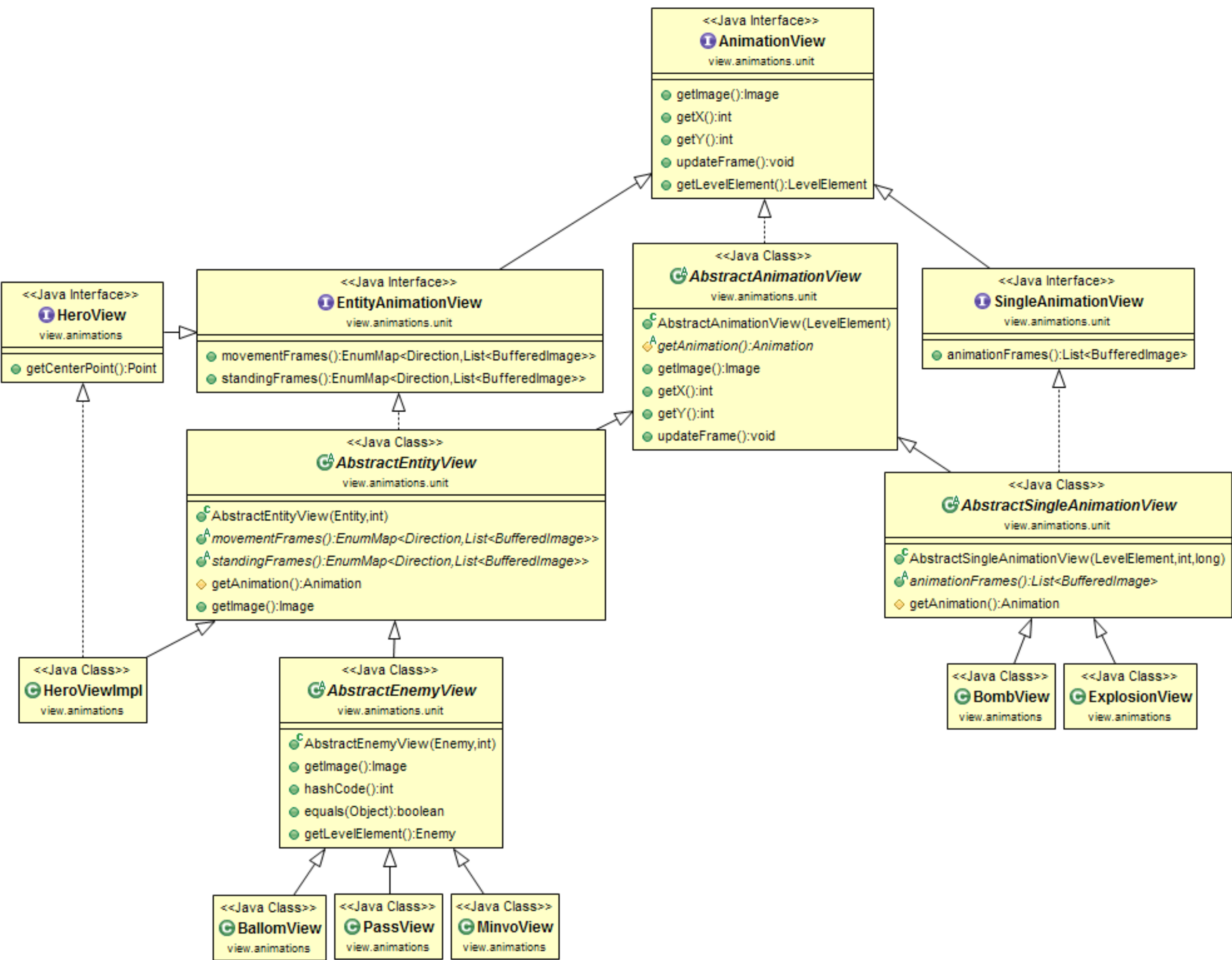
Un altro aspetto cruciale inerente la progettazione della View (che ha richiesto una maggior complessità architetturale) è quello riguardante la gestione delle animazioni.
Si è cercato fin da subito, infatti, di adottare uno stile grafico facente fede al videogioco originale e, dunque, non basato su semplici immagini statiche (ad eccezione delle tile e dei powerup che, invece, non necessitano di alcuna animazione).
Per la rappresentazione degli elementi di gioco dinamici, delle entità che compongono un livello e delle relative animazioni si è voluto ricorrere all'impiego di sprite sia per fattori prestazionali che per motivi di pura flessibilità ed estendibilità.
L'uso di uno sprite-sheet piuttosto che di singole gif animate, per esempio, riduce sensibilmente il numero di accessi / caricamenti delle risorse e conferisce un maggior controllo sulle varie animazioni in vista di aggiornamenti futuri. In particolare, l'impiego di uno sprite-sheet permette di accedere alle varie immagini necessarie per la creazione delle animazioni con un unico caricamento (dato che i vari elementi sono presenti sul medesimo file).



La classe di utilità *Sprite* mette a disposizione dei metodi per accedere facilmente ai singoli frame presenti all'interno dello sprite-sheet, mediante la specifica delle coordinate d'interesse.
Il suo funzionamento è legato alla conoscenza della dimensione (in pixel) assunta da ogni singolo sprite contenuto all'interno della risorsa. Se in futuro si volesse decidere di cambiare la dimensione delle immagini costituenti le varie animazioni, sarebbe sufficiente aggiornare il valore delle apposite costanti definite all'interno di questa classe.
La classe che invece si occupa di realizzare un'animazione vera e propria, a partire da un elenco di immagini, è *Animation*. Essa possiede differenti metodi per la gestione dell'animazione cui si riferisce, ma tra i più importanti vi è update(). Mediante esso è possibile richiedere l'avanzamento del frame all'interno dell'animazione stessa.



La progettazione della View, sotto questo punto di vista, è avvenuta con l'obiettivo di realizzare un completo disaccoppiamento tra gli elementi del Model e la loro rappresentazione grafica.
Si è scelto, quindi, di sviluppare specifiche classi interamente dedicate alla definizione dell'aspetto visivo (e quindi delle animazioni) di un'entità o di un componente di gioco.



I soggetti che vanno a popolare un livello possiedono caratteristiche ed esigenze grafiche diverse. Entità, quali eroe e nemici, possiedono più animazioni a loro associate, poiché possono sia muoversi sulla mappa in una determinata direzione che rimanere in posizione ferma.
Elementi, quali bombe ed esplosioni, invece, possiedono un'unica animazione che non necessita di essere ripetuta nel tempo. Nello specifico è richiesta una sola sequenza d'immagini per poter visualizzare il "trigger" di una bomba o l'effetto di una sua detonazione su un insieme di tile.
L'interfaccia *AnimationView* definisce i metodi necessari indipendentemente dal fatto che l'animazione faccia riferimento a un'entità o sia singola.
L'interfaccia *EntityAnimationView* estende *AnimationView*, definendo i metodi necessari per la gestione delle animazioni di entità in grado di muoversi su schermo.
L'interfaccia *SingleAnimationView* estende anch'essa *AnimationView*, ma è finalizzata alla definizione del metodo dedicato alla sola animazione disponibile nel caso di elementi di gioco che non siano entità.
La classe astratta *AbstractAnimationView* raccoglie il codice comune ad entrambe le implementazioni.

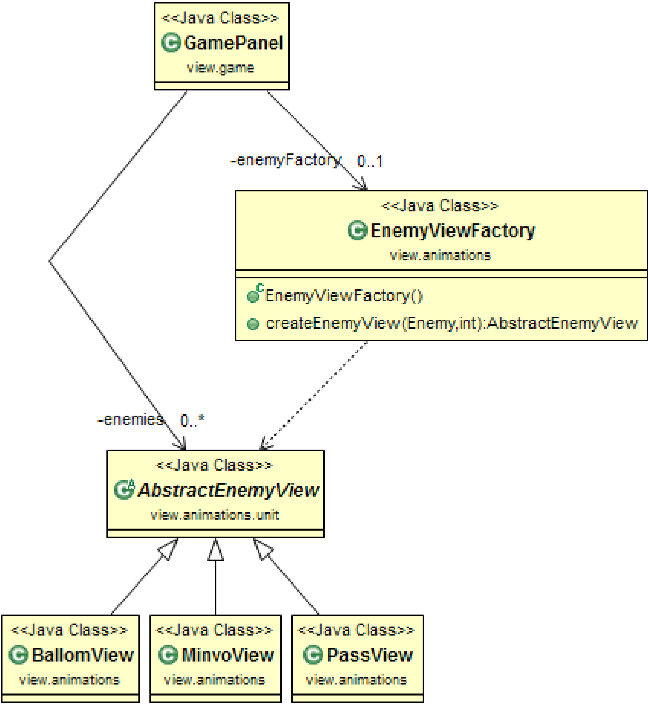
A partire da *AbstractEntityView* e da *AbstractSingleAnimationView* è possibile, mediante il pattern Template Method, specializzare il comportamento descritto (fornendo l'implementazione degli opportuni metodi astratti) per ottenere una specifica rappresentazione visiva inerente l'eroe, un tipo di nemico, una bomba, un'esplosione ecc.
Questa struttura apparentemente articolata rende in realtà l'applicazione molto flessibile e personalizzabile, garantendo anche un maggior grado di pulizia nel codice.
Con questa soluzione, infatti,

