UNIVERSITÀ DEGLI STUDI
DI PERUGIA

Tesina Finale di

Programmazione di Interfacce Grafiche e Dispositivi Mobili

Corso di Laurea in Ingegneria Informatica ed Elettronica – A.A. 2018-2019

Dipartimento di Ingegneria

docente Prof. Luca Grilli

Lazy Thief

applicazione desktop JavaFx

304709 **Locatelli Vincenzo** vincenzo.locatelli@studenti.unipg.it

Indice generale

1 Descrizione del Problema

L'obiettivo di questo progetto è la realizzazione di un'applicazione/gioco per computer chiamata Lazy Thief.

Il protagonista, il “ladro pigro”, è seduto in punto della schermata di gioco e lancia delle palline di colore rosso, chiamate “Bouncing red balls”. L'obiettivo del ladro è colpire una gemma situata in un altro punto del gioco.

Le “Bouncing red balls” possono essere lanciate a velocità differenti, ma una volta lanciate seguono l’andamento del moto rettilineo uniforme.

Il giocatore ha un numero limitato di palline a disposizione per colpire le gemme: quando le “bouncing balls” sono esaurite, o il giocatore raggiunge le gemme massime del gioco, la partita è terminata.

Tra il ladro e la gemma sono situati dei blocchi con assegnate le rispettive vite. Se la “bouncing ball” colpisce un blocco, rimbalza e cambia la sua trattoria.

Quando una gemma viene colpita da una pallina viene incrementato il punteggio del giocatore e cambia la disposizione dei blocchi e della gemma.

L'obiettivo è totalizzare più gemme possibile con le palline a disposizione all'interno di tre diversi scenari di gioco.

2 Specifica dei requisiti

Si intende sviluppare l'applicazione con JavaFX.

I requisiti vengono divisi per comodità in requisiti di gioco e requisiti utente

REQUISITI UTENTE

• Possibilità di creare un nuovo profilo per il giocatore

• Possibilità di riprendere una partita da dove si è lasciata, caricando il profilo del giocatore

• Visualizzare in qualunque momento le informazioni del giocatore attivo, quali “Bouncing Red Ball” rimanenti, attuali gemme nelle partita e record del giocatore

• Consultare la classifica di tutti i giocatori con i relativi record (la partita migliore del giocatore viene automaticamente registrata come record)

REQUISITI DI GIOCO

• L'interazione dell'utente con il gioco si limita all'uso del mouse. L'utente lancia la "Bouncing red ball" trascinando il mouse in direzione della gemma. Viene creata in automatico una linea che mostra la direzione in cui verrà lanciata la pallina, al rilascio del mouse. È possibile regolare la potenza di lancio, proporzionale appunto alla lunghezza della linea tracciata dal giocatore

•Rilevazione delle collisioni tra la pallina e i blocchi nella schermata di gioco

• Gestione di un opportuno angolo di rimbalzo quando la "Bouncing ball" impatta uno dei blocchi

• Animazione per i blocchi

• Fluidità del movimento della Bouncing ball durante il lancio

• Gestione di sottofondo musicale e effetti sonori quando si colpisce la gemma o i blocchi

• Struttura tabellare con ordinamento decrescente nella classifica dei giocatori

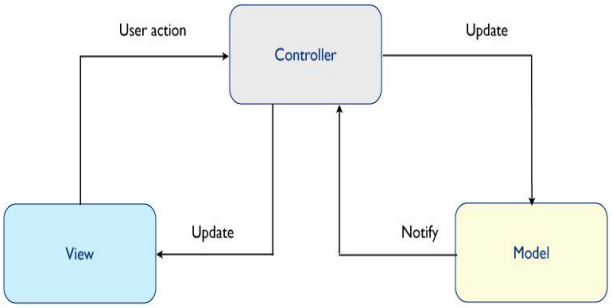
• Creazione di 3 scenari con diverse disposizioni dei blocchi

3 Progetto

Viene ora descritta la struttura dell'applicazione realizzata, illustrandone prima l'architettura software per poi scendere nel dettaglio dei blocchi funzionali che la compongono.

Architettura software

I pattern di programmazione scelti sono Model View Controller e Singleton . Quest'ultimo è design pattern di tipo creazionale. La sua funzione è quella di garantire, all'interno di un ambiente software, un'unica istanza di una determinata classe.



