BRICKS DESTROYER

Tesina di Elementi di informatica grafica

Studente: Belal Mehdi

SOMMARIO

1. DESCRIZIONE DEL PROBLEMA

Questo progetto è finalizzato allo sviluppo in ambiente Java di un'applicazione grafica ispirato ai giochi arcade Arkanoid e Breakout.

1.1 Il gioco: Arkanoid e Breakout

In questo genere di giochi lo scopo è abbattere un muro di mattoni posto nella parte superiore dello schermo, in basso c'è solamente una piccola barra che può essere mossa a destra e sinistra; con questa bisogna colpire una palla che rimbalza, in modo che distrugga tutti i mattoni che compongono il muro. Se il giocatore non riesce a colpire la palla con la propria barra, questa esce dalla schermata e si perde una vita, perse tutte le vite si perde il gioco; mentre volta abbattuti tutti i mattoncini il giocatore passa al livello successivo. Viene assegnato un punteggio ogni volta che viene colpito un mattoncino, in base alla tipologia dello stesso, ed ogni volta che ci si aggiudica un power-up (nel caso di Arkanoid).

1.2 L'applicazione Bricks Destroyer

Bricks Destroyer non è una fedele riproduzione delle due applicazioni da cui trae ispirazione. Obbiettivo del giocatore è il superamento dei 30 livelli presenti e la totalizzazione del massimo punteggio possibile, distruggendo tutti i mattoncini presenti nel livello senza perdere tutte le vite, pena la perdita della partita; da tenere in considerazione il fatto che il punteggio ottenuto è indipendente dalla difficoltà del livello: un livello estremamente difficile può far totalizzare un punteggio non molto alto, ciò che conta è l'abilità nel riuscire a superare il massimo numero di livelli possibile con le sole vite a disposizione. Il giocatore interagisce solamente tramite il *Paddle,* ovvero la barra con cui bisogna evitare di far cadere la pallina sul fondo dello schermo, per il resto il movimento della pallina è governato dalla topologia con cui i mattoncini sono disposti. Ogni volta che la palla colpisce un ostacolo essa rimbalza cambiando direzione, con un angolo pari all'angolo di incidenza; mentre quando la palla colpisce il paddle rimbalza in base all'angolo con cui questo viene colpito, il giocatore deve avere l'abilità di posizionare il paddle in modo che quando la pallina lo colpisce questa vada nella direzione desiderata.

Nella figura 1 viene mostrato il primo livello dell'applicazione.

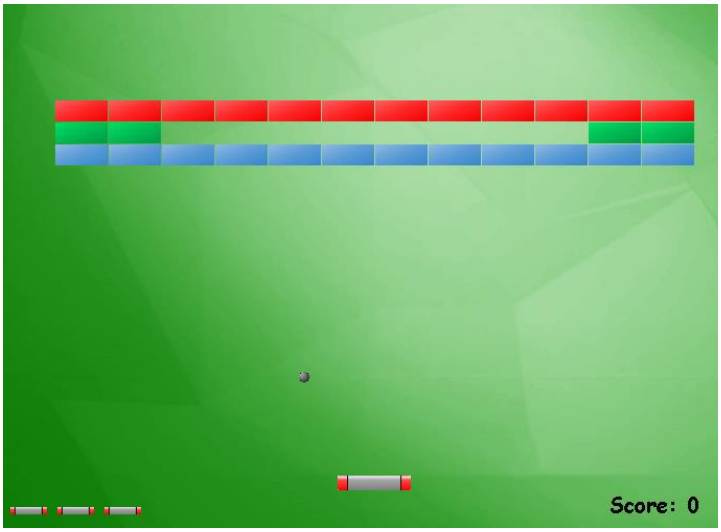


Figura 1

Tutte le specifiche del gioco vengono riportate al paragrafo successivo.

2. SPECIFICA DEI REQUISTI

2.1 Requisiti di gioco

- L'interazione con il giocare è gestita sia con l'uso del mouse che della tastiera, la barra "*Paddle*" segue il movimento del mouse sulla schermata di gioco, l'unico tasto della tastiera utilizzabile è lo spazio, utilizzato per iniziare un livello o per mettere il gioco in pausa
- L'interfaccia di gioco prevede solamente la schermata di gioco, per un'interazione più pulita, mettendo in pausa il gioco compaiono i tasti che permettono di tornare alla schermata principale o di riprenderlo
- Presenza di 30 livelli di gioco, gli elementi che discriminano la difficoltà di ogni livello sono la disposizione dei mattoncini e il tipo di mattoncini presenti
- Ogni volta che pallina incontra un ostacolo rimbalza con un angolo pari all'angolo con cui l'ha colpito, tranne nel caso del "*Paddle*" in cui l'angolo di rimbalzo dipende dal punto in cui lo colpisce, permettendo la direzionabilità della pallina.
- La velocità della pallina aumenta proporzionalmente al tempo impiegato per completare il livello
- Presenza di 3 vite, al termine delle quali il giocatore potrà decidere di rigiocare il livello perso oppure tornare al menu principale.