- è possibile creare una nuova animazione visiva per un nemico (o, in generale, per un'entità) semplicemente specificando le immagini da associare al movimento e alla posizione ferma del personaggio nelle varie direzioni
- è possibile creare una nuova animazione singola specificando soltanto la sequenza di immagini che la compongono
- è possibile impiegare la stessa rappresentazione visiva per nemici di tipo diverso (dato il disaccoppiamento dal Model), trattandola metaforicamente come un "costume" applicabile
- data la presenza di una classe per ogni rappresentazione visiva (grazie all'impiego di Template Method) è possibile personalizzare il comportamento di ciascuna di esse qualora dovesse rivelarsi necessario
- è possibile differenziare il rendering dei nemici da quello delle altre entità (come l'eroe). A tal proposito, infatti, è stato scelto di inserire attraverso *AbstractEnemyView* la visualizzazione (in basso a sinistra) del numero di vite rimanenti. Se in futuro si decidesse di mostrare al suo posto una barra dell'energia, sarebbero richieste soltanto poche modifiche.
- è possibile trattare il supporto di nuove animazioni con semplicità

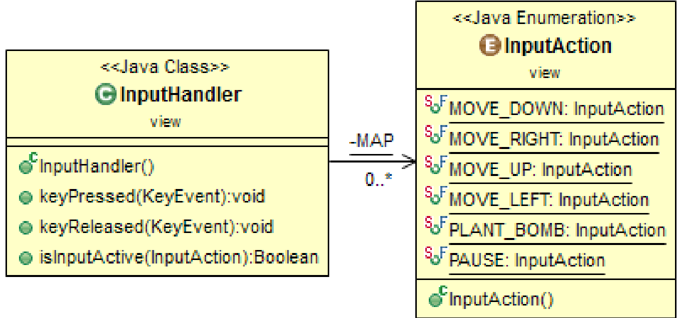
L'interfaccia *HeroView*, infine, aggiunge la definizione del metodo getCenterPoint() in maniera tale da poter risalire al punto centrale della rappresentazione grafica dell'eroe in cui disegnare lo spotlight qualora sia attiva la DarkMode.

In conclusione, ad ogni "tick" del *GameController*, il *GamePanel* crea le rappresentazioni visive di nuovi elementi di gioco, rimuove quelle non più necessarie sulla base dei dati forniti dal Controller stesso e aggiorna il frame legato alle animazioni attualmente attive su schermo. Successivamente, le disegna sfruttando le informazioni messe a disposizione grazie all'interfaccia *AnimationView*.

Per poter stabilire quale rappresentazione visiva associare ad ogni nemico fornito dal Controller, si è scelto di adottare il pattern creazionale Simple Factory.



La View si occupa anche di rilevare gli input durante una partita.



Ciò avviene all'interno di una classe separata da quella della visualizzazione. In questo modo, se in futuro si volesse supportare (oltre che la tastiera) anche una differente periferica quale il gamepad, sarebbe sufficiente andare a compiere modifiche soltanto all'interno di *InputHandler*. Per consentire una risposta più fluida ai comandi, inoltre, è stato scelto di differenziare la pressione di un tasto dal suo rilascio.
Le informazioni raccolte da questa classe vengono poi richieste dal Controller al fine di poter operare scelte riguardanti l'aggiornamento del Model o del gameloop.
Un ultimo compito della View riguarda la gestione del comparto sonoro mediante la classe *SoundEffect*.

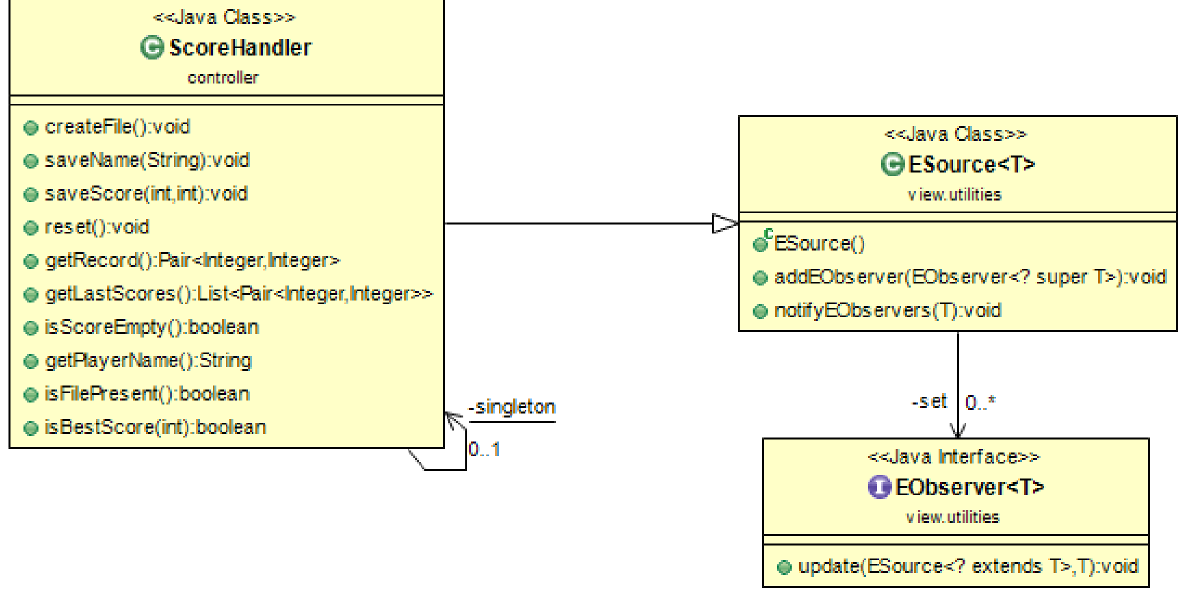
La gestione dei package rispecchia le scelte di progettazione viste sinora. Questa suddivisione è stata operata con l'intento di rendere il codice il più possibile navigabile ed accessibile.

- view
- view.animations
- view.animations.factory
- view.animations.unit
- view.game
- view.menu
- view.menu.components
- view.menu.scenes
- view.menu.scenes.panels
- view.utilities

Controller – *Giulia Giombini*

Il controller si occupa di coordinare le diverse chiamate tra View e Model.

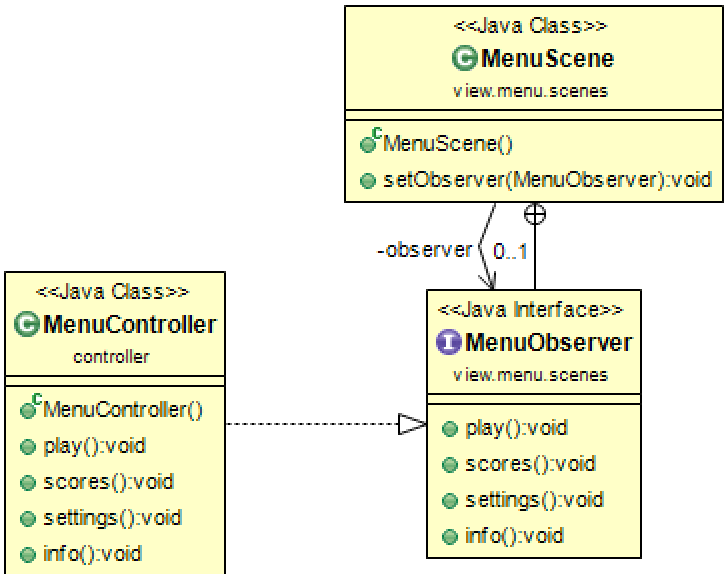
All'avvio dell'applicazione viene controllata la presenza di un file denominato "*Scores.dat*" all'interno della directory utente, utilizzato per il salvataggio locale del nome del giocatore, del record e degli ultimi rispettivi dieci punteggi. Per la creazione del file ho deciso di utilizzare il pattern Singleton per far si che ci sia una sola entità (istanza) che si occupi della gestione degli Scores. La classe *ScoreHandler* fa uso anche del pattern Observer per notificare alla View che sono avvenuti dei cambiamenti all'interno del file (nuovo record oppure salvataggio di un nuovo punteggio).



Verificare la presenza del file serve per riferire alla View quale scena visualizzare, se quella iniziale dove viene chiesto all'utente di inserire il proprio nome o quella del menù e settare di conseguenza l'observer.

La classe *MenuController* è la parte di controller dedicata al menù, che grazie proprio all'utilizzo del pattern Observer riesce a cambiare in modo semplice e veloce la schermata da mostrare indicando la vista che si intende mostrare direttamente tramite l'enumerazione *MenuCard*.

