Relazione progetto “Bomberman”

Paolo Penazzi, Matteo Ragazzini, Andrea Rettaroli,

Marta Spadoni, Lucia Sternini

21 agosto 2019

Indice

Capitolo 1

Analisi

L'applicazione mira alla realizzazione di un remake di Bomberman, il noto gioco sviluppato in Giappone negli anni '80 da Shinichi Nakamoto. Il gioco appartiene al genere *Maze*, in altre parole a quella categoria di videogiochi ambientati all'interno di un labirinto.

1.1 Requisiti

Requisiti funzionali

L'applicazione presenta all'avvio un menù principale con le seguenti opzioni: "Start Game", "Leaderboard", "Settings", "Help".

- La modalità di gioco single player è ambientata in un labirinto composto da blocchi fissi e rompibili. Il personaggio sarà armato di bombe con le quali potrà eliminare i mostri che lo ostacoleranno durante la sua avventura e rompere i blocchi del labirinto; sotto a quest'ultimi sono infatti nascosti "bonus" e "malus" come ad esempio la conquista di una nuova vita, l'aumento della velocità, l'inversione dei comandi per il movimento dell'eroe o il freeze.
  Lo scopo del gioco è trovare la chiave nascosta nel labirinto che permetterà l'apertura della porta e l'accesso al livello successivo.
- Il software sarà in grado di generare il terreno di gioco in maniera casuale e aumentare la difficoltà ad ogni livello.
- Nella sezione "Leaderboard" l'utente potrà visualizzare una classifica basata sul punteggio registrato a fine livello. Il punteggio viene calcolato in base al numero di mostri uccisi e ai "PowerUp" presi. Inoltre, viene mostrato il tempo impiegato a completare il livello stesso.
- Nella finestra "Settings" sarà possibile settare la musica di sottofondo e gli effetti del gioco.
- Nell' "Help" menù una grafica mostrerà in modo intuitivo i tasti da utilizzare per giocare e una breve descrizione dell'obbiettivo del gioco.

Requisiti non funzionali

- BmbMan dovrà garantire una buona giocabilità e portabilità.
- L'applicazione dovrà essere scalabile in base alla risoluzione dello schermo su cui viene avviata.

1.2 Analisi e modello del dominio

L'utente dovrà guidare Bomberman all'interno di un labirinto bidimensionale, popolato da entità statiche e dinamiche. Ogni entità ha una posizione nel terreno e una dimensione fissata. Le figure in movimento, eroe e mostri, sono inoltre dotati di una velocità. Tra le

entità statiche troviamo: le piastrelle, i muri e i blocchi rompibili, delle quali solo le prime sono percorribili.

L'eroe per proteggersi dai mostri avrà a disposizione un numero variabile di bombe. Quest'ultime, una volta piazzate, aspetteranno un determinato lasso di tempo prima di esplodere e quindi eliminare le entità rompibili nel loro raggio di azione. Al di sotto dei blocchi si potrebbero celare dei PowerUp, che avranno un effetto, positivo o negativo, sull'eroe e sulle caratteristiche delle bombe che esso ha disposizione. I potenziamenti possono avere una durata illimitata, come l'aggiunta di una vita, oppure un effetto momentaneo come l'incremento della velocità. La chiave e le porta per uscire dal labirinto possono essere identificati come due tipi particolari di PowerUp.

La difficoltà principale sarà permettere un'ottima coesistenza tra gli elementi costitutivi del dominio, in particolare mireremo ad una buona gestione delle collisioni tra le entità in movimento, i blocchi del terreno e le esplosioni delle bombe.

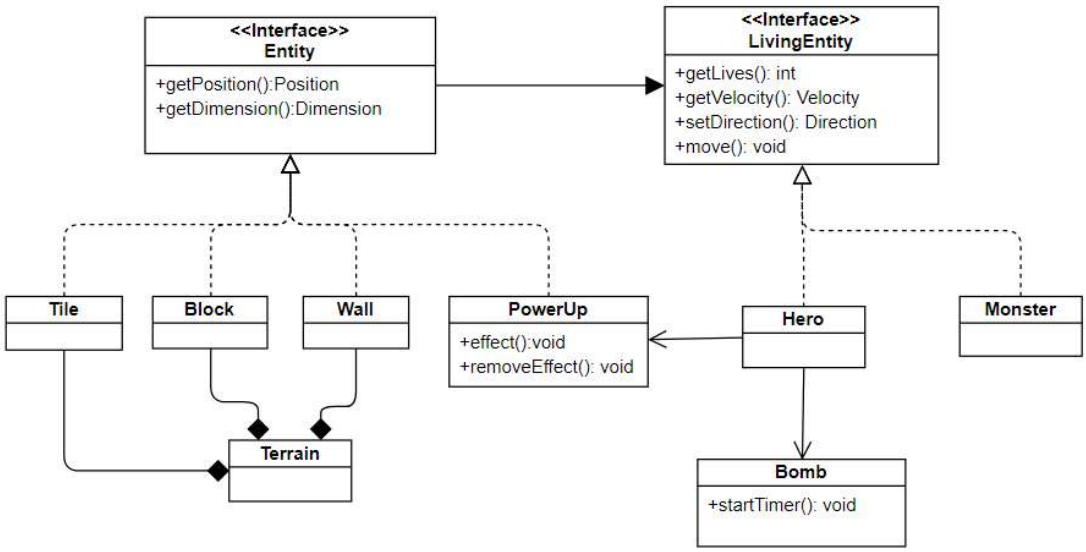

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali e le relazioni fra esse

Capitolo 2

Design

2.1 Architettura

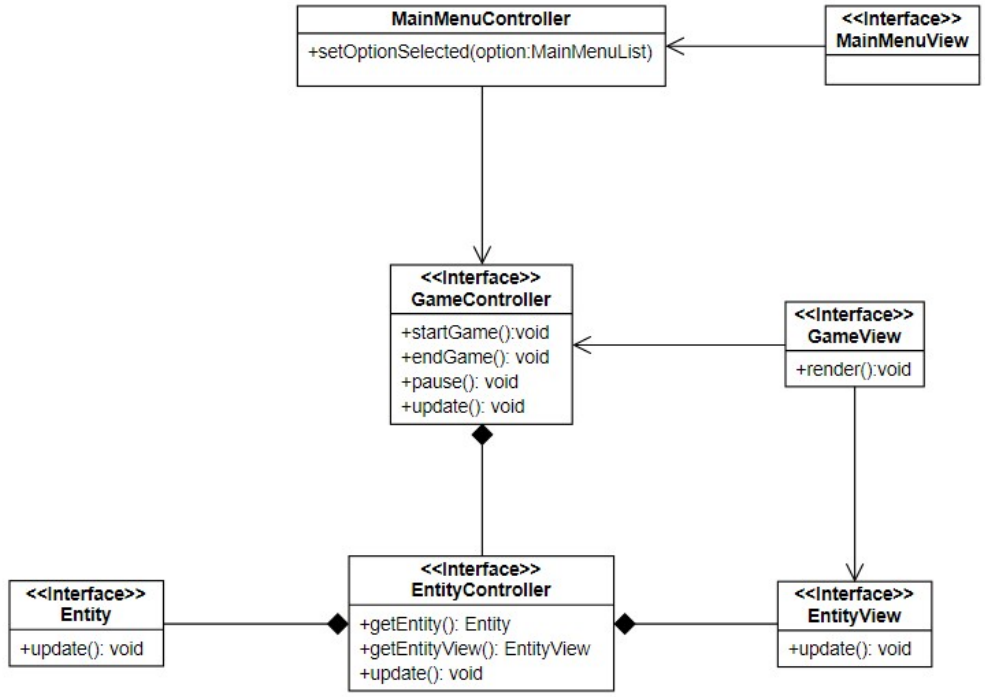

Figura 2.1: Schema UML architetturale di BmbMan.

Nel realizzare in nostro software abbiamo deciso di utilizzare il pattern architetturale MVC, implementato come mostrato in figura. Abbiamo scelto questo pattern, in quanto, come visto a lezione, esso ci permette di suddividere i due aspetti principali dell'applicazione: logica e grafica, e di utilizzare il controller per gestirli.

L'applicazione è avviata dalla MainMenuView, l'opzione selezionata dall'utente viene catturata e gestita dal MainMenuController, il quale istanzia il GameController nel caso in cui l'opzione scelta sia "Start Game".
Il GameController inizializza e coordina gli aspetti generali del model ma rimanda ai singoli EntityController la gestione mirata delle entità a cui essi si riferiscono, in particolare questi propagano alle rispettive EntityView le modifiche avvenute nel model di ciascuna entità (Entity). L'aggiornamento generale della view è invece delegato ad una implementazione della GameView, la quale si occupa del rendering di tutte le EntityView in gioco.