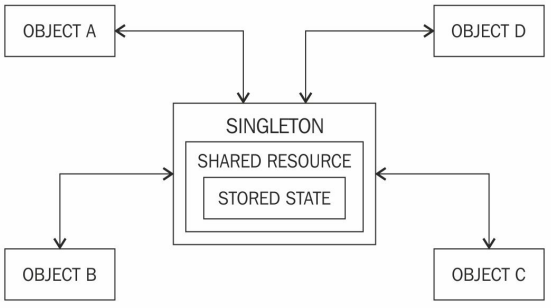


Figura 1 **Figura 2**

Il Model si occupa di gestire le informazioni che codificano lo stato dell'applicazione, quindi i parametri del giocatore come nome, numero di “Bouncing balls” disponibili, gemme collezionate nell'attuale partita e record.

Il View si occupa dell’interfaccia grafica e del posizionamento dei componenti, come le immagini negli scenari e rendering delle animazioni.

Il Controller gestisce la logica dell'applicazione, cioè il meccanismo di lancio delle “Bouncing balls” e le loro collisioni.

Package LazyThief

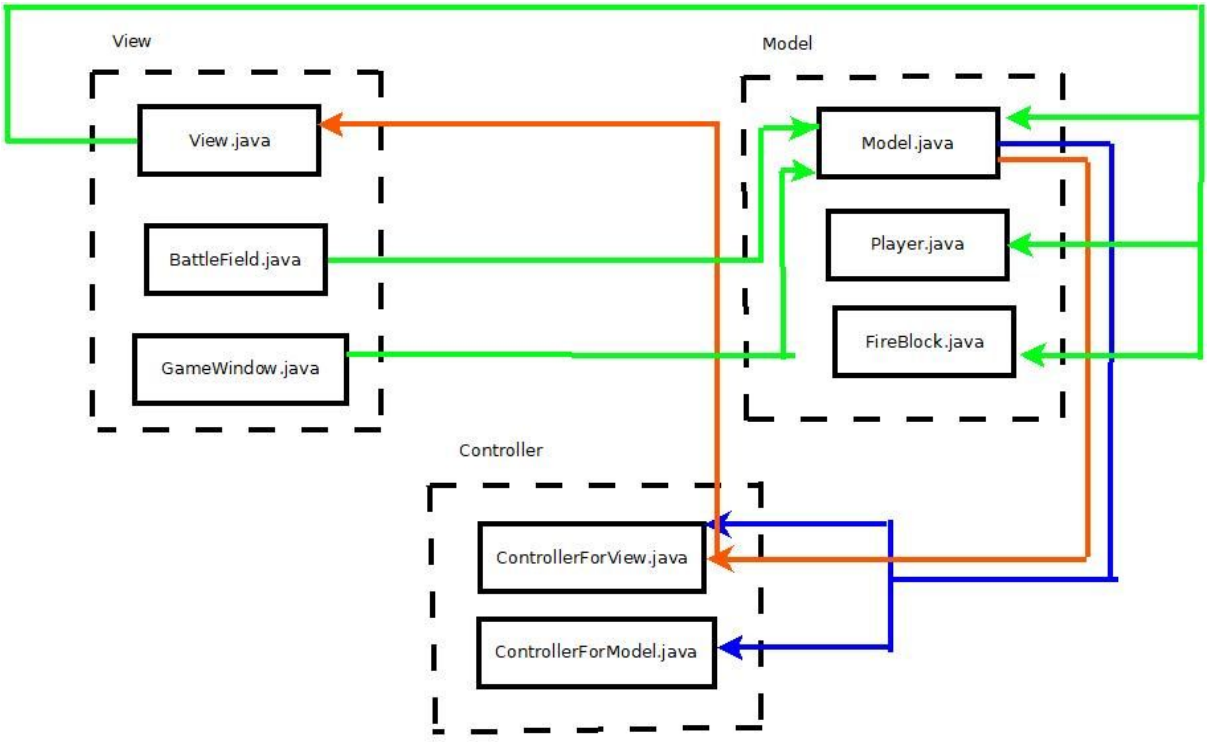


Figura 3: architettura dell’applicazione

Oltre alle classi rappresentate nel precedente schema l'applicazione contiene altre classi nel package Utils che consentono di gestire la classe Properties, classi di lettura e scrittura per file csv e classi per gestione delle immagini.