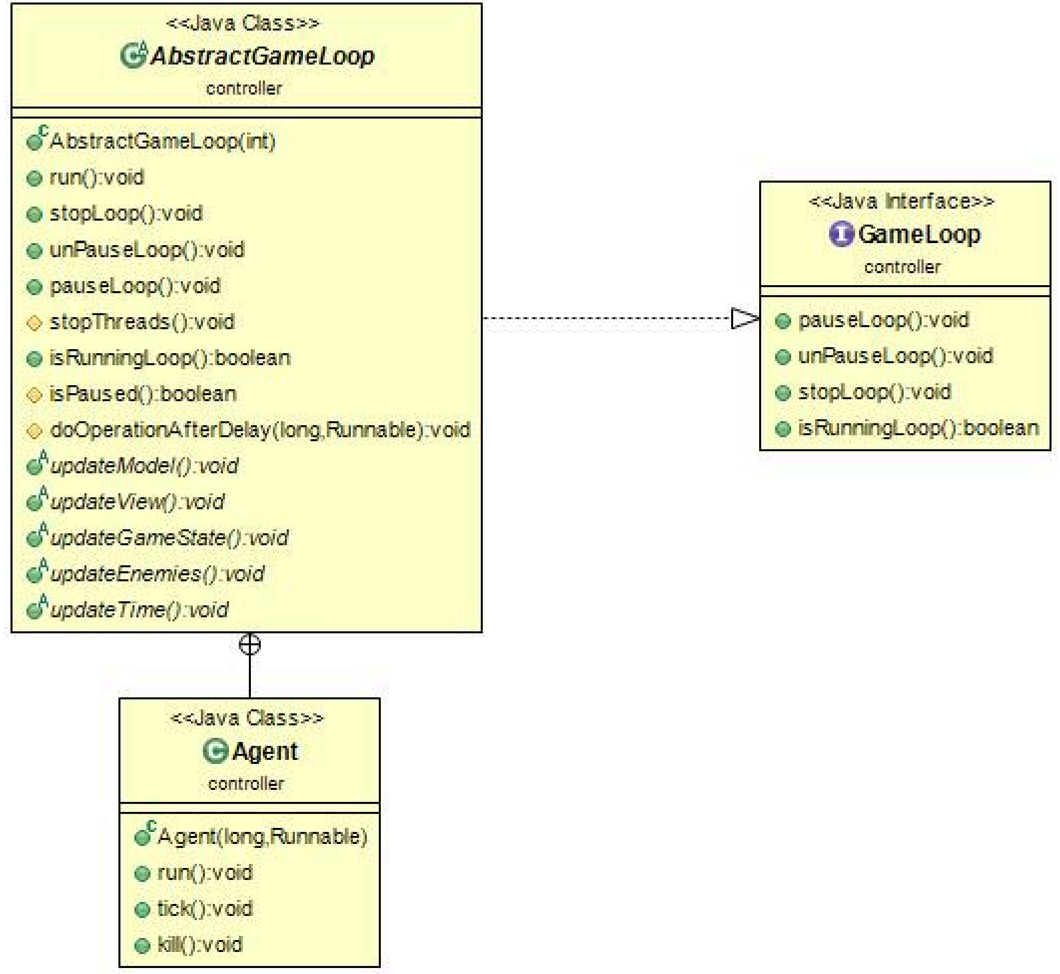
Di seguito il diagramma che illustra come il MenuController interagisce da "osservatore" alle diverse scene.



GameController

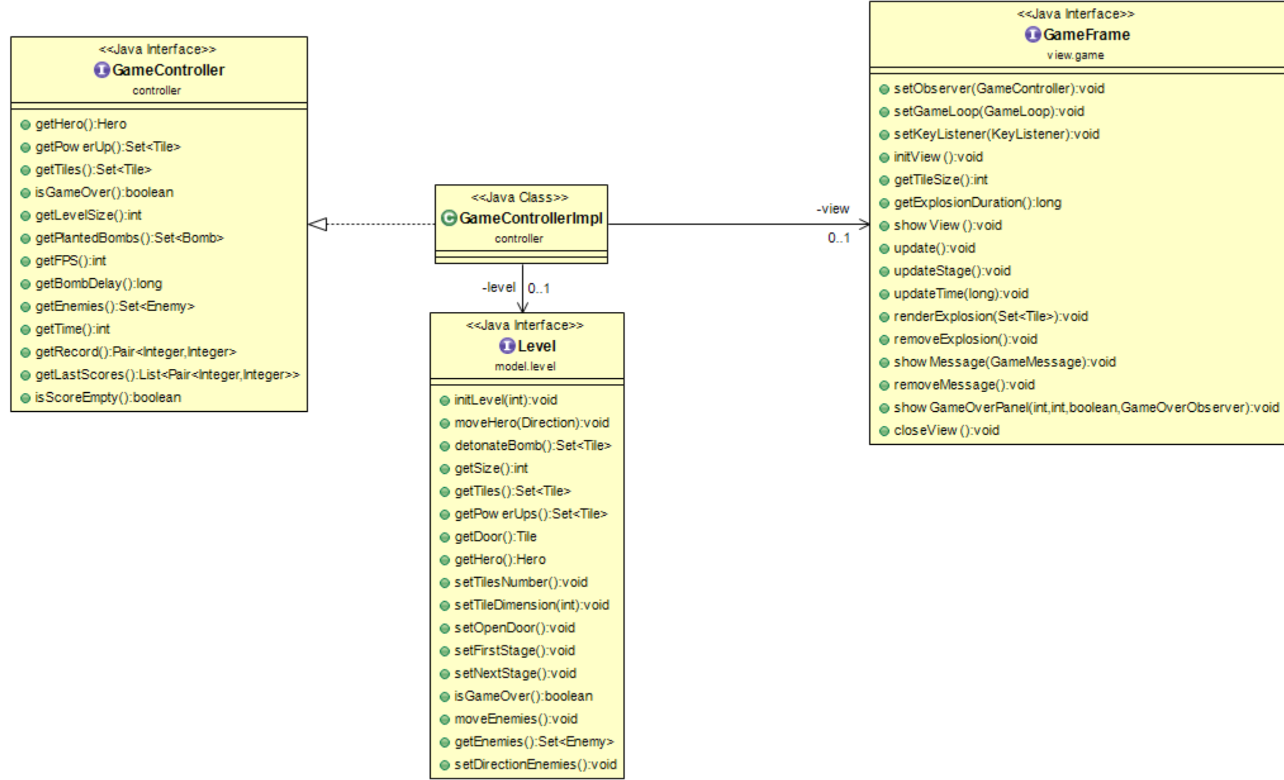
La classe *GameController* è la parte di controller che si occupa di gestire una partita di gioco. Esso inizializza Model e View e crea un'istanza della classe *AbstractGameLoop* che ha lo scopo di sincronizzare le operazioni di aggiornamento e disegno. Ho deciso di implementare il GameLoop tramite una classe astratta per rendere il codice più flessibile, infatti se si vuole utilizzare questa classe per sincronizzare altre situazioni basta semplicemente creare una nuova istanza e decidere che cosa implementare nei diversi metodi astratti. Il GameLoop è gestito attraverso un Thread che ad ogni frame aggiorna il Model e subito dopo si ordina alla View di far visualizzare la nuova scena. Se la parte di disegno richiede poco tempo, il GameLoop attende un numero di millisecondi tale da rendere costante la frequenza di frame per secondo. Il metodo *doOperationAfterDelay*() crea a sua volta un altro Thread che lavora in parallelo al GameLoop e viene utilizzato per compiere tutte quelle azioni che devono avvenire dopo un lasso tempo (ad esempio viene utilizzato per l'esplosione delle bombe, quando richiesto viene piantata una bomba e dopo un numero di secondi viene fatta esplodere, dal momento in cui esplode dico alla View di iniziare a disegnare l'animazione dell'esplosione e viene riutilizzato il metodo per dire di smettere di disegnare l'animazione sempre per un certo tempo).

Tramite altri metodi del *AbstractGameLoop* è possibile mettere in pausa, riprendere o terminare il gioco.



I metodi astratti all'interno del *GameController* sono stati così utilizzati:

- *UpdateModel*(): vengono fatti diversi controlli per apportare le conseguenti modifiche a livello di Model spesso interpellando la classe *InputHandler*;
- *UpdateView*(): viene ordinato alla View di ridisegnare la scena di gioco;
- *UpdateGameState*(): interrogando nuovamente la classe InputHandler viene modificato lo stato del gioco (pausa, fine pausa, perdita), in caso di perdita viene salvato su file il punteggio ed il rispettivo tempo della partita, si chiede alla View di cambiare scena e la classe *GameController* viene impostata come observer di tale schermata dando la possibilità di rigiocare oppure di uscire e tornare al menù principale;
- *UpdateEnemies*(): questo metodo viene chiamato all'interno del GameLoop ogni secondo e viene utilizzato per aggiornare la direzione di movimento di alcuni tipi di nemici;
- *UpdateTime*(): anche questo metodo viene chiamato ogni secondo ed aggiorna il tempo della partita.