2.2 Requisiti dell'applicazione e di sistema

- Possibilità di mettere in pausa il gioco
- Possibilità di salvare una partita, con la possibilità di riprenderla in un secondo momento; ogni volta che si applica un salvataggio il precedente viene sovrascritto
- Possibilità di eliminare i progressi (punteggio migliore e salvataggio automatico)
- Presenza di una schermata "How to play", alla quale si può accedere dalla schermata principale; essa descrive le modalità del gioco e i suoi elementi, l'assegnazione dei punteggi, in italiano e in inglese

2.3 Requisiti aggiuntivi/facoltativi

- Presenza di musica di sottofondo e di effetti sonori nel gioco
- Possibilità di modificare il volume della musica di sottofondo e degli effetti sonori
- In aggiunta ai requisiti già presentati, per rendere l'applicazione più gradevole per l'utente sono state aggiunte alcune personalizzazioni grafiche: ovvero la possibilità di modificare lo stile dello sfondo di gioco e della pallina

3 PROGETTO

Viene ora descritta l'architettura del sistema software, con una prima descrizione generale, per poi descrivere i vari moduli e il loro funzionamento.

3.1 Architettura software

Per la realizzazione di questo progetto si è scelto di attenersi al pattern Model-View-Adapter, variante del Model-Controller-View in cui i blocchi Model e View non interagiscono direttamente ma comunicano solamente attraverso il Controller (fatta eccezione di alcuni casi che verranno illustrati nel seguito in dettaglio).

Il Model incapsula lo stato del gioco, comprendendo tutte le variabili che descrivono lo stato attuale del gioco. Il View, costituendo l'interfaccia grafica, è responsabile della visualizzazione delle informazioni grafiche, dell'interfacciamento tra le varie finestre e inoltre delle animazioni. Il Controller gestisce le informazioni ricevute in input tramite l'interfaccia messa a disposizione dal View, e le elabora attraverso i dati messi a disposizione dal Model.

Anche se dal punto di vista di un videogame questo pattern non è l'ideale, scopo principale di questa suddivisione è quello di realizzare un programma maggiormente ordinato ed improntato verso la logica della programmazione ad oggetti, infatti una struttura di questo tipo permette maggiori possibilità di futuri aggiornamenti e aggiunta di funzionalità. In figura 2 l'architettura generale dell'applicazione.