Model (LazyThief.model)

Le classi appartenenti al blocco Model di LazyThief si trovano nel package LazyThief.model. La loro struttura è rappresentata nel diagramma UML in figura 4.

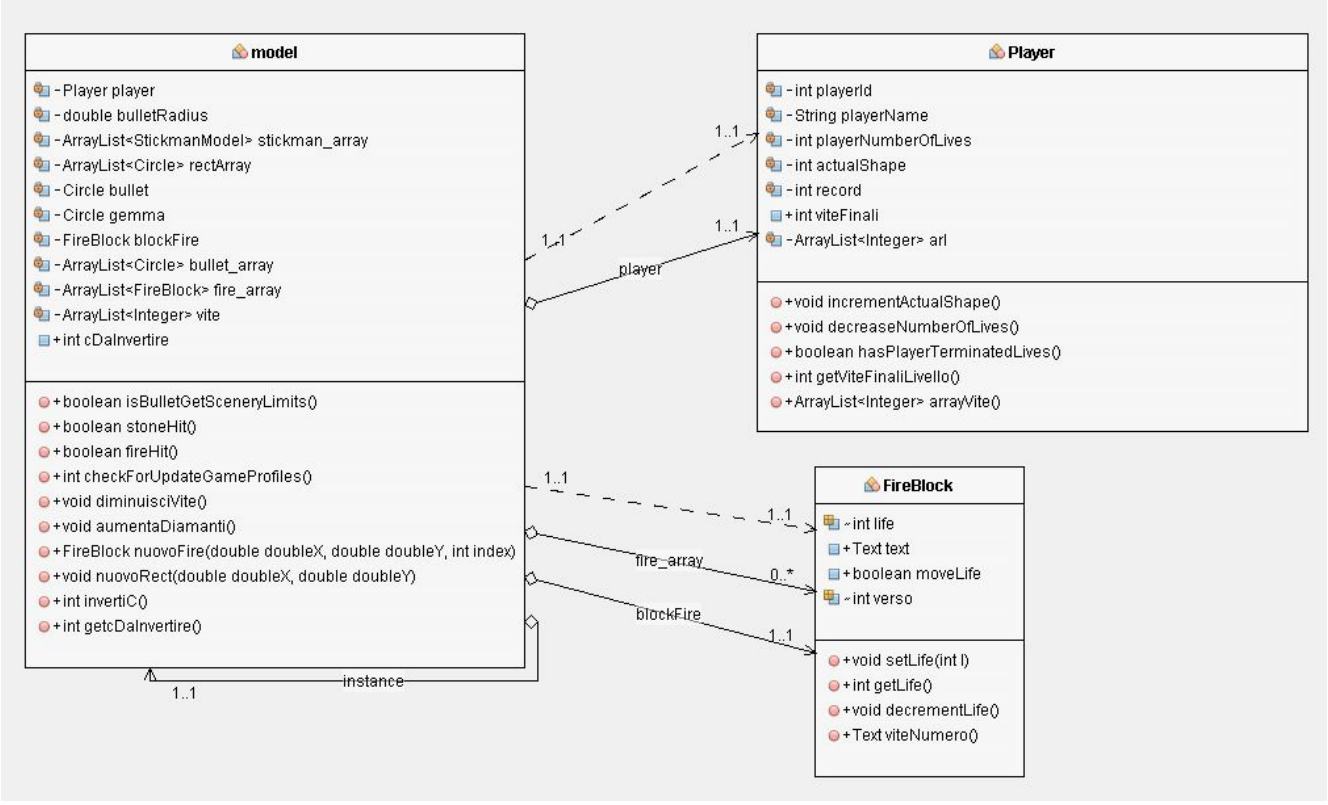


Figura 4 : LazyThief.model

In particolare:

La classe **Player.java** contiene tutti i parametri del giocatore e alcuni metodi come *incrementActualShape()* per incrementare le gemme accumulate durante la partita, o *decreaseNumberOfLives()* per indicare che una pallina è stata lanciata da quel giocatore.