Essendo la parte di controller composta di molti sorgenti, ho suddiviso .controller in sotto-package, come illustrato:

- controller
- controller.test
- controller.utilities

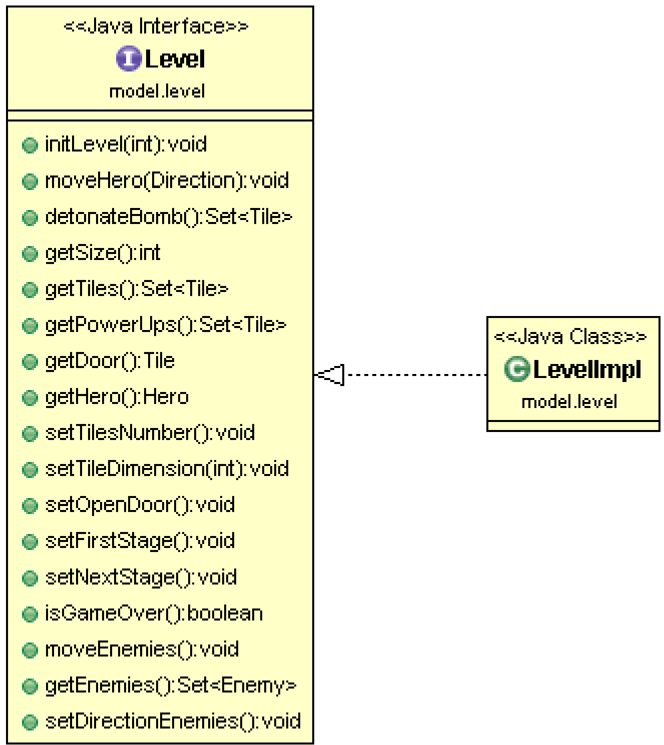
Model – *Sofia Rossi*

Il Level si occupa di gestire interamente l'aspetto logico del gioco, ovvero tutte le entità e la relativa mappa. Rappresenta quindi il livello a cui l'utente sta giocando. In particolare, quindi, l'esecuzione degli input passati dal Controller e dei relativi controlli, principalmente le collisioni tra entità e entità-mappa.

Il Level viene creato e correttamente inizializzato dal Controller nel momento in cui l'utente decide di iniziare una partita. La costruzione del livello però è divisa in due fasi:

1. Nel costruttore viene fissata la grandezza della mappa di gioco;
2. La chiamata al metodo *initLevel(),* da parte del Controller, configura correttamente le entità e la mappa stessa.

La difficoltà è legata solamente al progredire del numero di livello, risulta essere quindi un aspetto interno. Specificatamente con difficoltà si intende l'incremento di attacco e vita degli *Enemy*, mentre l'Hero giova di questo rafforzamento solamente se nel livello precedente ha acquisito benefici di specifici *PowerUp*.

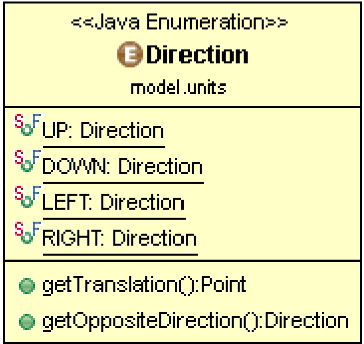


Ciclicamente il *GameLoop* si occupa di richiamare i metodi per il movimento delle entità non statiche, ovvero l'*Hero* e gli *Enemy*. In particolare verrà quindi effettuata la chiamata al metodo *moveHero(Direction),* il cui parametro identifica la direzione scelta dall'utente, e *moveEnemies().* Per una corretta gestione del movimento degli *Enemy*, ad un certo intervallo di tempo il *GameLoop* richiama il metodo *setDirectionEnemies().* Successivamente, nel momento dell'aggiornamento

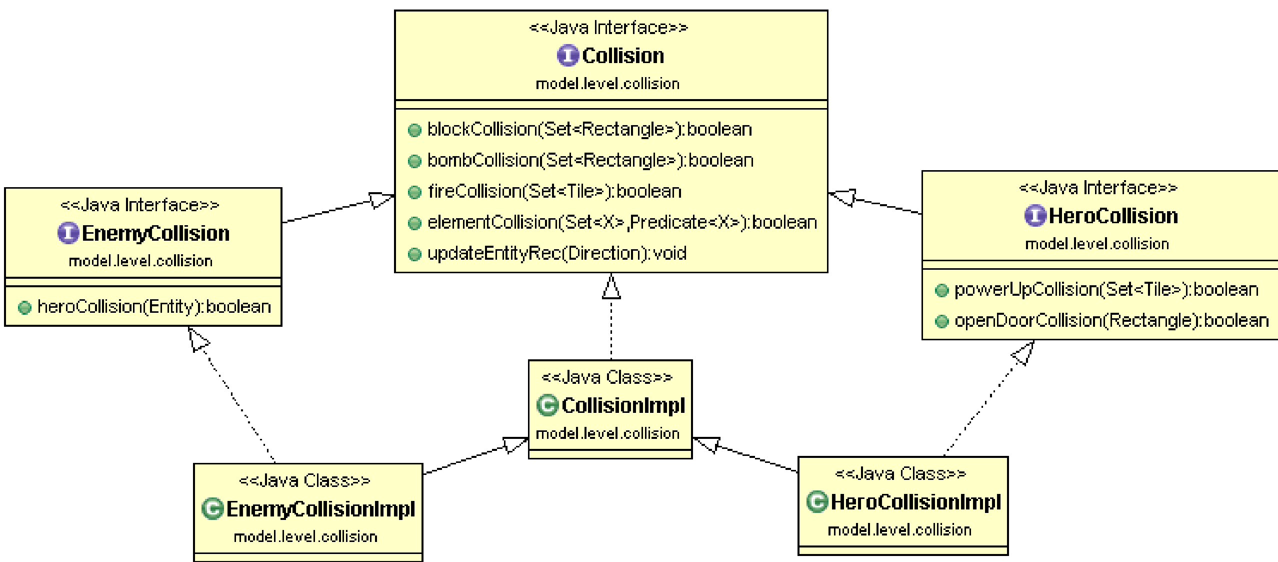
della View, vengono richiamati *getTiles(), getPowerUps(), getEnemies() e getHero()* per una corretta visione dello stato del gioco.

Collision and Movement

Il movimento è gestito attraverso l'Enum *Direction* che, oltre ad identificare le possibili direzioni, permette di gestire in modo chiaro e conciso l'aggiornamento della posizione delle entità.



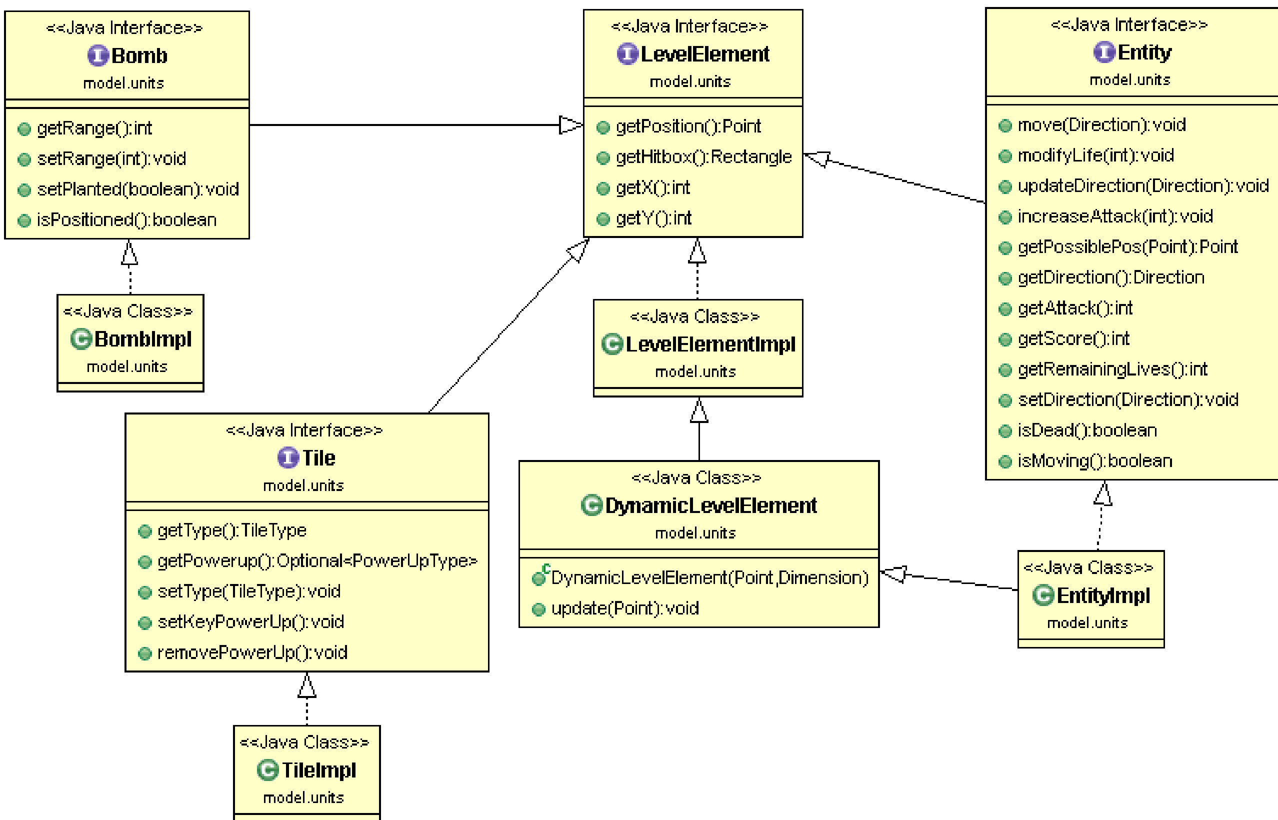
Prima di poter aggiornare la posizione delle entità, vengono verificate eventuali collisioni con gli altri elementi di gioco: sia quelli statici che dinamici. Tutto ciò è gestito tramite l'oggetto *Collision*, correttamente esteso per una specifica gestione. Infatti tutto ciò che è utile per il controllo del movimento *Enemy/Hero*, come collisioni con blocchi della mappa, bombe e fuoco, è stato mantenuto comune. In seguito le specializzazioni *EnemyCollision* e *HeroCollision* implementano metodi aggiuntivi che sono specifici per il tipo di entità. Le collisioni aventi conseguenze sulle entità sono *fireCollsion, powerUpCollision, heroCollision,* quali diminuzione di vite o applicazione dei relativi effetti dei PowerUp.