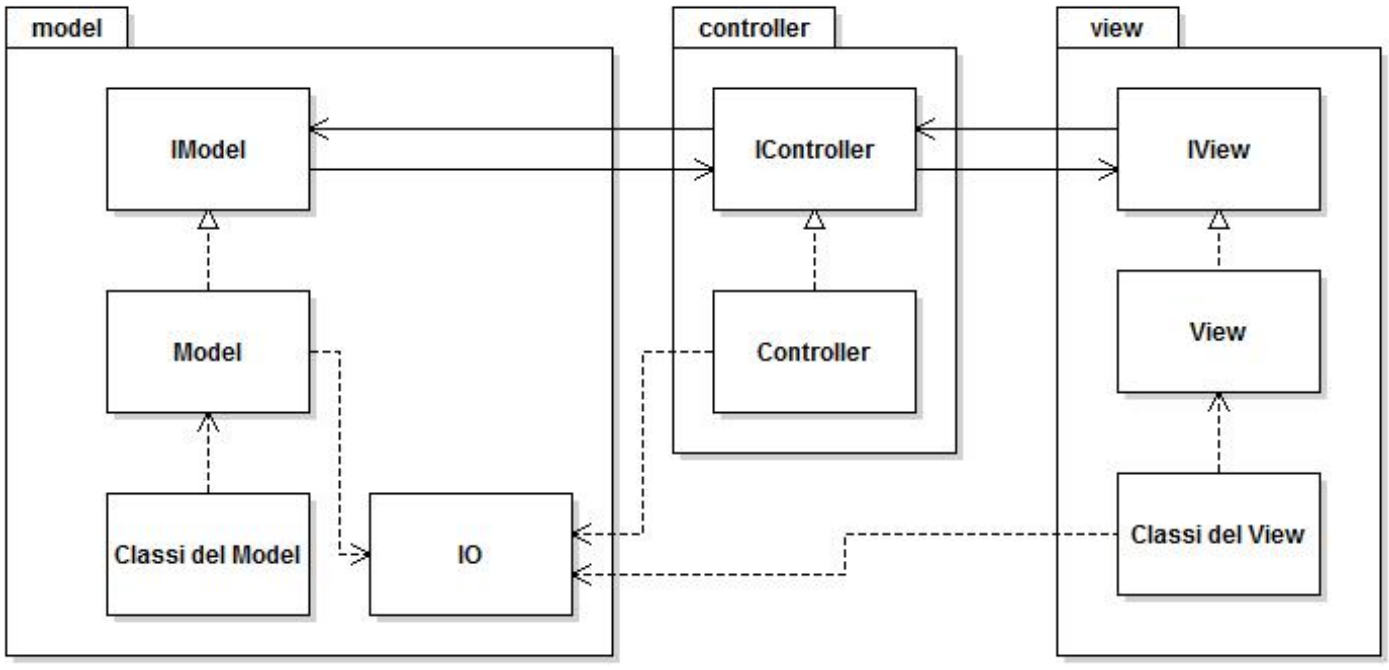


Figura 2

3.2 Descrizione dei moduli

- **MODEL**

La struttura del Model e delle classi che a questo fanno capo sono rappresentate nel diagramma che segue.

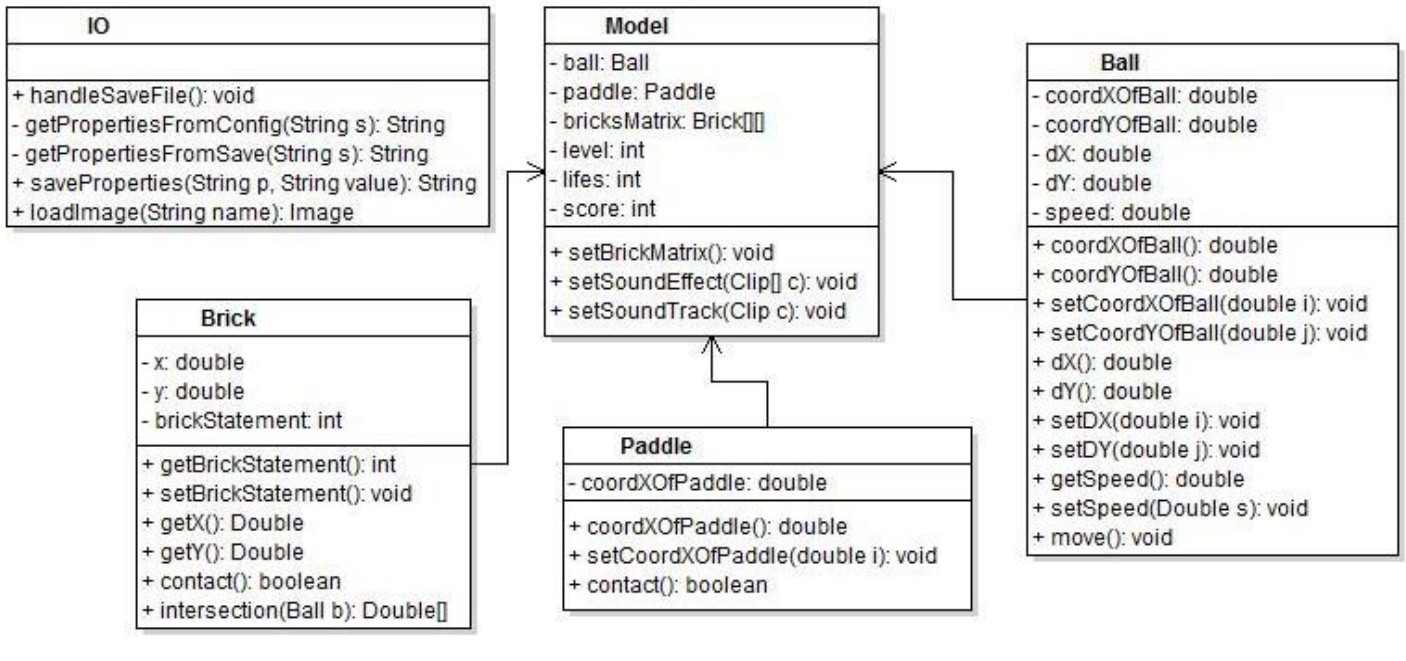


Figura 3

In particolare:

- **Model**: è la classe che si occupa di "incapsulare" e di tenere traccia di tutti gli elementi del gioco in esecuzione, oltre a contenere i metodi necessari per la loro gestione. Per la precisione nel Model vengono memorizzati le informazioni relative al gioco come: numero di vite (*lifes: int*), il punteggio (*score: int*), il livello attualmente in corso (*level: int*), e i vari oggetti istanziati dalle altre classi del package model (come i mattoncini, il *paddle* e la pallina).
  La cosa che discrimina un livello da un altro sono la disposizione dei mattoncini e il tipo di mattoncini presenti, questi vengono memorizzati nella variabile brickMatrix:Brick[][], che non è altro che una matrice di mattoncini $8 \; x \; 12$ in cui viene memorizzata la disposizione dei mattoncini a ogni livello. All'inizio di un livello ogni elemento della matrice viene inizializzato con il valore del mattoncino relativo.

La figura 4 sottostante mostra la disposizione relativa al livello 1.

0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
4	4	4	4	4	4	4	4	4	4	4	4
6	6	0	0	0	0	0	0	0	0	6	6
1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0

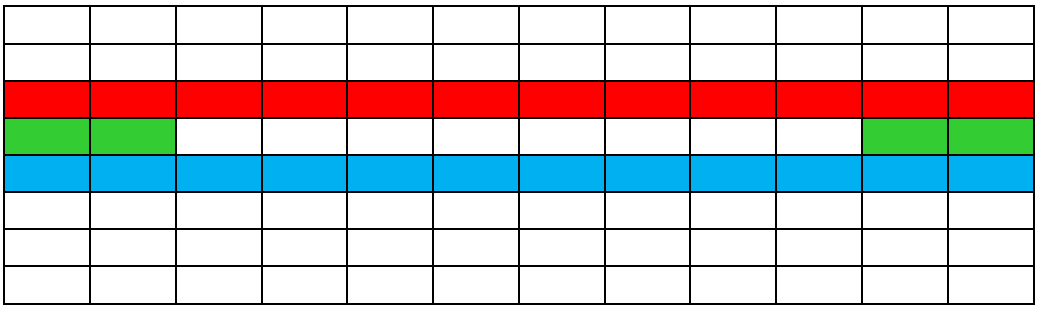


Figura 4

Oltre ai metodi descritti prima per l'accesso ai dati del gioco in esecuzione altri metodi di rilevanza sono:

- *setBrickMatrix(String[] s)*: è il metodo che si occupa effettivamente di impostare lo stato della matrice di mattoncini. In questo metodo si scandisce la *bricksMatrix* inserendo il valore dello stato di ogni mattoncino. Questo metodo viene invocato all'inizio di ogni livello, passandogli come parametro un array, che rappresenta la disposizione dei vari elementi di un particolare livello o del livello salvato.
- Altri metodi come *setSoundTrack(Clip c)* e *setSoundEffect(Clip[] c),* sono responsabili di inizializzare gli elementi di tipo Clip che rappresentano la musica di sottofondo e gli effetti sonori del gioco.

- **IO**: questa classe si occupa di mettere a disposizione dei metodi di Input/Output per gestire la lettura dei vari livelli, il salvataggio delle partite e il loro recupero, e alcune operazioni di Input/Output di aspetto grafico. I metodi della classe IO.java vengono utilizzati anche da alcune classi del package view che hanno bisogno di importare immagini; anche se questa scelta va in contraddizione con la politica del pattern Model-View-Adapter (in cui le classi di Model e di View comunicano per mezzo del Controller), si è optato per questa scelta per non aggravare questa operazione al Controller, inoltre si tratta di operazioni saltuarie che non restituiscono alcun dato necessario al Controller per la gestione del gioco. i metodi più rilevanti sono:

  - *createSaveFile()*: questo metodo è il primo che viene richiamato dal metodo main dell'omonima classe; si occupa essenzialmente di creare il file di salvataggio, se non è già presente, in cui verranno salvati (eventualmente) i dati di gioco, questa operazione viene fatta preliminarmente in quanto altre parti del programma si basano sull'esistenza di questo file (come il metodo che stampa sulla finestra d'avvio il punteggio migliore).
  - *getPropertiesFromConfig(String s)* e *getPropertiesFromSave(String s)*: sono i due metodi fondamentali che si occupano di ottenere tutti i dati relativi a un livello nuovo o che è stato salvato, essi vengono richiamati da altri metodi della stessa classe che ottengono i dati necessari per il caricamento di un livello (come per esempio *getBricksStatementsAtLevel(String s)*)
  - *saveProperties (String p, String value)*: questo metodo si occupa invece del salvataggio dei dati di gioco, esso viene richiamato da tutti i metodi necessari per il salvataggio dei dati.
  - *loadImage(String name)*: questo metodo restituisce un'immagine, viene richiamato da alcune classi del package view che necessitano del caricamento di un'immagine.
- **Ball**: è la classe che istanza l'oggetto di tipo pallina, la classe Ball è dotata di attributi propri come la velocità, lo spostamento *dX* è *dY* e la posizione, e dei metodi da invocare su un oggetto di tipo Ball; oltre ai soliti metodi per ottenere i dati relativi all'oggetto (*getSpeed(), setSpeed()*, …);
  - *move()*: è il metodo che si occupa di aggiornare la posizione in base ai valori dello spostamento.
- **Brick**: questa classe istanza oggetti di tipo *brick,* essi possiedono attributi propri come *x: double* e *y: double* che rappresentano la posizione dell'estremo superiore destro di ogni mattoncino, la lunghezza e l'altezza sono definite come costanti, essendo uguali per ogni *brick*. Un altro attributo importante è *brickStatement: int*, esso permette di distinguere la tipologia del mattoncino (indistruttibile, colorato, distruttibile con più colpi), e lo stato del mattoncino (distrutto, integro), come abbiamo visto i valori di *brickStatement* di ogni elemento viene caricato all'interno della *brickMatrix* della classe Model, che rappresenta la disposizione e lo stato dei *bricks* del livello in corso. I metodi più importanti di questa classe sono:
  - *contact(Ball b)*: questo metodo restituisce true se il mattoncino viene colpito dalla pallina, e serve proprio per gestire le collisioni, esso è una rielaborazione del metodo *intersects(double x, double y, double w, double h)* della classe Ellipse2D.java del package java.awt.geom.
  - *intersection(Ball b)*: questo metodo serve per gestire il "rimbalzo" della pallina dopo aver colpito un *brick*. Esso restituisce una coppa di valori double memorizzati in un array, che rappresenta le dimensioni del rettangolo di intersezione tra la palla e il brick. Questo metodo è una rielaborazione del

metodo *intersect(Rectangle2D src1, Rectangle2D src2, Rectangle2D dest)* della classe Rectangle2D.java del package java.awt.geom.

- **Paddle**: questa classe istanza l'oggetto che rappresenta il *paddle*, l'unico parametro ritenuto importate per questo tipo di oggetto è *coordXOfPaddle: double*, lui solo è sufficiente a determinare la posizione del paddle in quanto si muove solamente lungo l'asse orizzontale, le sue dimensioni e gli altri suoi parametri di posizione sono costanti.
    - *contact(Ball b)*: anche questo oggetto presenta questo metodo per rilevare una collisione tra la pallina e il paddle; in realtà il metodo *contact()* della classe Paddle ha un codice diverso da quello della classe Brick (meno elaborato, poiché la pallina può colpire un solo *paddle*, mentre ci sono molti *bricks*).

- **CONTROLLER**

La struttura del package controller e delle sue classi è rappresentato nel seguente diagramma UML in figura 5.

Figura 5

Questa porzione dell'applicazione è formata solamente da una classe e un'interfaccia, nonostante la semplicità gestisce gran parte dell'attività dell'applicazione:

- **Controller**: questa classe, oltre a permettere la comunicazione tra gli elementi di model e di view gestisce la dinamica del gioco stesso. I metodi di rilevanza sono:
    - *checkCollision():* questo metodo gestisce la dinamica del gioco, si occupa di rilevare le collisioni della pallina con i vari elementi del gioco (mattoncini, paddle, pareti,..) e li gestisce di conseguenza, può essere visto in diverse fasi:
        - *Gestione livello completato:* si valuta se sono ancora presenti mattoncini da colpire, in caso contrario il livello è stato completato e viene richiamato il metodo levelCompleted().
        - *Gestione contatto tra pallina e paddle:* si valuta se la palla colpisce il paddle con il metodo *contact(Ball b)* della classe Paddle, in caso affermativo vengono calcolate le nuove coordinate[1], e viene aumentata la velocità della pallina.
        - *Gestione contatto tra pallina e bricks:* si valuta se la palla colpisce un mattoncino con il metodo *contact(Ball b)* della classe Brick, in caso affermativo vengono calcolate le nuove coordinate, e le nuove direzioni della pallina attraverso il metodo *intersection(Ball b)* della classe Brick, che permette di determinare da che direzione è stato colpito il mattoncino. In seguito viene gestita l'eventuale eliminazione del mattoncino.
        - *Gestione contatto tra pallina e pareti:* si valuta se la pallina colpisce una delle pareti e in caso affermativo vengono calcolate le nuove direzioni.
    - *next():* questo metodo, seppur molto semplice, è quello che si occupa di muovere tutto il meccanismo di gioco, esso si occupa semplicemente di "muovere" la pallina (invocando il metodo *move()* della classe Ball sull'elemento *ball*), e invocare il metodo *checkCollision()* decritto precedentemente per determinare la prossima azione da fare. Questo metodo è azionato a ogni "clock" del timer.
    - Altri metodi come *startNewGame*, *startSavedGame*, *loadLevel*, etc… permettono il caricamento e il salvataggio, impostando i valori di ogni livello (di un nuovo gioco o di un gioco salvato) nel primo caso, e il salvataggio dei dati di gioco nel secondo caso.