Questa struttura permette una facile modifica di ogni aspetto, il model è infatti totalmente scorporato dal controller e la gestione della grafica è tale da permettere una sostituzione della view in blocco.

2.2 Design dettagliato

Paolo Penazzi

La mia parte di progetto riguardava la creazione del menù principale, del menù help e dei power-up, tra cui in particolare la chiave e la porta per passare al livello successivo.

Per lo sviluppo del menù principale, per quanto riguarda la parte di controller ho sfruttato l'implementazione realizzata da Matteo Ragazzini, rimando quindi al suo paragrafo per maggiori dettagli. Per realizzare la parte di view, al fine di garantire l'uniformità e la coerenza dello stile dell'applicazione, ho deciso insieme a Matteo Ragazzini, il quale aveva come compito la realizzazione dell'options menu, di fare ricorso al pattern Factory, per la creazione e il settaggio di tutti i JComponent utili nelle nostre view. Abbiamo cosi creato l'interfaccia GuiFactory e la classe GuiFactoryImpl. Per quanto riguarda il MainMenuView ho associato ad ogni pulsante la corrispondente opzione della MainMenuList tramite un ActionListener.

Per la classe HelpView, notando che non necessitava di alcuna interazione con l'utente, ho deciso di non realizzare un'interfaccia grafica ma di caricare un'immagine esplicativa da me creata con i comandi di gioco.

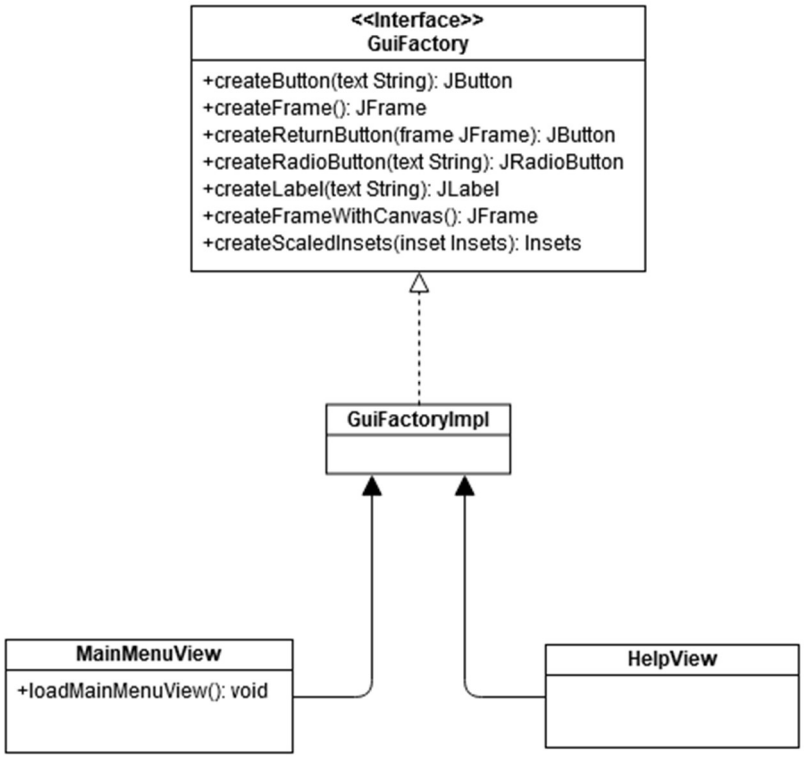

Figura 2.2: Schema UML dell'implementazione del pattern Factory per la creazione delle componenti di View.

Per quanto riguarda i Power-up, essendo questi un'entità del gioco, dovevano necessariamente estendere AbstractEntity; ma da un'ulteriore analisi, ho scelto di utilizzare il pattern Template Method per l'attivazione e la rimozione dell'effetto di ognuno. Di conseguenza, ho creato un'ulteriore classe astratta AbstractPowerupEntity. Qui implemento i metodi *onCollision()* e

update() di AbstractEntity rendendoli i metodi template del pattern, all'interno di essi invoco rispettivamente i metodi astratti *powerupEffect()* e *removeEffect()*.

Anche se non sono propriamente dei bonus o dei malus, ho gestito la chiave e la porta come tali, in quanto il loro comportamento è molto simile ai power-up "classici".

Per la parte di view ho notato che il modo di visualizzare il power-up è uguale per tutti, l'unica cosa che cambia è l'immagine. Ho esteso quindi AbstractEntityView, creando la classe PowerupView che prende in input il percorso dell'immagine da caricare. Successivamente ho creato un Enum PowerUpType, contenente ogni tipo di powerUp e ho associato ad ognuno il percorso dell'immagine specifica.

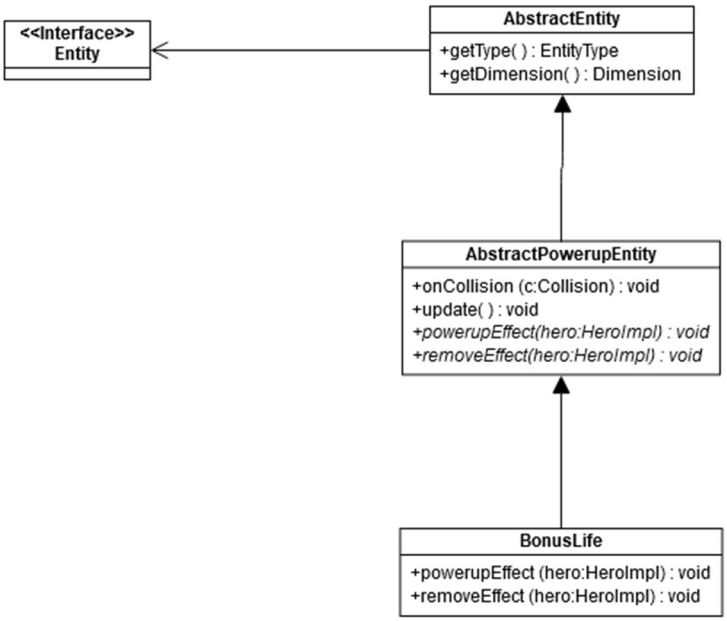

Figura 2.3: Schema UML dell'implementazione del pattern Template Method.

Successivamente ho realizzato alcune classi di utility.

La classe ScreenToolUtils, realizzata in collaborazione con Matteo Ragazzini, scala la grafica in base alla risoluzione dello schermo. Per maggiori dettagli, rimando al suo paragrafo.

Inoltre, ho realizzato la classe GameFont, utilizzata per caricare il font personalizzato.

Infine, per scalare un'immagine in base alla dimensione del panel, ho trovato e aggiunto al progetto la classe BackgroundPanel (https://tips4java.wordpress.com/2008/10/12/background-panel/). Di questa ho modificato solamente la javadoc, alcuni modificatori dei campi e l'indentazione del codice. L'implementazione dei metodi è stata lasciata come trovata. Questa classe ha dato problemi in quanto le immagini in alcuni casi risultavano sfuocate. A questo punto ho cambiato approccio, creando la classe ImageLoaderUtils e caricando un'immagine diversa a seconda della risoluzione dello schermo.

La classe BackgroundPanel non è stata eliminata in quanto è stata utilizzata da Matteo Ragazzini per impostare un'immagine di sfondo nelle sue classi di view.

Matteo Ragazzini

Mi sono occupato principalmente della realizzazione dei mostri, del caricamento di musica ed effetti e della gestione di due parti di View: menù opzioni e menù di GameOver/Win.

Durante la fase di analisi e modello del dominio svolta unitamente dal gruppo, si è determinato che tutte le entità in gioco avessero caratteristiche comuni, quali ad esempio la dimensione e la posizione sul terreno di gioco. Si è quindi deciso, per massimizzare il riutilizzo del codice, di creare una classe astratta AbstractEntity che modellasse questo concetto mantenendo astratti i soli metodi che caratterizzano le singole entità. Avendo notato che i mostri e l'eroe in quanto entità vive presentavano molte somiglianze ho proseguito la fase di analisi con Marta Spadoni. Insieme abbiamo concepito la necessità di un'ulteriore classe astratta, AbstractLivingEntity, che esprimesse il concetto di entità con una (o più vite) e in grado di muoversi, dalla quale estendere successivamente le nostre classi.
Per quanto riguarda la gestione del movimento ci siamo resi conto che per le due tipologie di entità ciò che differiva non era la modalità di movimento, ma il modo in cui l'entità operasse il cambio di direzione, per l'eroe ciò avviene tramite un input da tastiera, mentre per il mostro tramite una strategia implementata via software. Abbiamo cosi proceduto alla realizzazione del movimento convenendo che per quest'ultimo servissero due concetti: la direzione e la velocità. Il primo è stato modellato tramite una Enum Direction mentre il secondo è stato modellato con una classe ad hoc, Velocity, che rappresenta una sorta di vettore, nel senso fisico del termine.