La classe **FireBlock.java** è la classe che consente di istanziare un oggetto blocco

- *life* sono le vite del blocco
- *moveLife* indica se al blocco è associata un'animazione
- *verso* indica eventualmente il verso orizzontale o verticale dell'animazione

Nella classe **Model**

- ***bullet*** è la pallina lanciata dal giocatore. Si tratta di un'istanza della classe *Circle*
- ***gemma*** sono le pietre azzurre disperse negli scenari che devono essere colpite. Si tratta di un'istanza della classe *Circle.*
- ***blockFire*** sono i blocchi che possono essere colpiti dalla pallina. Si tratta di un'istanza della classe *FireBlock* , la quale estende la classe *Rectangle*

Il metodo ***stoneHit()*** controlla se la pallina intercetta una delle pietre azzurre durante il suo tragitto e provvede a incrementare il punteggio del giocatore nel model. Richiama metodi della classe View per aggiornare la grafica.

Il metodo ***blockHit()*** rileva le collisioni tra la pallina e i blocchi. Utilizza il metodo ***invertiC***() per capire in che modo deve rimbalzare la pallina a seconda del punto in cui viene colpito il blocco. Controlla se le vite del blocco sono terminate ed eventualmente lo rimuove nell'array dei blocchi nel model e poi dalla grafica richiamando un metodo nel package View.

Il metodo ***checkForUpdateGameProfiles*****()** controlla se il giocatore ha totalizzato un record migliore rispetto ai suoi record precedenti in uno di questi due casi

- ❏ se ha terminato le sue vite/palline
- ❏ se è riuscito a completare tutti gli scenari

Il metodo ***invertiC***() si occupa della gestione dei rimbalzi. Si riporta sotto un esempio di rimbalzo sul lato sinistro di un blocco quando avviene la collisione.

```
@Override
public int invertiC() {
    int b = 0;

    Bounds rectangleBounds = model.getInstance().getBlockfire().getLayoutBounds();

    boolean rectangle_left = ((getPlayerBullet().getLayoutX()) + 85) <= (rectangleBounds.getMinX());
```

Figura 5

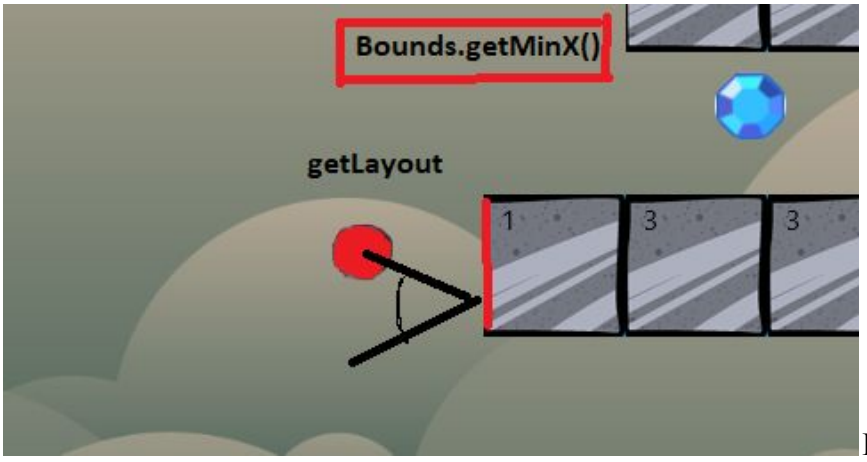


Figura 6

In figura 6 viene mostrato l'attimo precedente in cui la pallina urterà il blocco.

L'oggetto *rectangleBounds* restituisce i limiti in termini di coordinate del blocco in che stiamo prendendo in considerazione. Quando avviene la collisione, se la coordinata X della pallina è minore di *rectangleBounds.getMinX(),* cioè trova a sinistra rispetto al blocco, dovrà invertire una componente della sua velocità, per cambiare direzione. In questo caso sarà la componente orizzontale.

Un altra classe presente nel package model si chiama **PlayerData.java** di cui è riportato il diagramma UML in figura 7 sotto.