[1] Vedi approfondimento in appendice

- **VIEW**

La struttura del package view e delle sue classi è mostrata nel seguente diagramma UML in Figura 6:

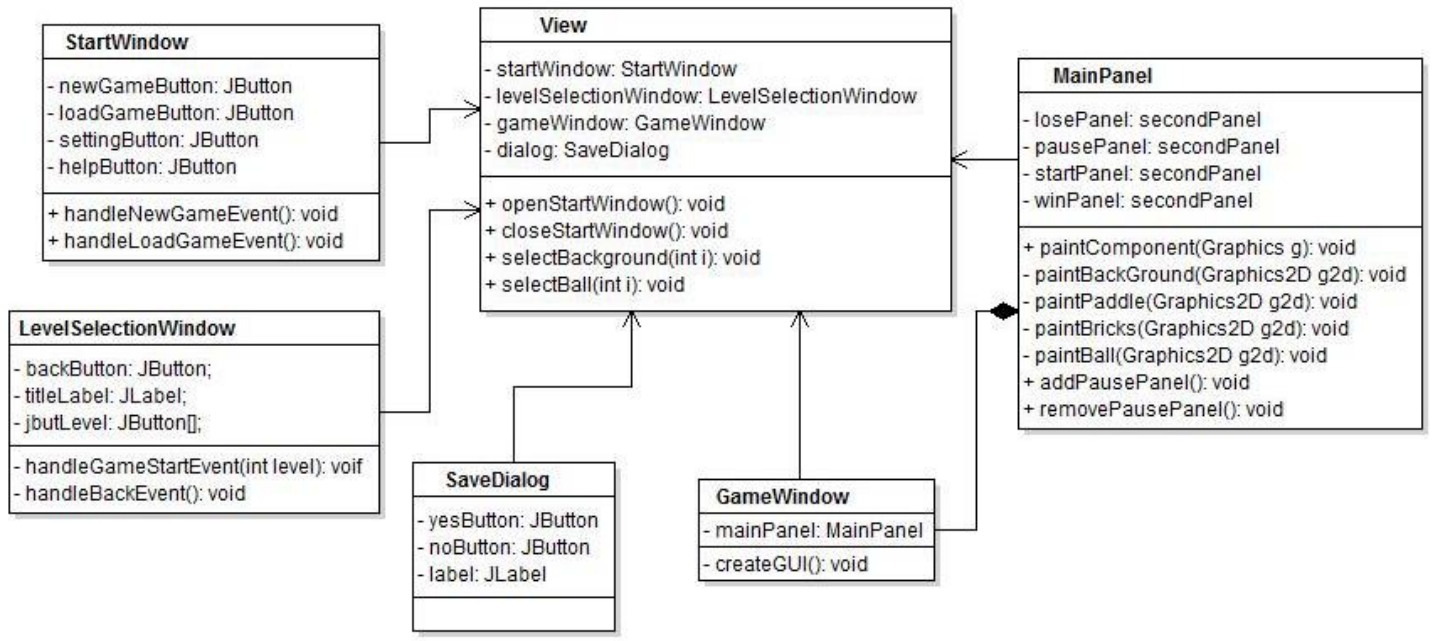


Figura 6

Le classi in questione servono a modellare l'interfaccia grafica con cui l'utente interagisce, esse sono classi che estendono componenti grafiche (JFrame, JPanel, JDialog), fatta eccezione per la classe View, e per IView, l'interfaccia che implementa.