Le collisioni possono quindi determinare sia un cambiamento nella composizione della mappa, sia lo sviluppo del gioco stesso. Infatti sono proprio queste la causa principale della fine del livello, in quanto determinano la morte dell'*Hero*, ma anche della diminuzione della difficoltà, perché appunto comportano l'eliminazione degli *Enemy*.

Game's Elements

Gli elementi caratteristici del gioco sono stati implementati partendo da una interfaccia *LevelElement*, la quale definisce metodi generali e comuni ad ogni elemento. Questa interfaccia è stata poi correttamente estesa secondo una divisione ben precisa: *elementi statici e dinamici*.



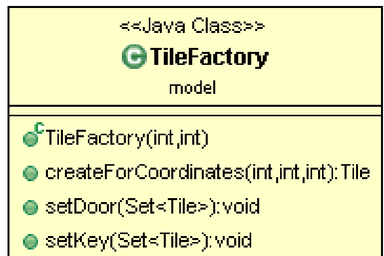
La suddivisione è caratterizzata dalla classe *DynamicLevelElement* e ciò che la caratterizza è appunto il metodo *update(Point),* il quale permette quindi un aggiornamento dei dati dell'elemento, specificatamente della posizione. Per gli elementi statici non è stato necessario aggiungere metodi specifici in quanto le uniche operazioni che è possibile fare su di essi sono semplicemente dei getter (*getPosition(), getHitbox(), getX() e getY()*) utili per assicurare un corretto svolgimento del gioco e relativa visualizzazione.

Static Elements

Come si può notare nella suddivisione generale degli elementi, quelli statici sono caratterizzati dalle interfacce *Tile* e *Bomb*. Per statico si intende non in movimento. Infatti, per quanto riguarda le *Tile* è comunque necessario applicarvi delle modifiche, quali *setType(TileType)* e *removePowerUp(),* determinate dallo svolgimento del gioco.

Tile rappresenta i diversi elementi di cui si compone la mappa di gioco. La loro creazione è gestita dalla classe *TileFactory*, di cui il relativo oggetto è creato all'interno della classe *LevelImpl*, perchè è la classe che gestisce la creazione dei vari elementi tra cui anche la mappa. Nello specifico è il metodo createForCoordinates*(int, int, int)* che restituisce una *Tile* inizializzata, cioè con una corretta posizione, dimensione, *TileType*, *PowerUpType*. Viene applicata una distinzione per l'impostazione

degli elementi che permettono di accedere al livello successivo: la porta e la chiave. La porta è realizzata associando il relativo tipo ad una *Tile*, mentre la chiave viene impostata come *PowerUp*.



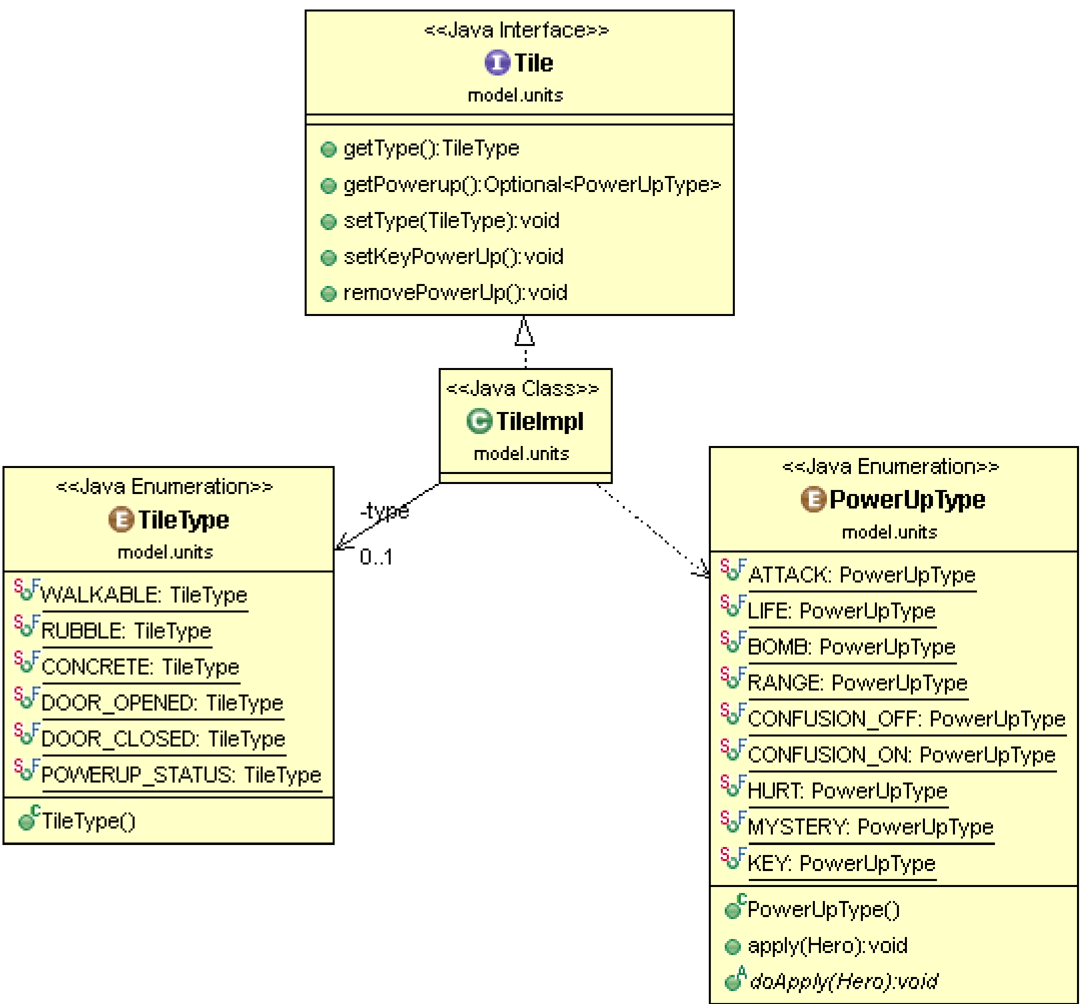
Le caratteristiche principali di una *Tile* risultano quindi essere il *TileType*, cioè il tipo, e il *PowerUpType*, ovvero il *PowerUp* che possono nascondere al loro interno.

Per determinare il tipo è stato deciso di implementare una Enum, *TileType*, in quanto in questo modo si può gestire correttamente il dominio dei tipi. Inoltre anche eventuali modifiche o estensioni sono facilmente applicabili.

Il tipo più rilevante è sicuramente RUBBLE: solamente se una *Tile* possiede questo tipo può eventualmente nascondere un *PowerUpType*. Questa decisione deriva dal fatto che la loro visualizzazione e partecipazione al gioco inizia solamente nel momento in cui il blocco RUBBLE che ne possiede uno esplode.