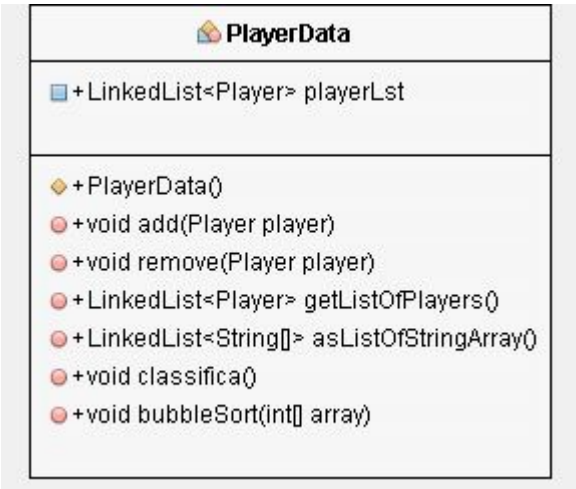


Figura 7

L'attributo principale *playerLst* è definito di classe LinkedList. Si tratta di una lista concatenata e ordinata in cui la posizione dei singoli elementi è importante. In questo caso è un lista di oggetti di classe Player, quindi con il metodo *add*() vengono aggiunti i nuovi profili utenti creati.

Un metodo significativo è *asListOfStringArray*(). In questo metodo si utilizza la precedente lista *playerLst* per creare una LinkedList di stringhe in cui, oltre ai parametri del giocatore, vengono accodate stringhe contenenti le vite di tutti i rispettivi blocchi del livello a cui il giocatore è arrivato. Quello che restituisce questo metodo è un array di stringhe per ogni giocatore. Queste informazioni verranno poi salvate dentro un file csv e aggiornate ogni volta che il giocatore va avanti nella partita. In questo modo è possibile creare una struttura di "memoria" e recuperare i salvataggi del giocatore. Per esempio si può tenere traccia di quali blocchi ha colpito un giocatore nel livello a cui si interrotto.

ControllerForView(LazyThief.controllerForView)

Le classi appartenenti al blocco ControllerForView di LazyThief si trovano nel package LazyThief.controllerForView . La loro struttura è rappresentata nel diagramma UML in figura 8.

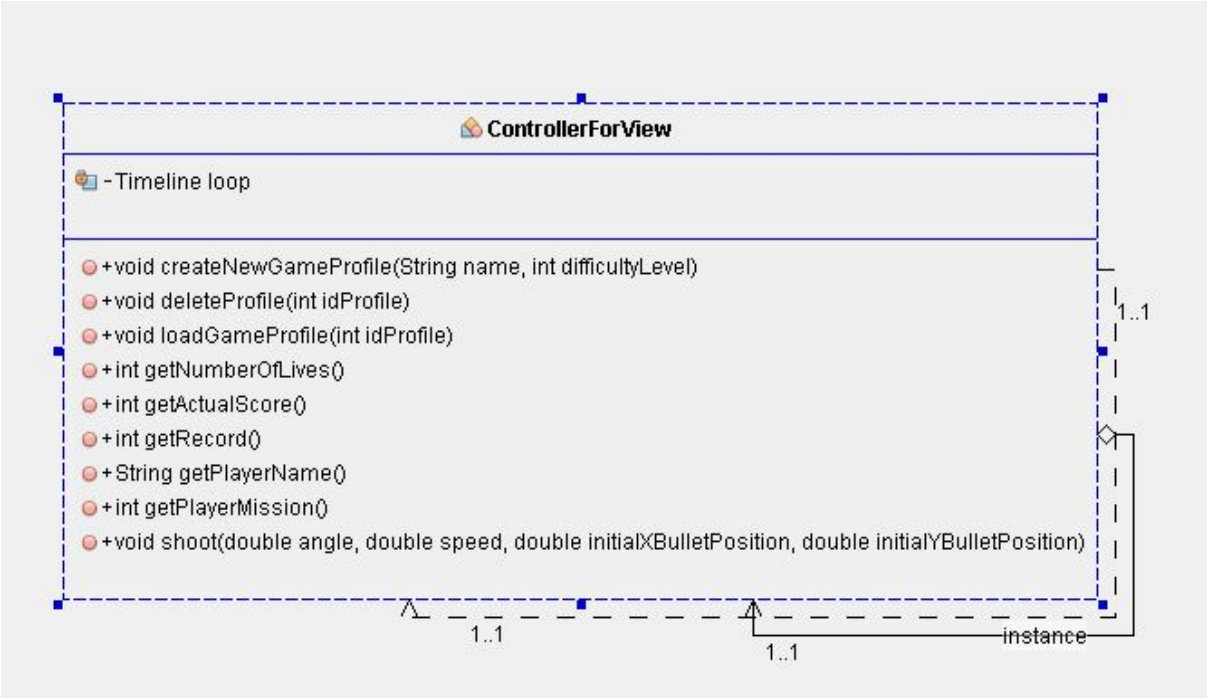


Figura 8: LazyThief.controllerForView

Il metodo ***shoot()*** si occupa del movimento della pallina e viene lanciato da una classe del package View quando quando si rilascia il mouse. Ha come parametri la velocità e l'angolo di lancio i quali vengono calcolati dalla classe GameWindow, spiegata successivamente.