- **View**: questa classe si occupa della gestione dei componenti grafici del gioco, istanziando o eliminando le varie finestre, si occupa inoltre di fornire degli accessi per l'aggiornamento di alcuni aspetti grafici, come per esempio l'applicazione della schermata di pausa.
- **StartWindow**: questa classe rappresenta la prima finestra che il giocatore incontra all'avvio del gioco, è una semplice finestra con 4 pulsanti che consentono di avviare un nuovo gioco, riprendere un gioco salvato, accedere alle impostazioni o accedere alla finestra "*How to play*", con le istruzioni per giocare.
- **LevelSelectionWindow**: questa classe rappresenta una finestra che permette di selezionare il livello da giocare, viene mostrata solamente se si sceglie di iniziare un nuovo gioco.
- **GameWindow:** questa classe rappresenta la finestra di gioco, e si occupa semplicemente di contenere il pannello di gioco (illustrato nel seguito), e di fornirgli alcuni metodi.
- **MainPanel**: questa classe estende il componente JPanel.java del package javax.swing delle API di java. Rappresenta il pannello di gioco, essa contiene i metodi e gli attributi necessari alla realizzazione della grafica del gioco. I vari elementi sono rappresentati

tramite delle immagini caricate con l'ausilio della classe IO del model, i principali metodi di questa classe:

- *paintBricks(Graphics2D g2D)*: è il metodo che si occupa di disegnare i *bricks* sul pannello di gioco, questo metodo valuta lo stato di ogni mattoncino sulla matrice di *bricks* (come visto i valori di questa matrice determinano o stato di ogni elemento; se è stato colpito o meno e il tipo di mattoncino presente), chiaramente accedendo a questa informazione per mezzo del controller; in seguito viene disegnato il mattoncino in ogni "casella" della matrice in base a questo valore.
- Altri metodi come *paintPaddle(Graphics2D g2D), paintBall(Graphics2D g2D), printScore(Graphics2D g2d)*, etc… hanno scopi analoghi, per tutti gli altri elementi grafici e funzionano sostanzialmente allo stesso modo.

La classe MainPanel contiene una sottoclasse chiamata **secondPanel** che rappresenta un piccolo pannello che si sovrappone a quest'ultimo nei momenti in cui il gioco è fermo, in particolare si distinguono 5 casi:

- *Lose:* che è il caso in cui il giocatore ha perso la partita
- *Pause:* è lo stato di pausa del gioco
- *Start:* è lo stato in cui il gioco non è ancora iniziato, si è in attesa di cominciare
- *Win*: è lo stato in cui si ha appena completato un livello
- *Complete*: è lo stato in cui il gioco è stato completato

In base ai 4 casi per questa sottoclasse sono presenti 4 costruttori differenti, infatti le interfacce di questo pannello sono diverse nei 4 casi.

- **SaveDialog**: questa classe estende il componente JDialog del package javax.swing delle API di java, non è altro che una finestra di dialogo con due pulsanti, si occupa di chiedere al giocatore, qualora previsto, se intende salvare la partita in corso, se il giocatore decidesse di salvare viene richiamato il metodo *handleSaveEvent()* della classe Controller, il quale si occuperà di salvare i dati di gioco. In seguito il giocatore viene riportato alla schermata principale *StartWindow*.

3.3 Problemi riscontrati

L'aspetto che ha richiesto più lavoro è sicuramente quello della modellazione dell'applicazione secondo il Pattern Model-Controller-View, probabilmente il meno adatto per l'implementazione di applicazioni grafiche, vista la forte separazione tra l'aspetto grafico e delle animazioni con lo stato del gioco e i vari metodi che ne controllano la logica.

Un altro aspetto che ha richiesto molto lavoro è stata quella della modellazione del gioco stesso, in particolare nella gestione del movimento della pallina e del paddle, e della rilevazione delle collisioni. Inizialmente avevo pensato di modellare gli elementi del gioco

non come delle classi, ma come delle semplici variabili di interi (come la posizione della pallina, posizione e dimensioni di mattoncini e Paddle, etc..), si è poi creata la necessità di modellare questi ultimi come delle classi vere e proprie, e quindi degli oggetti, di modo da poter richiamare su di essi dei metodi (rilevazione di collisione, movimenti, etc…).

Un problema che ha richiesto molto tempo è stato quello di implementare correttamente il salvataggio del gioco, soprattutto la parte che riguarda lettura/scrittura di File attraverso l'applicazione, in particolare ho dovuto trovare una soluzione che funzionasse correttamente non solo attraverso l'IDE, ma che attraverso il File eseguibile (.jar); la soluzione è stata quella di rendere il file di salvataggio raggiungibile attraverso tutti i mezzi, posizionandolo in una cartella della home directory (C:\Users\utenteAttuale).