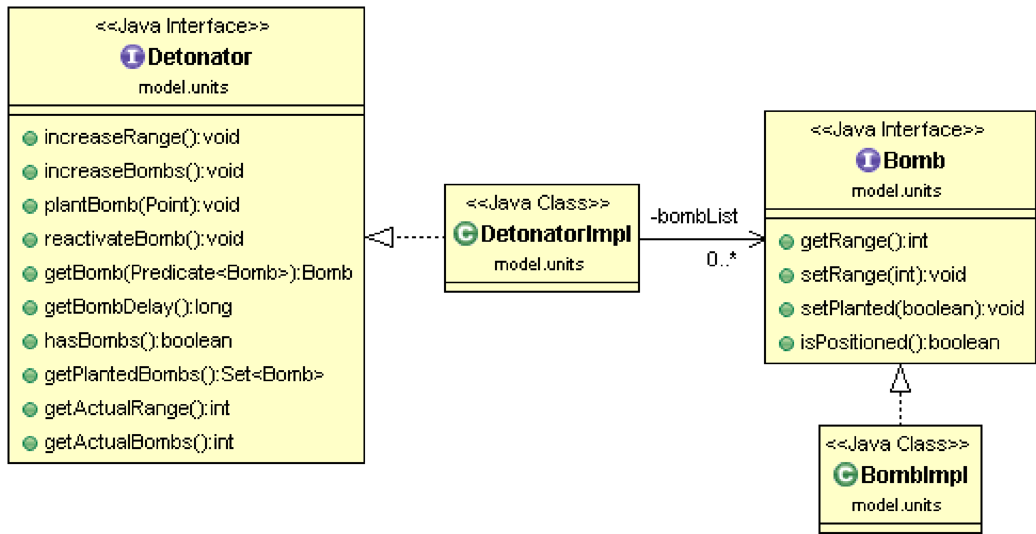
Anche per i *PowerUp* è stata realizzata una Enum che presenta appunto i *PowerUp* principali, tra cui *LIFE*, aumento vita, *BOMB*, incremento bombe, *RANGE*, aumento raggio bomba, *ATTACK*, aumento attacco ed infine *KEY*, elemento necessario per accedere al livello successivo. Al fine di rendere il gioco più innovativo, è stato scelto di inserire anche *PowerUp* caratterizzati da poteri svantaggiosi: *HURT*, decremento vita e *CONFISION_ON*, il quale inverte la direzione del movimento scelto dal giocatore.

Tra i diversi tipi emerge anche *MYSTERY*: questo tipo ne nasconde un altro al suo interno che il giocatore non è in grado di visualizzare. L'introduzione di questo *PowerUp* è appunto stata realizzata per questa ragione: non potendo visualizzare il tipo che realmente nasconde, l'utente deciderà, eventualmente rischiando, di acquisirne i poteri. L'Enum *PowerUpType* risulta essere anche utile per la gestione dell'applicazione degli effetti sull'entità Hero. Infatti, attraverso l'implementazione del metodo astratto *doApply(Hero),* nel momento in cui si verifica la collisone Hero – *PowerUp*, si è in grado di estrarre il tipo di *PowerUp* e gestire le relative conseguenze. Tutto ciò è ben estendibile per revisioni o modifiche del gioco.



Bomb costituisce l'interfaccia per quello che può essere considerato l'elemento principale del gioco: la bomba. La sua gestione è realizzata tramite l'interfaccia *Detonator*. L'Hero, infatti, possiede questo oggetto ed è grazie a questo che ha la facoltà di piantare delle bombe per eliminare nemici o andare alla scoperta di *PowerUp*. La scelta di avere *Bomb* come elemento statico del gioco deriva dal fatto che, una volta posizionata, una bomba non fa altro che esplodere, per cui la sua posizione risulta essere fissa a priori.

Oltre alla posizione, una caratteristica rilevante riguardante la bomba è il suo raggio. Quest'ultimo, insieme al numero di bombe piantabili contemporaneamente dall'Hero, è gestito al momento della creazione dal'oggetto *Detonator*. La logica del funzionamento delle *Bomb* nel gioco infatti dipende strettamente da *Detonator*, il quale identifica il punto di riferimento per conoscere informazioni e appunto gestire le *Bomb*, anche dal punto di vista dei *PowerUp* relativi a questo elemento. I metodi *increaseRange()* ed *increaseBombs()* permettono di applicare l'effetto dei *PowerUp*, mentre *getActualRange(), getActualBombs()* e *getPlantedBombs()* consentono sia una corretta visualizzazione delle caratteristiche delle *Bomb*, così come l'insieme di quelle che fanno parte del gioco in quello specifico momento, informazione utile anche per un esatto svolgimento.



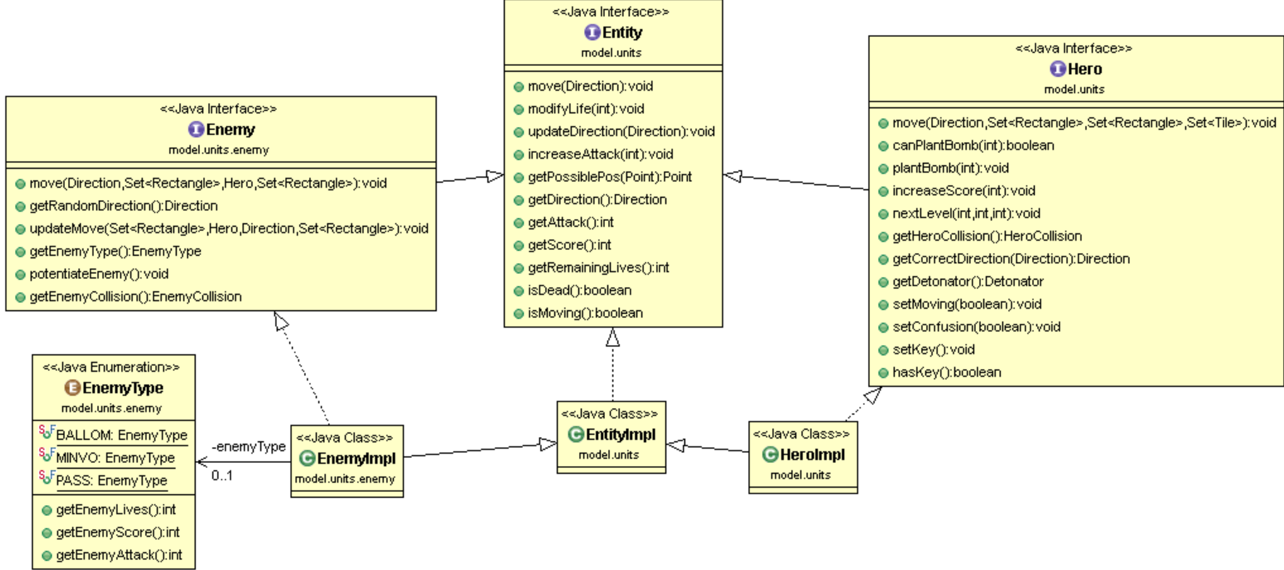
Dynamic Elements

Gli elementi dinamici sono quelli che hanno facoltà di movimento all'interno della mappa di gioco: *l'Hero* e gli *Enemy*. Per gestire questa categoria è stata creata una interfaccia *Entity* che riassume tutto ciò che questi personaggi hanno in comune, le varie specifiche sono state implementate con una corretta estensione: le interfacce *Hero* e *Enemy*.

Nell'interfaccia *Entity* viene gestito il movimento attraverso i metodi *move(Direction)* e *getPossiblePos()*. In questo però non vengono considerate le eventuali collisioni con gli altri elementi di gioco, le quali sono introdotte nello specifico nella classe *HeroImpl* e *EnemyImpl*.

Altri aspetti in comune ad entrambi i tipi di protagonisti sono l'attacco, il punteggio ed il numero di vite per i quali sono stati introdotti metodi per applicarvi modifiche e, nel caso delle vite, per verificare se l'entità è morta. Il metodo *isDead()* è particolarmente significativo: attraverso la verifica dell'eventuale morte dell'Hero il GameLoop viene stoppato, mentre per quanto riguarda i nermici, questi vengono eliminati apportando modifiche alla difficoltà del gioco.

I metodi *getAttack(), getScore()* e *getRemainingLife()* sono invece utili al fine di una corretta visualizzazione dello stato del gioco. Precisamente la View li utilizza per gestire il relativo pannello attraverso il quale il giocatore è in grado di sapere in tempo reale i dati specifici dell'Hero.



Hero

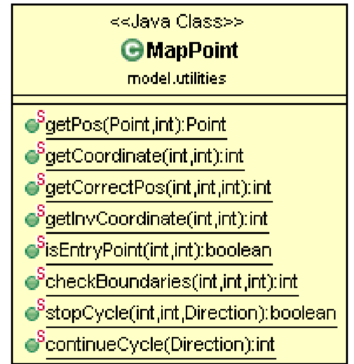
Hero è l'interfaccia che gestisce il personaggio principale del gioco, il cosiddetto *"Bombarolo"*. Oltre alla presenza del metodo *move*(), attraverso il quale viene gestito nello specifico il movimento considerando anche le eventuali collisioni, sono rilevanti e fondamentali per il funzionamento del gioco i metodi *canPlantBomb()* e *plantBomb(Point)*. Sono appunto questi i metodi che contraddistinguono questo personaggio rendendolo il protagonista. Come precedentemente detto, l'Hero possiede un *Detonator*. Grazie alla cooperazione di questo oggetto ed i metodi precedentemente citati viene realizzato l'aspetto principale del gioco: il posizionamento delle bombe e la relativa esplosione.
Inoltre il protagonista è l'unico personaggio che subisce gli effetti dei *PowerUp* presenti nel gioco. I metodi *setKey()* e *setConfusion()* realizzano l'applicazione di due di essi. Direttamente collegato al *setConfusion()* vi è il *getCorrectDirection()*, il quale permette una corretta gestione del movimento.
Il metodo *nextLevel()* gestisce l'aggiornamento di valori quali punteggio, vita e attacco per il livello successivo al quale si accede solamente se l'Hero possiede la chiave, verificato attraverso *hasKey(),* e la collisione con la porta aperta, operazioni compiute dal *GameLoop*.