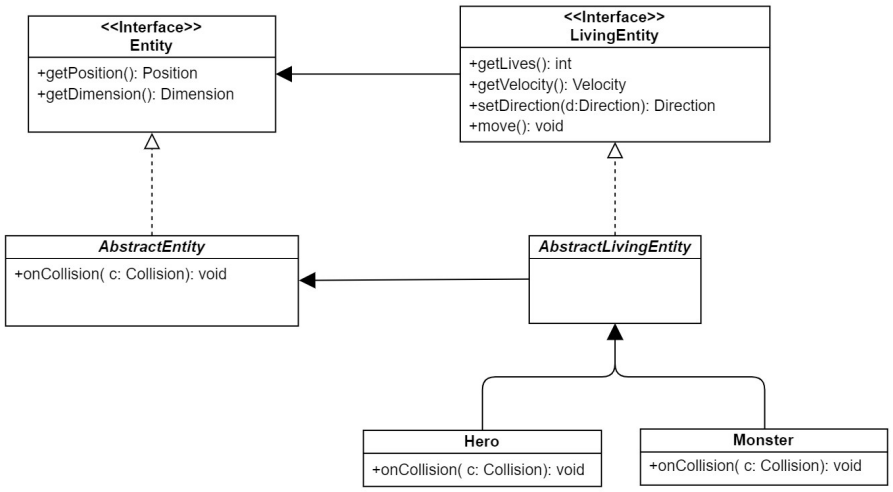

Figura 2.4: Schema UML del model generale delle entità.

Questa efficiente scomposizione del problema mi ha permesso di creare una classe Monster estremamente snella, in quanto rimaneva da implementare solamente la modalità con cui quest'ultimo cambiasse la sua direzione e il metodo *onCollision()*. Per favorire l'utente e offrirgli un gioco godibile ho voluto intenzionalmente modellare il movimento del mostro in maniera semplice, quest'ultimo infatti cambia in maniera random la sua direzione in caso di collisione con una qualsiasi entità, mentre inverte la sua rotta in caso di collisione con l'eroe. Per quanto riguarda la componente grafica del mostro avendo seguito una modellazione molto simile a quella già esposta per il model essa è coerente con quella di tutte le altre EntityView e viene trattata in dettaglio nel paragrafo di Marta Spadoni.

Per quanto riguarda il menù opzioni ho cercato una soluzione che mi permettesse un facile ed immediato ampliamento del menù senza mettere mano alla parte di interfaccia grafica ma agendo solo sul controller. Allo scopo ho realizzato un'interfaccia generica riusabile anche in altri menù, un Enum OptionsMenuList contenente tutte le varie opzioni settabili tramite il menù stesso, e un

OptionsMenuController che in base ad ogni elemento dell'Enum effettua un operazione specifica. Nella classe OptionsView vengono creati in maniera automatica tanti bottoni quante sono le voci dell'optionsMenuController e ho adattato i metodi realizzati nel controller come ActionListener per i pulsanti grazie al pattern Adapter. Per la realizzazione del pattern ho inserito un inner class nella OptionsView che implementa l'interfaccia ActionListener.

Inoltre, per mantenere coerente lo stile della grafica del gioco ho fatto uso della Factory creata insieme a Paolo Penazzi e da lui descritta.

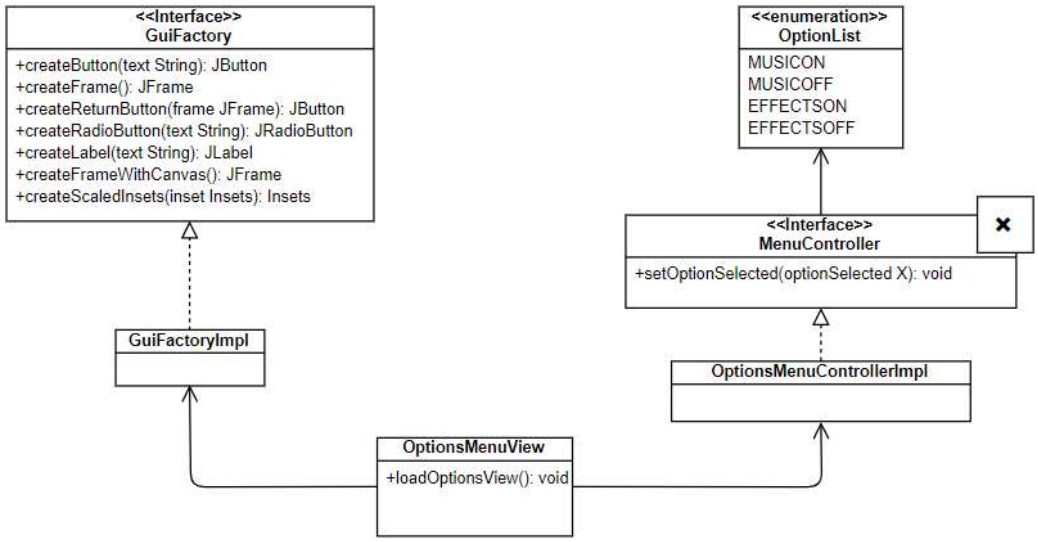

Figura 2.5: Schema UML rappresentante la gestione della view dell'optionMenu e del relativo controller.

Per la realizzazione del menù di fine partita, rendendomi conto che sia in caso di vittoria che in caso di sconfitta l'utente avrebbe voluto vedere il suo punteggio ed eventualmente salvarlo nella leaderbord, ho realizzato una sola classe EndView alla quale viene passato il GameController che stabilisce se l'utente ha vinto ho meno e solo in caso positivo viene creato un pulsante che permette di accedere al livello successivo.

Dovendo gestire nel menù opzioni aspetti relativi alla musica e agli effetti sonori mi sono fatto carico della gestione di questi ultimi. Ho realizzato a proposito l'interfaccia Sound, e la sua implementazione SoundImpl nella quale utilizzo la classe javax.sound.sampled.Clip la quale mi permette di effettuare operazioni come la riproduzione in loop di un suono senza effettuare controlli diretti sullo stato di quest'ultimo nella GameEngine. Successivamente ho realizzato la classe SoundsController per la gestione dei vari Sounds, creati come opzionali in quanto il loro caricamento dipende dalla selezione dell'utente nel menù opzioni. Inoltre, i vari metodi getter per la restituzione dei suoni sono stati impostati statici per permettere il loro utilizzo nelle altre classi senza la necessità di istanziare molteplici volte SoundsController.

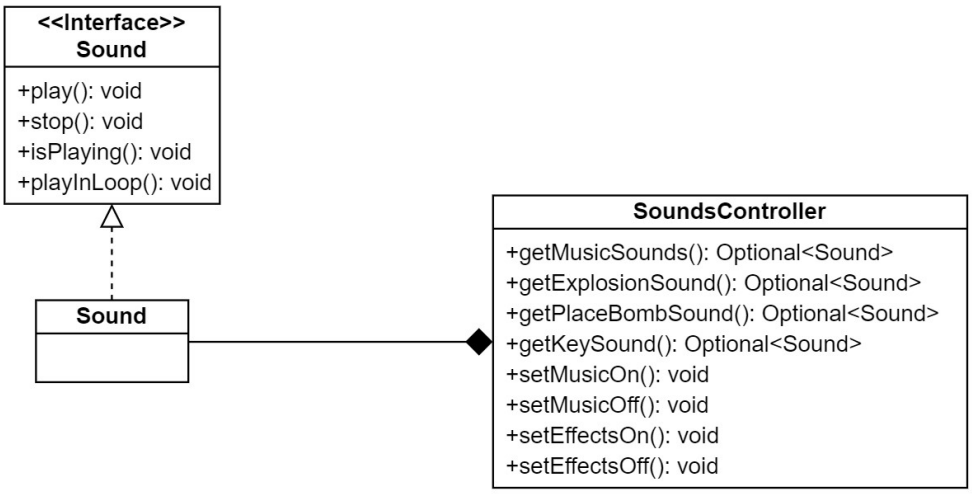

Figura 2.6: Schema UML generale della gestione dei suoni.