4 APPENDICE

Per quanto riguarda il movimento della pallina dovrebbe essere chiaro che se essa colpisce un ostacolo come le pareti o un mattoncino, dopo la collisione rimbalza a specchio, mentre se urta contro il *paddle* rimbalza in base al punto di contatto, per rendere possibile la direzionabilità della palla. Per rendere più chiaro cosa si intende per "**direzionabilità**" della pallina si riporta la parte di codice che gestisce il contatto tra pallina e *Paddle*:

```
if (this.paddle().contact(this.ball())){
      this.increaseSpeed();
      this.playSoundEffect(2);
      double tmp = ball().getCoordXOfBall() - this.paddle().coordXOfPaddle() –
                        (Paddle.WIDHT_PADDLE/2);
      this.ball().setDX(tmp * 0.04);
      this.ball().setDY(-(Math.sqrt(Model.getInstance().getBall().getSpeed() –
                              Math.pow(this.dX(), 2))));
}
```

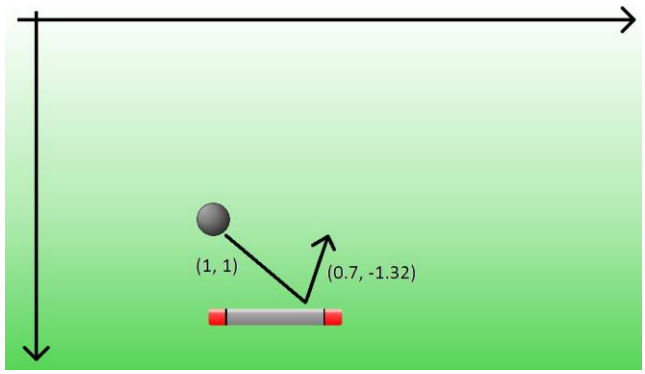
Questa parte di codice fa parte del metodo *checkCollision()*; come già detto si occupa di gestire il contatto tra la pallina e il Paddle, rilevando un eventuale contatto e gestendo la nuova direzione della pallina;

1. In prima analisi viene verificato il "contatto" tra i due elementi, se ciò avviene si entra nel corpo dell'*if*;
2. Viene aumentata la velocità della pallina, infatti ciò avviene ogni volta che la palla colpisce il *Paddle*, questo per rendere più difficoltoso il gioco se si impiega molto tempo a completare il livello (vedi Specifiche dei requisiti).
3. Viene avviato l'effetto sonoro del contatto con il *paddle*.
4. A questo punto viene calcolata la nuova direzione della pallina, nel seguente modo:
    a. Si calcola il punto in cui la pallina colpisce il *Paddle*, più precisamente si calcola la distanza il centro del *Paddle* e il punto di contatto, e si memorizza il dato nella variabile *tmp* (notare che questo valore può essere positivo o negativo).

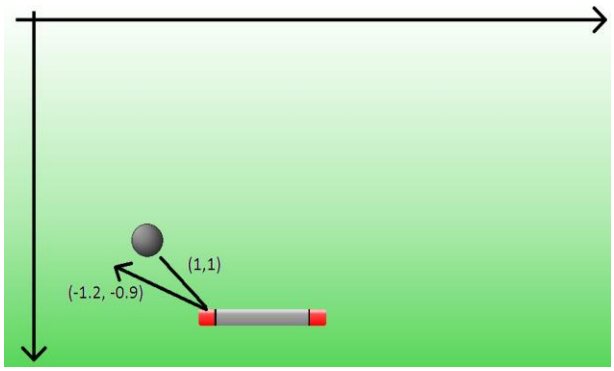
b. Il nuovo valore dx della pallina viene calcolato moltiplicando il valore in *tmp* per un coefficiente pari a 0.04 (scelto dopo una serie di prove sperimentali), mentre il nuovo valore di *dy* viene calcolato come:

$$dy = -\sqrt{v^2 - dx^2}$$

In modo che, in primo luogo lungo x la pallina segua una direzione che sia proporzionale al punto di contatto, mentre lungo y sia proporzionale alla velocità e a *dx*, ma sempre contraria alla direzione di arrivo (grazie al segno meno), in modo da garantire il "rimbalzo".



	PRIMA	*DOPO*
speed	1	1.5
dx	1	0.7
dy	1	-1.32



	PRIMA	*DOPO*
speed	1	1.5
dx	1	-1.2
dy	1	-0.9

Per velocità si intende il numero di pixel percorsi a ogni "clock" del timer (10 ms)

5 BIBLIOGRAFIA

- Wikipedia, "Breakout", https://en.wikipedia.org/wiki/Breakout_(video_game)
- Wikipedia, "Arkanoid", https://en.wikipedia.org/wiki/Arkanoid
- Stackoverflow.com, http://stackoverflow.com/
- Html.it, http://www.html.it/
- API di Java, http://docs.oracle.com/javase/7/docs/api/