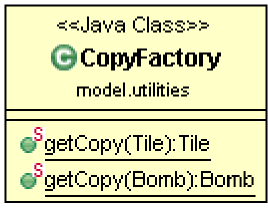
Enemies

La classe *EnemyImpl* estende la classe *EntityImpl* ricavando molte cose in comune con l'eroe. Ho implementato tre tipi di nemici, differenti per vita, attacco, punteggio e movimento descritti attraverso l'enumerazione *EnemyType*.
Due tipologie di nemici cambiano direzione scegliendola random nel momento in cui si imbattono contro qualcosa, mentre a uno soltanto grazie al GameLoop viene modificata la direzione ogni secondo. Ho implementato anche la classe *EnemyCollision* dove controllo se il nemico si scontra con l'Hero.

Utilities

Nel package model.utilities vi sono le classi *MapPoint* e *CopyFactory*. *MapPoint* è stata realizzata al fine di fornire metodi per agire e creare la mappa, precisamente sulla posizione degli elementi.





CopyFactory invece ha lo scopo di restituire copie difensive per *Tile* e *Bomb*, in modo tale che né il Controller né la View possano modificare questi oggetti comportando modifiche nel Model.

La realizzazione del model è stata strutturata all’interno di package che permettono una chiara e corretta navigabilità.

- model
- model.level
- model.level.collision
- model.test
- model.units
- model.units.enemy
- model.utilities

Sviluppo

3.1 Testing automatizzato

Sono stati creati alcuni test JUnit per verificare il funzionamento delle seguenti classi:

- controller.GameLoop: creazione e terminazione, pausa e ripresa del gioco, corretta precisione dei frame;
- model.units.Bomb: verifica del corretto posizionamento della bomba in base alle coordinate dell'Hero;
- model.level.collision.Collision: test della collisione tra eroe-blocchi, nemico-blocchi, eroe-bomba, nemico-bomba, eroe-powerUp, eroe-nemico;
- model.utilities.CopyFactory: accertamento della giusta restituzione delle copie di tile e bombe.

Per verificarne il corretto funzionamento, l'applicazione è stata manualmente testata nei seguenti sistemi operativi: Windows 7, Windows 8.1, Windows 10, Mac OS X 10.10, Debian 8 Jessie (64bit).

3.2 Metodologia di lavoro

Al fine di consentire uno sviluppo indipendente e parallelo, il gruppo ha inizialmente discusso la suddivisione in classi delle componenti principali dell'applicazione. Così facendo sono stati stabiliti i metodi delle interfacce da implementare per realizzare la comunicazione all'interno dell'architettura scelta: l'MVC. A ogni membro, dunque, è stata assegnata la realizzazione di specifiche classi con l'intento di dividere equamente i vari carichi di lavoro.
Si è adottato un processo di progettazione top-down con raffinamento successivo (processo "a spirale"). Conseguentemente le funzionalità del software sono state realizzate in maniera graduale, impostando progressivamente le relative scadenze.
Il lavoro in parallelo è avvenuto mediante l'impiego di Mercurial come DVCS. Avvalendosi di Bitbucket, i membri hanno caricato frequentemente le proprie modifiche locali al fine di renderle disponibili agli altri.

- ***Giacomo Frisoni***
  Ho realizzato in autonomia tutte le classi e le interfacce presenti all'interno del package view.
  Inoltre, mi sono occupato della componente grafica dell'applicazione, dei brani di gioco e dei file contenenti le traduzioni necessarie per la messa a disposizione del supporto multi-lingua.
  La scelta di alcune risorse è stata effettuata in gruppo con l'intento di privilegiare quelle ritenute più consone per il gioco.

Nella parte iniziale di analisi vi è stato un confronto con Giombini allo scopo di convenire sull'uso delle interfacce e stabilire i criteri di comunicazione fra la View e il Controller, in maniera tale da procedere indipendentemente allo sviluppo.

- ***Giulia Giombini***
  Mi sono occupata della realizzazione di tutte le classi e interfacce all'interno del package controller e model.units.enemy, dell'interfaccia EnemyCollision e la rispettiva implementazione, dei metodi initEnemies, createEnemies, moveEnemies, setDirectionEnemies e checkCollisionWithExplosionBomb all'interno della classe model.level.LevelImpl e di tutti i test nei seguenti package: contoller.test e model.test. Mi sono dovuta confrontare spesso con i miei compagni per ritoccare le varie interfacce fungendo da intermediaria soprattutto con Rossi quando ho iniziato la creazione dei nemici, dato che le mie classi ne estendevano altre da lei implementate.

- ***Sofia Rossi***
  Personalmente mi sono occupata dello sviluppo di tutte le classi e le interfacce all'interno del package model, ad eccezione di model.units.enemy e model.test che, come accordato, sono state sviluppate da Giombini per equilibrare il carico di lavoro. Ho implementato il model in maniera tale che fosse il più possibile indipendente dagli altri componenti, facilmente riadattabile per eventuali modifiche e per rispettare la corretta suddivisione dell'architettura scelta, ovvero il modello MVC. Il confronto con gli altri membri è stato soprattutto relativo al funzionamento del gioco al fine di implementare una corretta logica, in quanto, prima di concordare la proposta di progetto, non conoscevo il gioco stesso.
  Ho lavorato quindi in maniera totalmente indipendente accordandomi principalmente con Frisoni per la corretta visualizzazione degli elementi, ad esempio su quali di essi applicare il concetto di copia difensiva, e con Giombini per spiegare il funzionamento dei principali metodi in modo tale che potesse implementare la parte del Controller senza eventuali incomprensioni.

3.3 Note di sviluppo

La realizzazione della classe SpotlightLayerUI è avvenuta basandosi su un esempio di possibile impiego di JLayer disponibile in un tutorial Oracle:
http://docs.oracle.com/javase/tutorial/uiswing/misc/jlayer.html.
Al fine di realizzare una GUI il più possibile scalabile, si è fatto uso di una classe per la realizzazione d'icone "stretchable", in grado di adattarsi e ridimensionarsi automaticamente sulla base dell'area disponibile in un determinato componente.
A questo scopo si è scelto di ricorrere all'impiego di *StretchIcon* (una classe individuata in rete e disponibile al seguente link: https://tips4java.wordpress.com/2012/03/31/stretch-icon/).

Per una ottimale realizzazione del GameLoop si sono seguiti i ragionamenti descritti nel seguente articolo: http://entropyinteractive.com/2011/02/game-engine-design-the-game-loop/.

Le parti di codice sopraccitate sono state in ogni caso adeguate agli standard e allo stile del progetto.

Ove possibile si è cercato di ricorrere all'impiego di librerie esterne con l'intento di rendere più efficace il lavoro svolto. Per la creazione dei grafici si è fatto uso della libreria *JFreeChart*.
Al fine di poter usufruire di oggetti Optional serializzabili e dunque memorizzabili su file, invece, ci si è avvalsi della nota libreria open-source Google Guava.
Per l'analisi e il superamento di alcune problematiche incontrate durante lo sviluppo, si sono consultate principalmente le seguenti fonti:

- Documentazione Oracle
- StackOverflow
- Blog di settore

Commenti finali

4.1 Autovalutazione

Il processo di sviluppo è stato lineare. Il team si è trovato bene nel lavoro di gruppo e i membri sono riusciti a collaborare e a interagire facilmente.

- **Giacomo Frisoni**
  Sono molto soddisfatto del lavoro compiuto e ritengo che l'impegno dedicato alla progettazione di questa applicazione abbia portato a un buon risultato.
  Mi sono concentrato fin da subito sulla realizzazione di un codice flessibile ed estendibile, che mi permettesse un'efficace gestione delle schermate e delle animazioni.
  Il frequente ricorso a pattern studiati a lezione, inoltre, ha favorito la manutenibilità del sorgente e l'assenza di ripetizioni al suo interno. Il progetto, dunque, ben si presta a prossimi miglioramenti.
  La scelta di Java 2D come motore di rendering si è rivelata soddisfacente per le necessità del gioco e il consumo di CPU, grazie anche al lavoro compiuto da Giombini, si attesta su bassi livelli. In futuro mi piacerebbe riconsiderare la creazione della View sulla base della più moderna libreria JavaFX, al fine di poter valutare gli eventuali vantaggi che ne conseguono.
  È stata un'esperienza istruttiva e impegnativa, che mi ha permesso di comprendere alcune delle principali meccaniche che devono essere considerate nel momento in cui ci si approccia alla realizzazione di un videogame.

- **Giulia Giombini**
  È stata fondamentale la prima parte di analisi del problema in quanto mi ha permesso di affrontare meglio il lavoro, considerando che non conoscevo il gioco originale.
  Sono abbastanza soddisfatta del lavoro compiuto e della collaborazione con i miei compagni. Penso di aver gestito il GameLoop in modo ottimale, il codice è flessibile per eventuali modifiche future: nella classe che si occupa della gestione dei punteggi ho già creato un metodo *reset*(), qualora si volesse aggiungere la possibilità di eliminare tutti i punteggi precedentemente salvati e ricominciare da capo.
  Per eventuali aggiornamenti potrei pensare di muovere i nemici attraverso l'uso di una intelligenza artificiale, intercettando la posizione dell'eroe e facendoli spostare verso di esso.
  È stata la mia prima esperienza all'interno di un gruppo con suddivisione di lavoro. Penso che siamo riusciti nel nostro intento grazie alla collaborazione di tutti.