Infine, per realizzare quello che era uno dei nostri requisiti non funzionali, ossia la creazione di un'applicazione scalabile in base alla risoluzione dello schermo, ho realizzato insieme a Paolo Penazzi una classe di utility ScreenToolUtils che identifica il tipo di schermo sul quale l'applicazione viene lanciata e in base a quest'ultimo fornisce esternamente un fattore di scala permettendo alle entità in gioco e agli elementi di view di adattarsi al meglio allo schermo.

Andrea Rettaroli

Principalmente mi sono dedicato alla creazione del campo da gioco e dei suoi componenti: muri, blocchi e piastrelle. Inoltre, mi sono occupato del game loop e della creazione del mondo. Inizialmente la priorità del gruppo era avere un game engine che permettesse di avviare e aggiornare l'applicazione. Dopo una attenta analisi, ho estrapolato quelli che sono i comportamenti principali del game loop, riassumibili in tre fasi:

-fase di avvio nella quali si sarebbero dovuti creare tutti i componenti del game.

-fase di interruzione che può derivare da una pausa o dalla chiusura del gioco.

-fase di gioco nella quale vengono aggiornate tutte le varie componenti.

Una volta individuati questi aspetti cardine ho creato l'interfaccia GameEngine che si occupa appunto di modellare queste tre fasi. Successivamente mi sono dedicato a sviluppare una classe capace di implementare nello specifico quanto detto. Queste fasi sono facilmente gestibili tramite l'uso di un Thread ed è per questo che GameEngineImpl estende la classe Thread e implementa l'interfaccia GameEngine. L'obbiettivo cardine di questa classe è scandire i tempi di aggiornamento a ogni componente del gioco ad ogni frame, sia nel model che nella view.

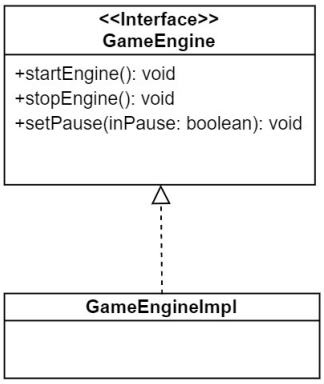

Figura 2.7: Schema UML dell'implementazione della GameEngine.

Successivamente mi sono dedicato alla creazione del terreno. È stato molto facile per me creare un terreno che permettesse ai miei compagni di testare le loro componenti in quanto la creazione dell'interfaccia Entity, sviluppata in gruppo, modella tutti gli aspetti comuni a tutte le entità presenti nel gioco. Per la creazione di Tile, Wall e Block mi è bastato estendere AbstractEntity che implementa Entity, lo stesso procedimento è stato utilizzato per i relativi componenti anche nella parte di view tramite AbstractEntityView e EntityView. Idealmente, grazie all'attenta analisi iniziale, ho sempre visto il terreno come una scacchiera composta da celle nelle quali sarei andato ad inserire piastrelle, muri e blocchi. Per garantire maggiore giocabilità l'eroe, all'inizio di ogni livello è al sicuro da mostri grazie a dei controlli fatti nella fase di creazione del terreno. Inizialmente ho creato un classe provvisoria che legasse queste componenti e permettesse a tutti di testare i loro progressi. In un secondo momento ho applicato il pattern Simple Factory. TerrainFactory genera un Terrain attraverso il metodo create. I due principali pattern che potevo applicare a queste classi erano Builder o Factory, il primo è stato escluso in quanto è più utile nella creazione di oggetti con campi fissi e campi opzionali. Nel nostro caso si ritiene necessario l'inserimento di tutti gli oggetti. Inoltre, dopo un analisi in chiave futura ho preferito usare factory in quanto aggiungendo un nuovo metodo di creazione del campo da gioco sarebbe stato facile creare un nuovo terreno per la modalità multiplayer passando come parametro al metodo un campo che indica la modalità.

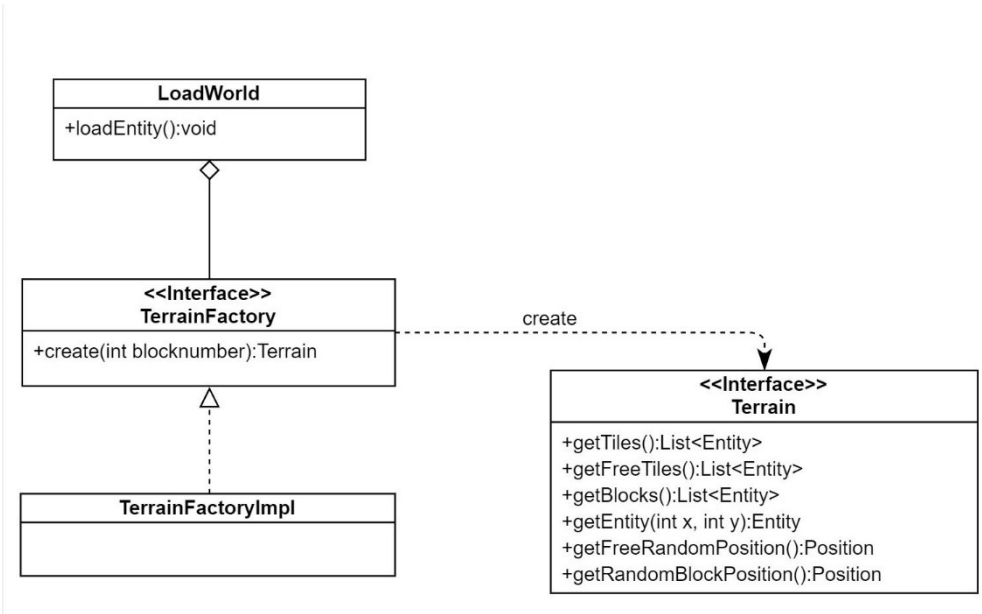

Figura 2.8: Schema UML realizzazione del pattern Simple Factory.

La classe LoadWorld si occupa di caricare ogni componente presente nel gioco, sia nel model che nella view. Ciò avviene in base al livello in cui ci troviamo. Ho creato anche un interfaccia Level che modella i livelli del gioco. La sua implementazione si occupa di decidere quanti e quali power up saranno presenti all'interno di quel livello, il numero dei mostri e il numero dei blocchi, nonché gli aspetti essenziali per far sì che la difficoltà aumenti di livello in livello.

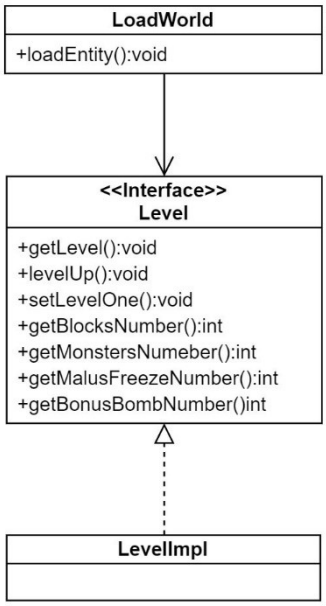

Figura 2.9: Schema UML dell'interazione di level e LoadWorld.

Marta Spadoni

Mi sono occupata di sviluppare i principali controller dell'applicazione, di gestire l'eroe e le collisioni e infine di realizzare le classi necessarie a creare le view e le animazioni.

Il GameController si occupa di gestire il gioco, dall'avvio fino al completamento del livello. In particolare, invoca il caricamento del terreno e coordina i vari avvenimenti. Per evitare di rendere questa classe una "God class" ho deciso di aggiungere il concetto di EntityController, in questo modo le singole entità non dovevano essere gestite dal controller principale.
Ciascun EntityController ha un riferimento ad una particolare entità nel model (Entity) e ad una view (EntityView) e si occupa di propagare gli effetti degli eventi avvenuti nel model, alla relativa componente grafica e di invocare l'aggiornamento di entrambe, contestualmente alla GameEngine.