Nel metodo la velocità viene scomposta nelle due componenti vx e vy per indirizzare la traslazione. Per lo spostamento della pallina si è scelto di usare la classe *TimeLine*. La classe *Bounds* viene invece usata per rilevare la collisione con le pareti o con gli oggetti nella schermata di gioco. Ogni volta che vengono rilevate delle collisioni, il metodo si occupa di invertire la componente della velocità orizzontale o verticale a seconda dei casi, per un opportuno angolo di rimbalzo uguale a quello di incidenza .

View(LazyThief.View)

Le classi appartenenti al blocco View di LazyThief si trovano nel package LazyThief.view . La loro struttura è rappresentata nel diagramma UML in figura 9.

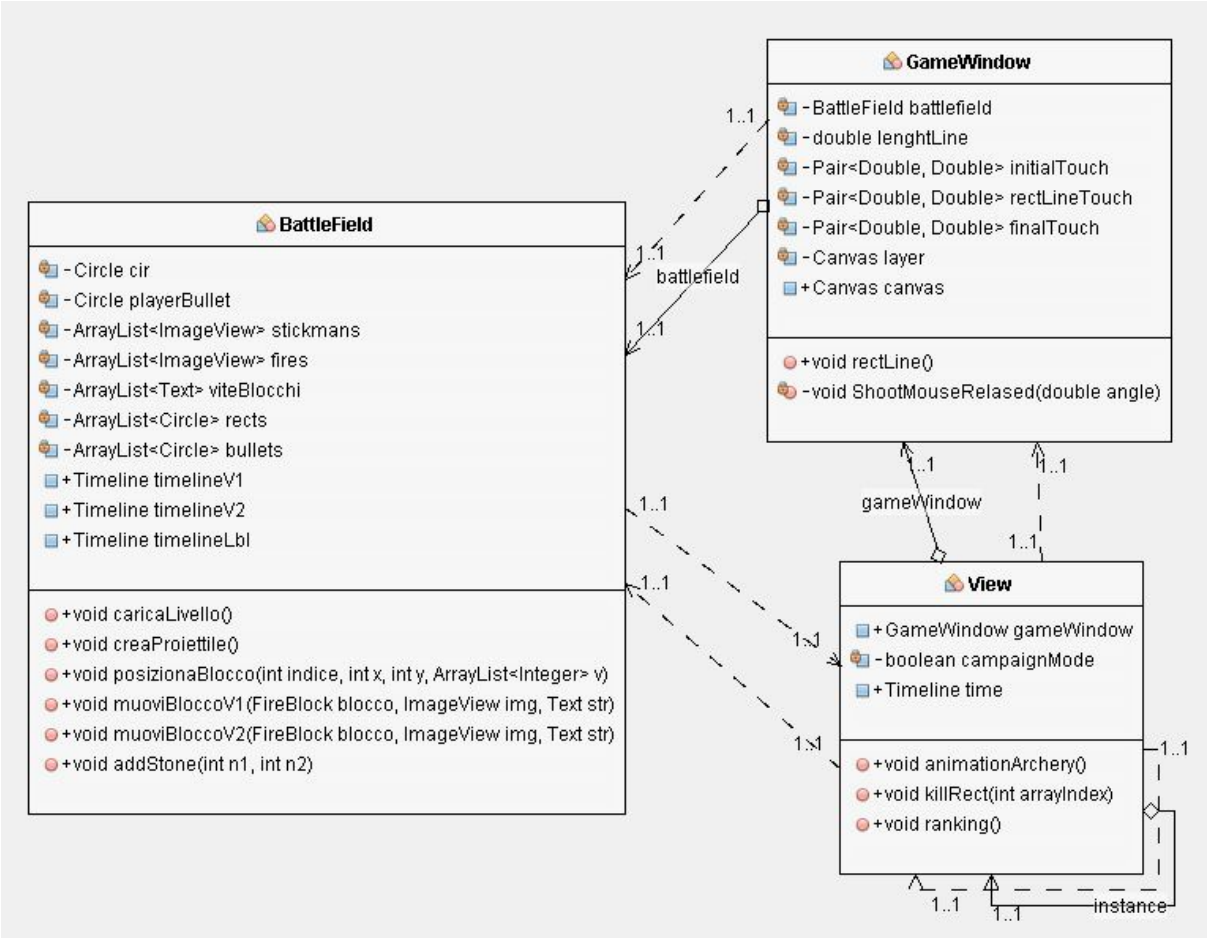


Figura 9: LazyThief.View

In particolare:

La classe **Battlefield.java** comprende tutti gli elementi della schermata di gioco dal punto di vista grafico. Ad esempio, se nel model viene istanziato un oggetto blocco, il metodo *posizionaBlocco*() della classe Battlefield.java posiziona nelle stesse coordinate un'immagine che corrisponde al contenuto del blocco.

Stessa cosa fa il metodo *addStone*() per le gemme che deve colpire il giocatore.

Le collisioni quindi vengono rilevate solo dal punto di vista logico. In caso di aggiornamenti futuri, come aggiunta di blocchi con caratteristiche differenti, sarebbe piuttosto semplice da implementare. Basterebbe associare al blocco la nuova immagine, ma la parte relativa ai metodi che gestiscono i rimbalzi non andrebbe modificata.

La classe **GameWindow.java** si occupa di gestire l'interazione con l'utente. Infatti aggiunge un ascoltatore di eventi che rileva eventi di MOUSE_PRESSED, MOUSE_DRAGGED e MOUSE_RELEASED. Quando il giocatore vuole effettuare un lancio trascina il mouse nella direzione scelta. Il metodo *rectLine*() calcola la lunghezza della linea e l'angolo con cui viene tracciata. I due parametri saranno poi passati al metodo *shoot*() che abbiamo già descritto

sopra. La lunghezza sarà proporzionale alla velocità della pallina e l'angolo servirà per scomporre la velocità in componente orizzontale e verticale.

La classe **View.java** consente all'untente di cambiare finestra dell'applicazione per iniziare nuove partite o riprendere partite già iniziate con il metodo *prepareSceneToShowWindow()*.

Problemi riscontrati

Il gioco, nonostante la sua semplicità ha richiesto un tempo notevole per la gestione della parte di salvataggio dei dati, perchè a ogni giocatore sarebbe stato associato un array per contenere le vite dei blocchi di un singolo livello. Questo va ripetuto per tutti livelli che ha affrontato quel giocatore. Ho pensato di semplificare in modo ragionevole tenendo conto che, in una partita, interessa tenere traccia solo delle vite relative ai blocchi nel livello a cui il giocatore è arrivato. Infatti quelli precedenti sono stati ormai "vinti" dal giocatore e non è più possibile affrontarli. Con questo tipo di ipotesi a ogni giocatore è associato un array di numeri, anzichè una matrice.

L'altro difficoltà riscontrata è stata nella gestione dei rimbalzi della pallina contro i blocchi. Infatti inizialmente avevo usato lo strumento *AnimationTimer* per la traslazione della pallina. Ogni volta che veniva rilevata una collisione, il metodo calcolava un angolo di rimbalzo tramite la formula *arcotangente (vy/vx)* $-\pi/2$. Veniva poi fatta una chiamata ricorsiva al metodo stesso di lancio passando come parametri la nuova posizione della pallina e il nuovo angolo di lancio. Con questa gestione dei rimbalzi, il risultato non era ottimale perchè si presentavano dei bug nel rimbalzo. La soluzione con TimeLine, invertendo le componenti della velocità è risultata molto più efficace e meno onerosa di codice.

Possibilità di estensione e personalizzazione

In futuro è possibile aggiungere nuovi scenari con differenti caratteristiche dei blocchi. Si possono prevedere blocchi di materiali diversi e quindi con un numero di vite differente. È possibile anche aggiungere animazioni più complicate rispetto a quella orizzontale e verticale. Oltre alla modalità facile si può aggiungere modalità intermedia e una difficile, con Boucing red balls iniziali minori rispetto a quelle attuali.

Bibliografia

https://docs.oracle.com/javase/8/javafx/api/javafx

https://code.makery.ch/it/library/javafx-tutorial/