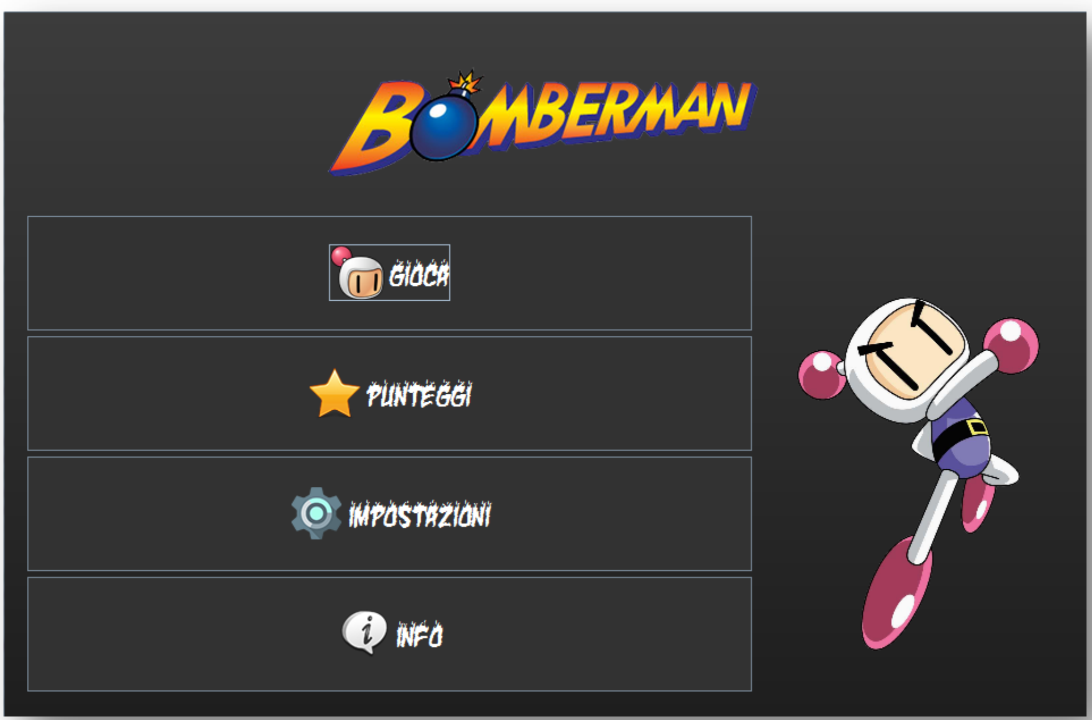
- **Sofia Rossi**
  Per realizzare correttamente il funzionamento logico di questo gioco, mi sono concentrata fin da subito, su una corretta suddivisione degli elementi in modo tale da renderla anche facilmente riadattabile. Ho ragionato quindi su come realizzare una giusta estensione degli aspetti comuni agli elementi statici e dinamici e degli oggetti di cui si compongono.

Ritengo di aver svolto complessivamente un lavoro che, nonostante possa essere migliorato in diversi punti, soddisfi, almeno in parte, le mie aspettative. In particolare la realizzazione delle collisioni con PowerUp potrebbe essere migliorata in modo tale da offrire eventuali messaggi da visualizzare che specifichino il relativo effetto. Concludendo, ritengo di aver posto particolare attenzione all'indipendenza con gli altri componenti ed a una facile estendibilità del model stesso offrendo una suddivisione in package chiara.

Appendice

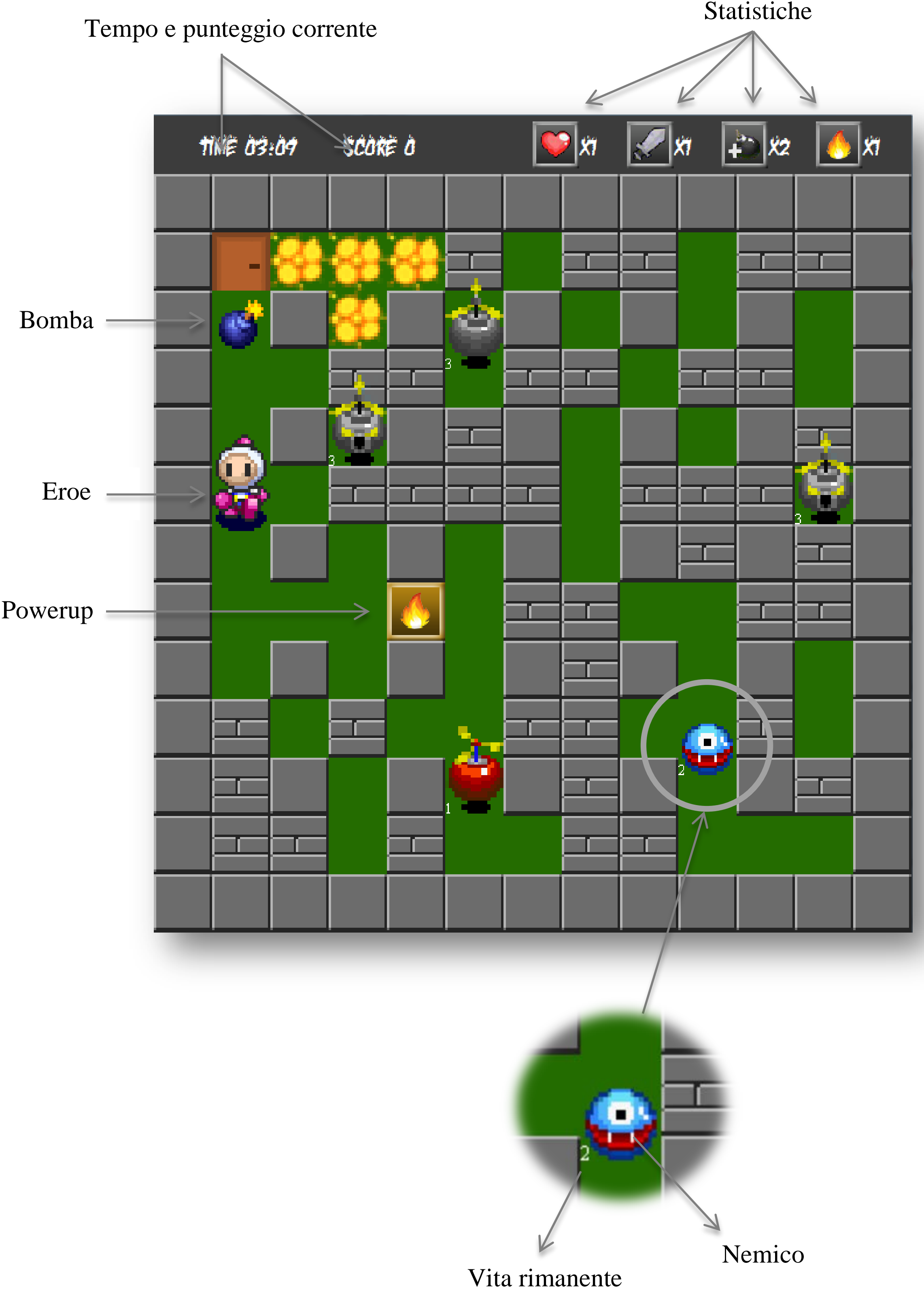
Guida utente

All'avvio dell'applicazione, com'è possibile osservare nella figura sottostante, viene mostrato il menù principale di gioco.



"Gioca" consente di iniziare una nuova partita con le opzioni correntemente selezionate.
"Punteggi" mostra il record detenuto dall'utente e un grafico rappresentante l'andamento delle ultime partite.
"Impostazioni" consente di disabilitare/riabilitare l'audio, impostare la DarkMode e selezionare la lingua di preferenza.
"Info" riepiloga i comandi di gioco, elenca i power-up ottenibili con le relative descrizioni e cita gli autori del software.

Durante una partita è possibile muovere l'eroe mediante la pressione delle frecce direzionali o dei classici pulsanti [W], [A], [S], [D].
Il rilascio di una bomba avviene mediante il tasto [BARRA SPAZIATRICE]. Inoltre, il gioco può essere messo in pausa in qualsiasi momento grazie alla digitazione del tasto [P].
I nemici si spostano all'interno della schermata di gioco secondo strategie differenti: il loro aspetto è legato alla resistenza e all'attacco che li contraddistinguono.
Distruggendo un blocco, occasionalmente, può apparire un power-up.



Lo scopo del gioco è quello di individuare la chiave nascosta all'interno del livello per poter passare a uno stage successivo, sbloccando la relativa porta. Durante questa fase di ricerca, il giocatore deve tentare di raccogliere il maggior numero di punti (uccidendo nemici) nel minor tempo possibile. Utilizzando la dark mode il punteggio finale terrà conto della maggior difficoltà e sarà raddoppiato. Alla fine della partita i risultati ottenuti verranno automaticamente memorizzati.