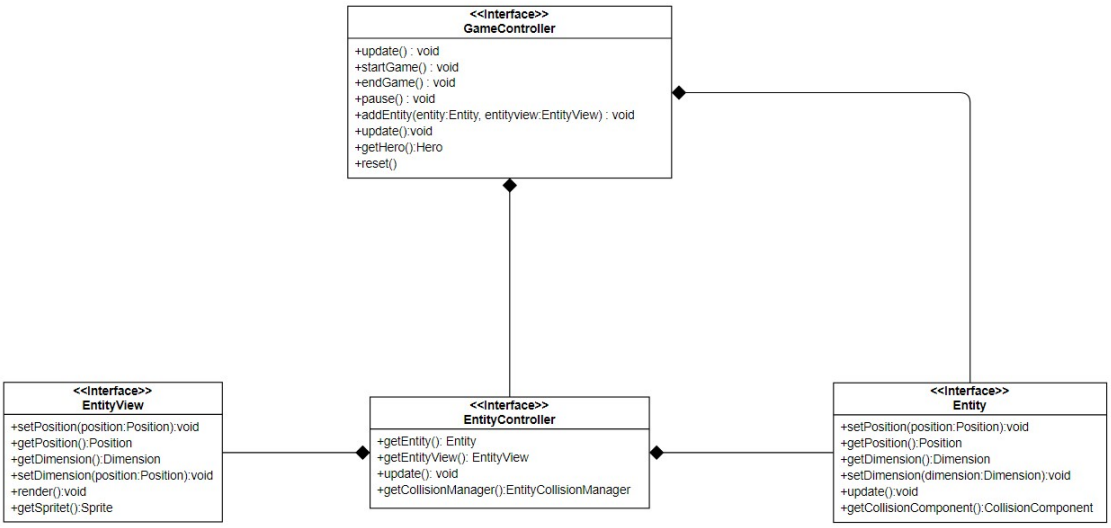

Figura 2.10: Schema UML dei principali controller dell'applicazione.

Per quanto riguarda la gestione delle collisioni, al fine di rendere questa parte il più autonoma possibile e quindi anche facilmente sostituibile con una versione più avanzata, ho deciso di introdurre nel concetto di entità una componente che rappresentasse la "struttura fisica" dell'oggetto, il CollisionComponent, e di associare a ciascuna figura del model, che ne avesse bisogno, un CollisionManager. Quest'ultimo, sotto richiesta del GameController, è in grado di determinare se l'entità seguita ha colliso con un'altra e in caso positivo di notificare le due entità coinvolte. Il CollisionManager, per determinare una collisione, interseca il CollisionComponent dell'entità seguita con quello di un'altra passata dal GameController. Il modo in cui determino una collisione è dunque abbastanza banale ma data la struttura ritengo che in futuro possa essere facilmente migliorata con una versione più avanzata. Nella nostra modellazione è l'entità che viene colpita a sapere come reagire alla collisione, per questo ognuna ha un metodo che prende in ingresso una Collisione cioè un oggetto che definisce con chi e dove si ha colliso, e in esso determina come comportarsi. Con un'attenta analisi ho potuto determinare che solo per le entità in movimento fossero necessari i CollisionManager dunque negli EntityController mantengo questa componente opzionale.

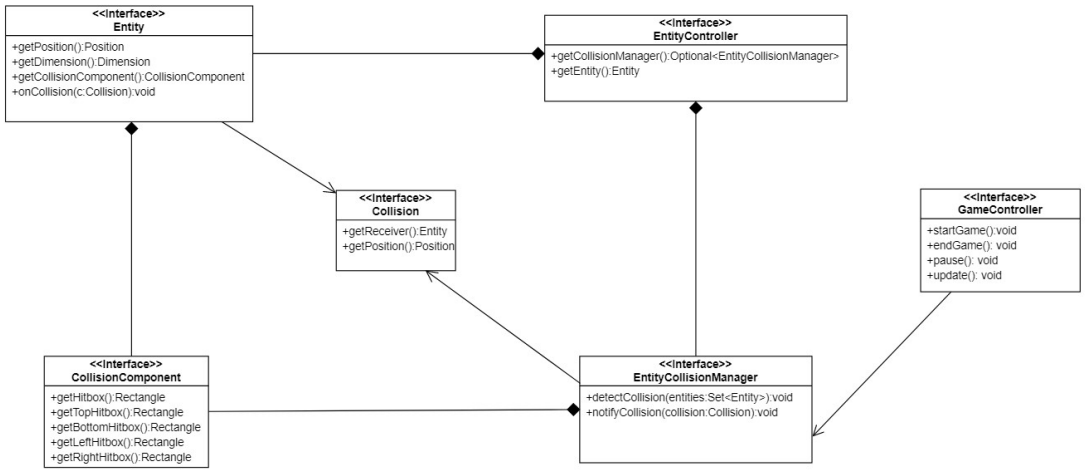

Figura 2.11: Schema UML della gestione delle collisioni con focus su CollisionComponent ed EntityCollisionManager.

Per la gestione dell'eroe, avendo già strutturato i principali componenti di ogni entità insieme a Matteo Ragazzini mi rimaneva da definire in che modo l'utente potesse direzionarlo. Grazie all' interfaccia KeyListener di awt ho potuto creare un KeyInput, questo trasforma la pressione di un

tasto della tastiera in una chiama ad un metodo del GameController per settare la direzione dell'eroe. Infatti, per come abbiamo modellato il movimento delle entità, ogni qualvolta si setta la direzione le componenti della velocità vengono modificate di conseguenza e la posizione viene aggiornata ad ogni update coerentemente.

La parte sugli aspetti grafici ha riguardato in principio la stesura dell'interfaccia EntityView definita in gruppo, in seguito però ho determinato la necessità di una classe astratta AbstractEntityView, in quanto tutte le view del nostro gioco si distinguono solo per l'immagine associata che deve essere renderizzata. In particolare proprio per questo ultimo aspetto ho deciso di utilizzare il pattern comportamentale Template Method, dove il metodo usato come template è, per l'appunto il metodo *render()*, questo al suo interno utilizza il metodo astratto *getSprite*(), settato solo in seguito dalle sottoclassi in base alle loro necessità.

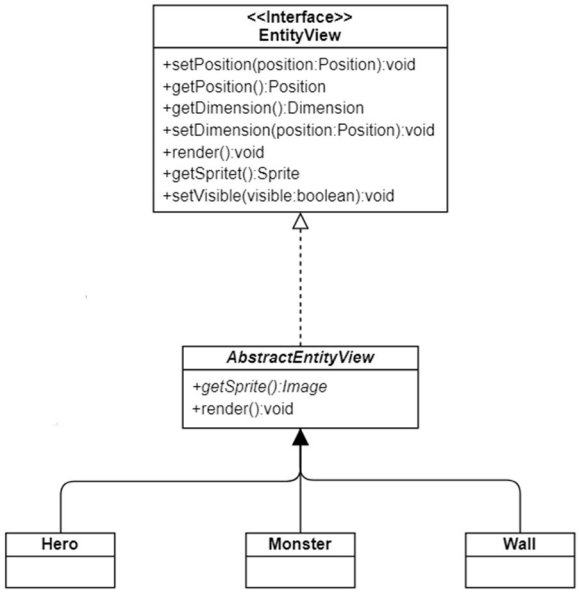

Figura 2.12: Schema UML del pattern Template applicato all'interno della classe AbstractEntityView.

L'utilizzo del pattern sopracitato mi ha inoltre facilitato l'inserimento del concetto di animazione, strutturato come segue: un'Animation è un insieme di Sprite che vengono restituite sequenzialmente ogni volta che l'entità deve essere renderizzata. Per facilitare la creazione di un'animazione ho utilizzato il pattern creazionale static factory, ho reso infatti privato il costruttore della classe AnimationImpl e creato un metodo statico *createAnimation()*, in questo modo "obbligo" la creazione di un'animazione partendo da un'immagine contenente su una riga tutti i frame necessari. Operando in questo modo lascio l'opportunità di creare in futuro altri metodi con cui generare un'animazione partendo da immagini strutturate diversamente. Per quanto riguarda invece la necessità di avere restituiti sequenzialmente i vari frame dell'animazione, ho deciso di utilizzare il pattern strutturale Iterator, attraverso la definizione dell'interfaccia AnimationIterator e inserendo nel contratto di Animation il metodo *createAnimationIterator()*. Nel nostro gioco, vi è la necessità di avere animazioni continue cioè una volta restituiti tutti i frame si deve ricominciare dal primo e così via, per questo ho deciso di realizzare un InfiniteAnimationIterator, il quale restituisce sequenzialmente e all'infinito le varie Sprite.
Le classi Sprite e SpriteSheet di utilizzo generico per la gestione delle immagini sono state realizzate in collaborazione con Matteo Ragazzini.

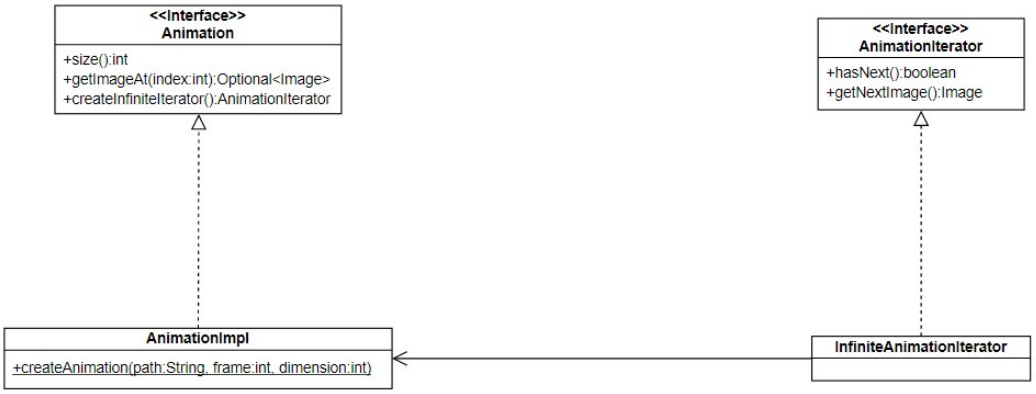

Figura 2.13: Schema UML raffigurante il pattern Static Factory implementato in AnimationImpl e il pattern Iterator.

Lucia Sternini

Mi sono dedicata alla gestione delle bombe e della loro esplosione; inoltre, mi sono occupata del calcolo del punteggio per livello e dell'elaborazione della classifica.

Per quanto riguarda la gestione delle bombe, ho realizzato due interfacce BombController e Bomb: la prima si occupa di gestire tutte le bombe piazzate durante il gioco mentre la seconda modella il concetto di bomba e della conseguente esplosione.
Ho deciso di realizzare il BombController per coordinare View e Model della bomba e gestire le collisioni di essa con le varie entità. L'azione di piazzamento della bomba viene catturata dal KeyInput che rimanda al GameController il compito di comunicare al BombController di creare la bomba e avviare il timer per stabilire il tempo di permanenza sul terreno.
La classe BombImpl eredita il comportamento di AbstractEntity, implementando i metodi lasciati astratti e aggiungendone altri propri del suo comportamento, definiti nell'interfaccia Bomb. Per quanto riguarda le sue proprietà, oltre a quella del timer, è dotata di uno stato e di un'esplosione, quest'ultima è tuttavia opzionale in quanto verrà creata solo nel momento in cui il timer termina. In riferimento allo stato, ho deciso di realizzare l'Enum BombState che modella le seguenti condizioni: piantata, in esplosione ed esplosa.

Ho scelto di realizzare l'oggetto Explosion mediante la specializzazione della classe generica Pair, sostituendo le type-variable con due Rectangle. Questi ultimi hanno dimensione variabile in base al range attuale della bomba e danno all'esplosione la forma di una croce. Tale implementazione permette una facile identificazione delle collisioni, data la struttura utilizzata anche per le altre entità del gioco, descritta nel paragrafo di Marta Spadoni.

Infine, la classe BombView si occupa della grafica di bomba ed esplosione, avvalendosi dello stato aggiornato nel model per caricare le sue animazioni.

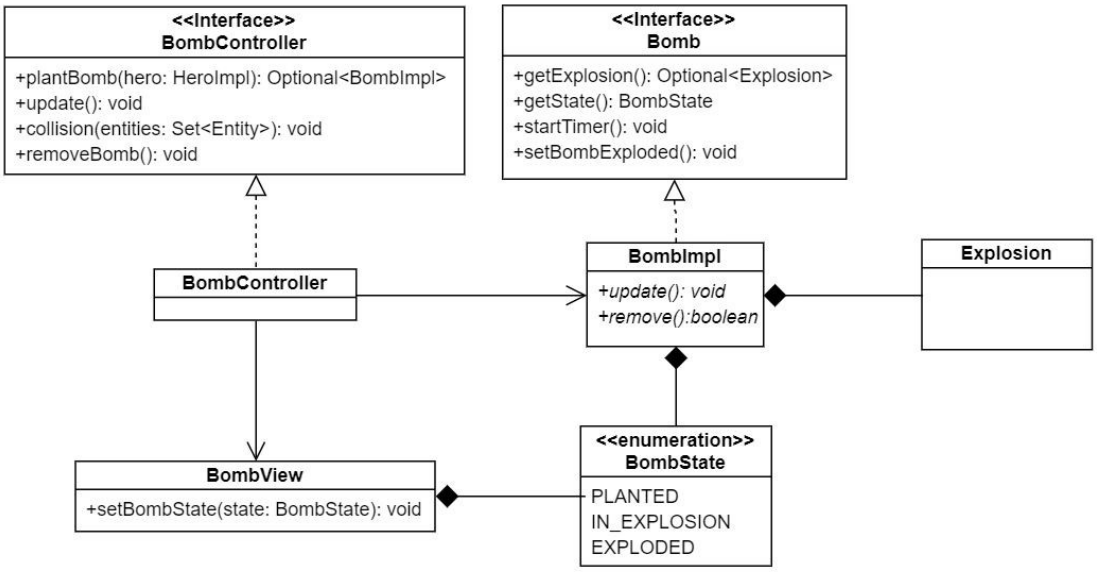

Figura 2.14: Schema UML raffigurante la gestione di bombe ed esplosioni tramite BombController.

In relazione alla gestione dei punteggi, la classe immutabile ScoreHandler legge gli oggetti serializzabili PlayerScore all'interno di un file e li aggiunge ad una lista ordinata. La lettura del file avviene all'avvio dell'applicazione, tramite il MainMenuController e l'ordinamento degli oggetti è effettuato in base al punteggio ottenuto per livello.

La scrittura su file invece avviene al termine della partita. In particolare, se il giocatore è già presente nel file e il punteggio attuale è superiore a quello esistente, questo viene aggiornato. Al contrario, non viene effettuato nulla poiché ho scelto di elaborare una classifica in relazione al punteggio massimo di ogni giocatore per ciascun livello.

Ho sfruttato facilmente il pattern Decorator alla base del modello di gerarchia delle classi nel package java.io, nella lettura e scrittura su file.
Il ruolo di Component è interpretato da InputStream e OutputStream mentre la funzione di ConcreteComponent è ricoperta da ObjectInputStream e ObjectOutputStream (estendono rispettivamente le classi astratte precedenti). I decoratori invece sono FileInputStream e FileOutputStream.

Per quanto riguarda l'aggiornamento del punteggio, questo viene effettuato all'interno di PlayerScoreImpl, sfruttando la classe Enum Scoring. Essa definisce un certo punteggio per ciascuna entità rilevante per il calcolo della classifica.

Per visualizzare in forma tabellare i punteggi, con i relativi giocatori e il tempo da loro impiegato per completare un determinato livello, ho scelto di utilizzare la classe JTable di Swing all'interno della LeaderboardView. Essa viene popolata tramite la classe ScoreTable che per ereditarietà implementa i metodi astratti di AbstractTableModel.

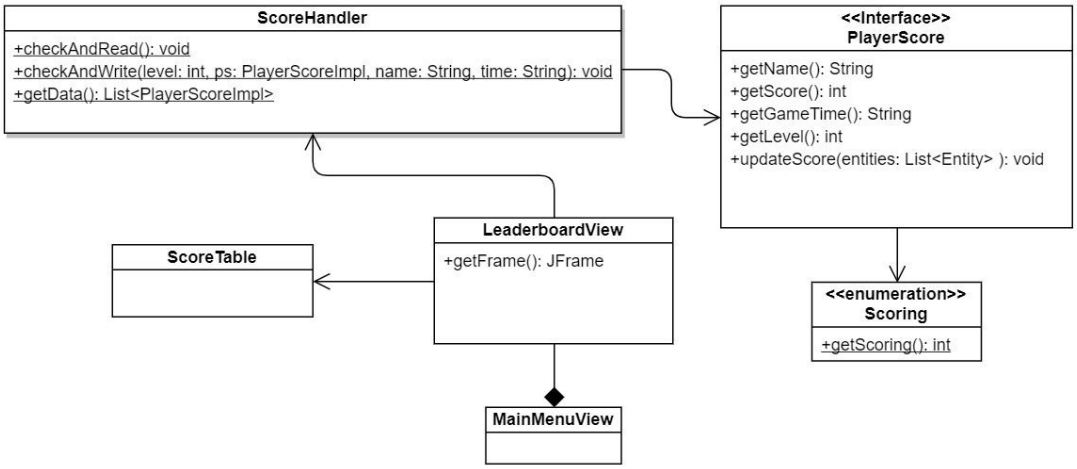

Figura 2.15: Schema UML per la gestione di punteggio e classifica.

Infine, ho realizzato la TopBar in SinglePlayerView per visualizzare durante il gioco le vite rimaste all'eroe, l'aggiornamento di volta in volta del punteggio, la chiave se in possesso all'eroe e lo scorrere del tempo di gioco. In particolare, quest'ultimo è stato creato con il componente Timer di Swing.

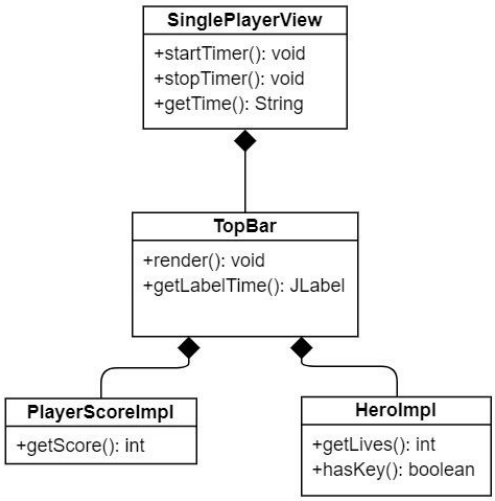

Figura 2.16: Schema UML della TopBar.

Capitolo 3

Sviluppo

3.1 Testing Automatizzato

Abbiamo realizzato molteplici test automatici basati sulla suite di JUnit.
Tutti i test si trovano nei package it.unibo.bmbman.tests e sono stati nominati in base alla classe che testano, ognuno di noi ha testato le proprie classi del model del gioco.
Sono di particolare rilevanza quelli sulle collisioni, realizzati nella classe TestCollision, quelli sulle bombe e la loro esplosione, realizzati in TestBomb e quelli sulla consistenza della generazione del terreno, realizzati in TestTerrain, in quanto queste tre classi testano le componenti fondamentali del gioco.

3.2 Metodologia di lavoro

La fase di analisi e la successiva di design architetturale sono state svolte in gruppo. Ci siamo infatti riuniti periodicamente per definire in modo preciso e completo i requisiti del software, il modello del dominio e per decidere insieme l'architettura dell'applicazione. Insieme abbiamo per l'appunto definito le interfacce principali di model, controller e view così da poter poi procedere quanto più possibile singolarmente e in parallelo con le proprie parti, di seguito esposte:

- Paolo Penazzi: gestione dei PowerUp e del menù principale di avvio.
- Matteo Ragazzini: gestione dei Mostri, del menù opzioni, del menù di fine partita e dell'inserimento della musica di sottofondo e degli effetti sonori.
- Andrea Rettaroli: creazione del labirinto, gestione delle sue componenti (Tile, Wall e Block) e realizzazione della GameEngine.
- Marta Spadoni: gestione dell'eroe, della fisica delle collisioni, implementazione dei principali controller e degli aspetti generali delle view delle entità.
- Lucia Sternini: gestione della Leaderboard, del timer di gioco e delle Bombe.

Ciascun membro del gruppo ha sviluppato, quando possibile, le proprie parti in tutti e tre gli aspetti di MVC. Tuttavia, al fine di dare una solida struttura al codice, classi relative ad aspetti comuni, sono state individuate e definite cooperando, mentre la scelta di come gestire le collisioni è stata presa ad alto livello da tutto il gruppo. Al fine di permettere un'ottima integrazione tra tutte le parti sviluppate in autonomia si è ritenuto agevole permettere la modifica da parte di tutti di alcune classi come ad esempio la classe Hero per l'uso dei PowerUp.

Il lavoro di gruppo è stato facilitato dall'uso del DVCS GIT, abbiamo scelto di utilizzare un branch develop in cui fare i commit durante tutta la fase di sviluppo del codice e di utilizzare, occasionalmente, branch specifici per realizzare in autonomia delle feature in modo tale da non impattare il lavoro degli altri. Infine, il branch master contiene solo la versione definitiva del software.

3.3 Note di sviluppo

Ognuno di hai fatto uso delle seguenti feature avanzate:

Paolo Penazzi

- InputFile per caricare le immagini e il font
- Lambda Expression nell'associare ad un JButton un ActionListener

Matteo Ragazzini

- Generici nell'interfaccia MenuController, per permettere l'implementazione di quest'ultima per ogni menù
- Optional per la classe SoundsController per evitare l'uso dei null nel momento in cui si scelga di non caricare il suono.
- Lambda per ottenere un codice più compatto e leggibile nelle GUI

Per comprendere come effettuare il caricamento dei suoni, argomento non trattato a lezione ho fatto uso di snippet di codice dal progetto Bomberman2018 migliorandone la gestione tramite l'apposito SoundsController da me realizzato.

Andrea Rettaroli

- Stream e lambda usate in particolar modo nella classe TerrainFactoryImpl per semplificare le operazioni sugli oggetti presenti nel terreno

Marta Spadoni

- Uso delle Lambda e degli stream, in particolare in GameController, al fine di rendere il codice più compatto e leggibile
- Uso degli Optional per evitare il null

Lucia Sternini

- Lambda per ottenere un codice più compatto
- Stream per lavorare sulle collezioni
- Optional nelle classi ScoreHandler e BombImpl
- Serializzazione, scrittura e lettura da file

Capitolo 4

Commenti finali

Data l'inesperienza del gruppo nell'affrontare progetti di tali dimensioni, abbiamo ritenuto necessario documentarci il più possibile tramite diverse fonti: slide fornite dai professori, video su Youtube, articoli e vecchi progetti presentateci come buon esempio da seguire. Riteniamo inoltre che, solamente con la preparazione fornitaci a lezione, sia stato difficile coordinarsi nel lavoro di gruppo e suddividere il progetto in parti nette così da permettere un lavoro autonomo e parallelo da parte di tutti gli esponenti del gruppo.

4.1 Autovalutazione e lavori futuri

Paolo Penazzi

Lavorare a questo progetto mi ha permesso di capire quali sono gli aspetti positivi e negativi dello sviluppo in team. Da un lato sono molto soddisfatto del risultato raggiunto con questo progetto, dall'altro sono consapevole che ci sono aspetti che possono essere migliorati, ma nel complesso mi sento di dire che abbia svolto un buon lavoro. Devo ammettere che il non essere arrivati a sviluppare la modalità di gioco multiplayer mi è dispiaciuto, così come non vado molto fiero del fatto che il gioco non scali bene su schermi con risoluzione diversa dal FullHD o dal 4K. A parte questo credo che il gioco presenti una buona giocabilità e, seppur semplice, una bella grafica. Per quanto riguarda lo sviluppo, ho affrontato diverse difficoltà, ma sono contento perché sono riuscito a superarle tutte, documentandomi e cercando di capire a fondo ciò che stavo facendo. Mi sono reso conto che il mio carico di lavoro era leggermente inferiore rispetto a quello dei miei compagni, per questo sono sempre stato disponibile ad aiutarli. Concludo dicendo che a livello di gruppo mi è piaciuto molto come abbiamo lavorato e credo ognuno abbia dato il meglio di sé per la riuscita di questo progetto.

Matteo Ragazzini

Sono complessivamente soddisfatto del risultato ottenuto, anche se sono consapevole del fatto che alcuni aspetti del gioco avrebbero potuto essere scomposti e migliorati ulteriormente. Ad esempio, prevedendo un inserimento futuro di varie metodologie di movimento del mostro, avrei potuto utilizzare un pattern strategy per rendere questa modifica meno invasiva sul codice. Inoltre riguardando le classi di view da me realizzate mi rendo conto della verbosità introdotta dall'utilizzo del GridBagLayout che sono sicuro possa essere sostituito con qualche layout più efficiente. Tuttavia l'ho ritenuto migliore rispetto ad un BorderLayout in quanto mi ha permesso tramite Constraint e Insets di gestire più aspetti sul posizionamento dei componenti. Concludo dicendo che avrei voluto vedere realizzato il multiplayer che purtroppo per ragioni di tempo abbiamo deciso di non implementare essendo comunque stato inserito fra i requisiti opzionali.

Andrea Rettaroli

Mi ritengo molto soddisfatto del lavoro svolto da me personalmente e dal team di ragazzi con cui ho collaborato per questo progetto, credo che siamo riusciti a centrare il nostro obbiettivo riuscendo a creare un prodotto ad alta giocabilità e portabilità. Sono molto soddisfatto che la fase iniziale di analisi generale per la quale abbiamo impiegato molto tempo abbia dato grandi risultati permettendoci di semplificare e dividere i problemi e il lavoro. Mi ritengo soddisfatto della qualità data al mio codice attraverso l'uso di quanto imparato a lezione ed in proprio. Un futuro obbiettivo che vedo molto vicino e facile da raggiungere a seguito delle impostazioni generali date alla nostra applicazione è lo sviluppo la creazione di una nuova modalità: Multiplayer.

Marta Spadoni

Sono abbastanza soddisfatta del lavoro svolto, anche se riconosco che con altro tempo a disposizione saremmo stati in grado di realizzare una versione multiplayer, la quale avrebbe reso il gioco più avvincente.
Nel corso dello sviluppo del progetto mi sono ritrovata spesso ad aiutare gli altri componenti del gruppo e forse questo mi ha tolto del tempo che avrei potuto impiegare per migliorare le mie parti. In particolare, per quanto riguarda la gestione delle collisioni mi sarebbe piaciuto implementare una versione più sofisticata magari sfruttando una qualche libreria messa a disposizione online. Inoltre, avrei voluto gestire i collegamenti fra le entità, ora modellati facendo un uso intensivo del GameController, attraverso l'uso del pattern Observer e della logica ad eventi. Ritengo comunque che data la modellazione fatta non sarà troppo oneroso apporre questi cambiamenti in futuro.

Lucia Sternini

Mi ritengo abbastanza soddisfatta del mio lavoro. Renderei più snella la classe ScoreHandler, scorporando da essa il controllo dell'esistenza di un giocatore e l'eventuale aggiornamento. Inoltre, sarebbe stato interessante inserire un login e memorizzare per ciascun giocatore la storia dei punteggi. Infine, mi sarebbe piaciuto effettuare un upgrade del mio codice per adattarlo ad una versione multiplayer del gioco.
Per quanto riguarda il lavoro di gruppo nella fase di analisi, ritengo sia stato realizzato efficacemente portando così alla realizzazione di buon progetto.

Appendice A

Guida Utente

All'avvio dell'applicazione viene mostrato il main menù (figura 1)

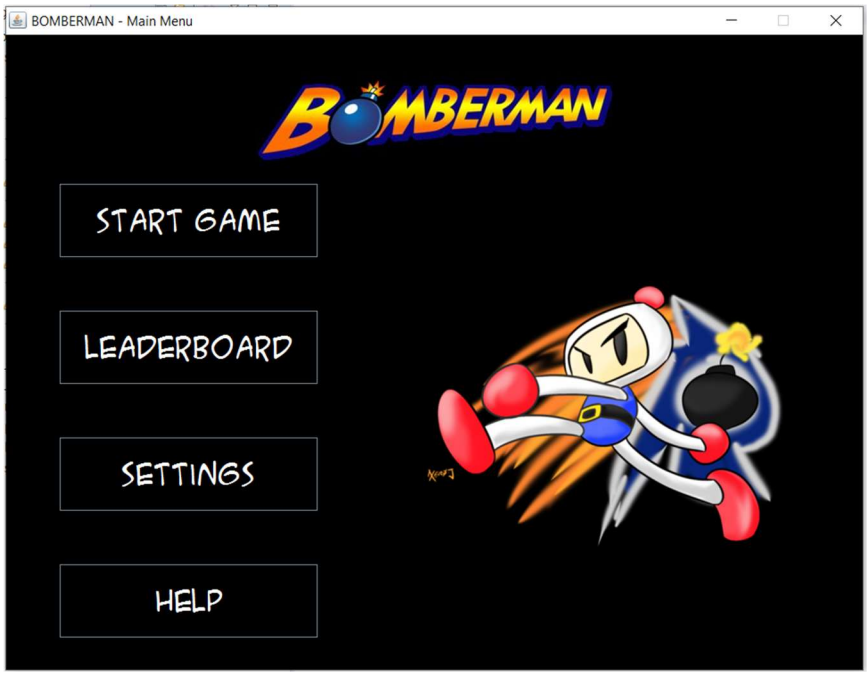

Figura 1: Screenshot MainMenù.

Le opzioni selezionabili tramite click del mouse sono le seguenti:

“Start Game” dà inizio ad una partita.

“Settings” riporta ad un menù opzioni all’interno del quale è possibile selezionare o deselezionare musica ed effetti.

“Leaderbord” mostra la classifica dei migliori giocatori salvati per nome, rank, punteggio, livello e il tempo registrato per completarlo.

“Help” mostra i comandi di gioco e una breve spiegazione su come affrontare la partita.

Nella figura 2 viene mostrato un momento di gioco, l’utente può direzionare l’eroe attraverso le arrow keys e piazzare una bomba usando lo “space”. Il tasto “P”, invece, mette in pausa il gioco.

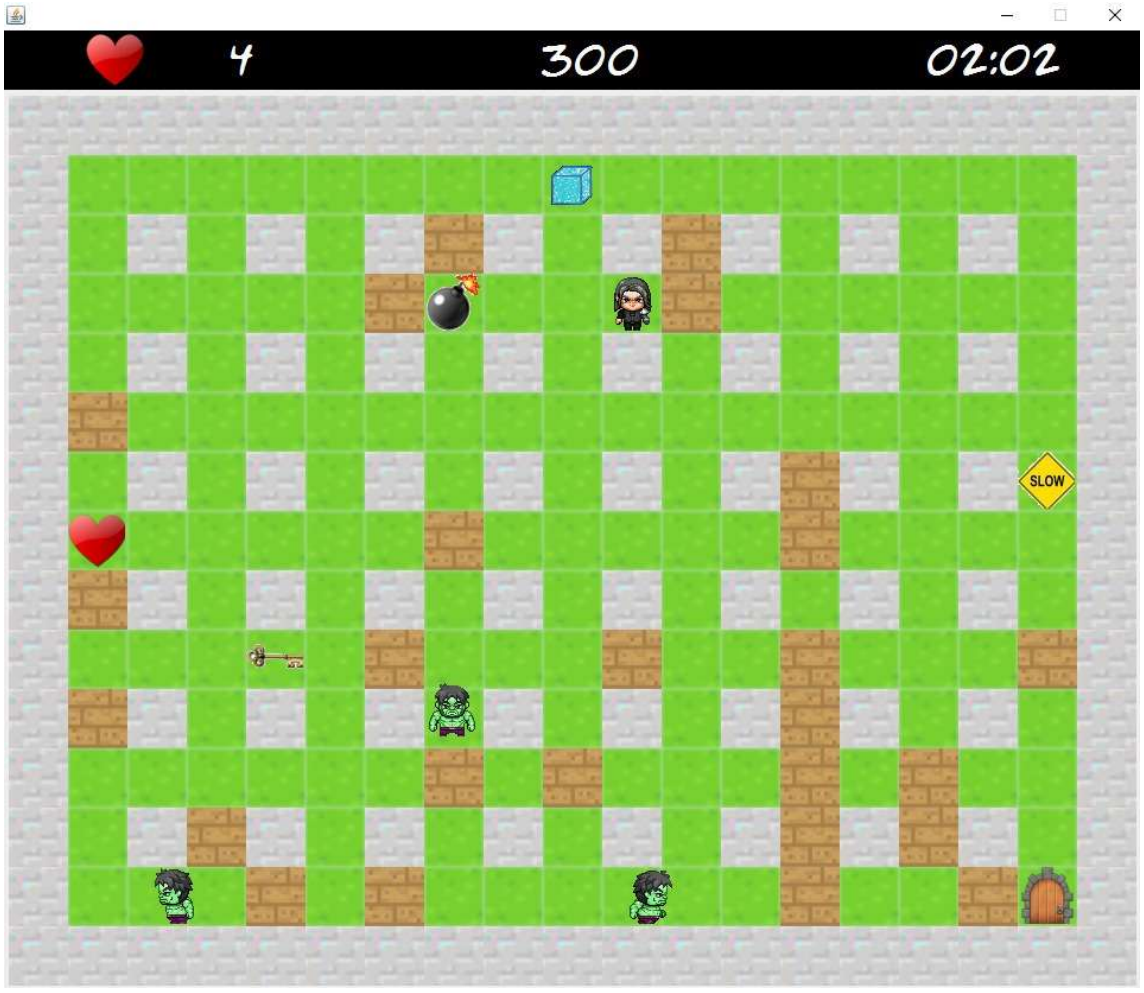

Figura 2: Schermata del gioco in azione con key, bomba, malus e bonus visibili.