Corso di Programmazione ad Oggetti – a.a. 2013/2014
Corso di Laurea in Ingegneria e Scienze Informatiche – Università di Bologna

Relazione relativa all'ingegnerizzazione di TETRIS ATTACK – Tetris Attack

Componenti del gruppo:Perroni Nicolas, Pezzolati Riccardo, Valmori Mauro

1 Analisi del Problema

In *Tetris Attack*, il giocatore è presentato come un terreno di gioco consistente in una matrice virtuale di quadrati, ognuno può essere occupato da un blocco colorato. I blocchi sono impilati uno sopra l'altro e lentamente si alzano verso la cima dell'area di gioco, con nuovi blocchi che vengono aggiunti dal basso. Il giocatore deve ordinare i blocchi orizzontalmente o verticalmente in linee di 3 o più facendo combaciare i loro colori scambiando i blocchi orizzontalmente due(adiacenti) alla volta. Appena vengono create queste combinazioni, i blocchi in questione vengono distrutti e scompaiono dallo schermo e i blocchi sopra di essi cadono per colmare il buco venutosi a creare. Il gioco finisce quando un qualunque blocco raggiunge la cima dell'area di gioco.
E' stato commissionato agli studenti Perroni, Pezzolati e Valmori la rivisitazione del gioco Tetris Attack in linguaggio Java. Le funzionalità del gioco risultano molteplici, legate alla fase di gioco(sezione 1.1)alle scelte del giocatore(sezione 1.2) e agli aspetti aggiuntivi(sezione 1.3)

1.1 Analisi del gioco

Distruggere 3 o più blocchi con un singolo scambio di pezzi crea una "combo". Esistono diversi tipi ci combo:

1. Combo semplice: una combo composta da soli 3 blocchi genera un punteggio di 2 punti;
2. Combo articolate: vedi descrizione classe CheckCombo.

Una combo è una serie di 3 o più blocchi adiacenti. Lo spostamento, da parte del cursore di gioco, dei blocchi genera un thread che controlla la presenza di combo partendo dai blocchi spostati. Le combo possono anche essere generate dalla caduta dei blocchi o dalla salita di nuove righe di blocchi nella matrice di gioco.

In particolare:
Esistono 5 colori diversi di blocchi: verde, giallo, rosso, grigio, azzurro. Giocando a difficoltà HARD si aggiunge un ulteriore colore: il magenta. Il colore dei blocchi è generato casualmente.
Le combinazioni vengono create dal "cursore" scambiando di posto 2 blocchi orizzontalmente adiacenti. E' possibile quindi creare contemporaneamente due combinazioni di colori differenti. Deve essere inoltre possibile spostare un blocco e una posizione vuota.
Vengono permessi 4 livelli diversi di difficoltà: EASY, NORMAL, HARD,VERY HARD. Ognuna delle quali aumenta rispettivamente la velocità con cui compaiono nuovi blocchi.
L'avanzare dal basso verso l'alto dei pezzi deve fermarsi quando vengono create combinazioni e durante l'assestamento dei pezzi per permettere al giocatore di cercare la combinazione successiva prima che nuovi blocchi comincino a comparire, deve perciò, poter postarsi il cursore in qualsiasi momento, in particolare quando:
i blocchi vengono distrutti e mentre cadono.
L'applicazione dovrà perciò garantire un ritardo fra un evento e la sua visualizzazione il più basso possibile per garantire la massima giocabilità.
Detto ciò, è subito evidente che sarà un'applicazione multithread GUI-based finalizzata al massimo verso la reattività di quest'ultima nonostante l'importantissima parte di concorrenza rispetto ai pezzi onde agire correttamente su di loro.
In particolare si dovrà prestare molta attenzione a prestare mutua esclusione fra chiunque potrà agire in fase di scrittura e lettura sui pezzi.

I thread che dovranno agire concorrentemente sono:
Event Dispatch Thread(EDT): in fase di lettura, sarà compito suo aggiornare la view tramite il metodo paintComponent().
Controller: lettura/scrittura, sarà compito suo cercare combinazioni dopo che la matrice dei pezzi subisce una variazione e dire al Faller quali pezzi dovranno cadere. Può essere attivo sulla matrice più di un controller alla volta.
Faller: lettura/scrittura Il suo compito sarà far cadere i pezzi in modo graficamente visibile. La caduta può verificarsi quando una combinazione avete pezzi sopra di essa viene distrutta o quando un blocco viene spostato dal cursore sopra uno spazio vuoto.
Swap: lettura/scrittura. lo scambio di posizione dei blocchi da parte del cursore.
ControlPlay: lettura/scrittura. Il thread che gestisce la parte Model del giocatore. Questo è responsabile dell'avanzamento dei nuovi blocchi.
Essendo che ogni thread agisce in fase di scrittura si è costretti a creare una mutua esclusione fra tutti i thread, ricordando di non abusarne troppo per non causare un ritardo nell'EDT che si rifletterebbe in un ritardo nell'aggiornamento dello stato visivo del gioco.

1.2 Analisi delle scelte

All'avvio dell'applicazione dopo la schermata di caricamento,il giocatore si troverà di fronte al menu grafico dove potrà scegliere tra 4 diverse opzioni:
- *New Game :* Consentirà al giocatore di iniziare una nuova partita usando come stage di gioco e difficoltà predefinita;
- *Game Options :* In questa nuova schermata il giocatore potrà scegliere tra 6 diversi stage di gioco, dopodichè effettuata la scelta e settata la difficoltà desiderata, inizierà la nuova partita senza dover ritornare alla schermata precedente;
- *How To Play :* In questa nuova schermata invece il giocatore potrà apprendere tramite un'animazione come giocare, quali pulsanti cliccare e potrà visualizzare in tempo reale ogni singolo pulsante a quale azione di gioco è legata, il pulsante "Start" consentirà di rivedere le istruzioni, mentre il pulsante "Back" consentirà il ritorno al menu di scelta;

- *Credits :* L'ultima opzione consentirà al giocatore di prendere visione dei crediti di gioco(riconoscimenti)sotto forma del classico scorrere verso l'alto dei crediti in una schermata nera.

1.3 Analisi degli aspetti aggiuntivi

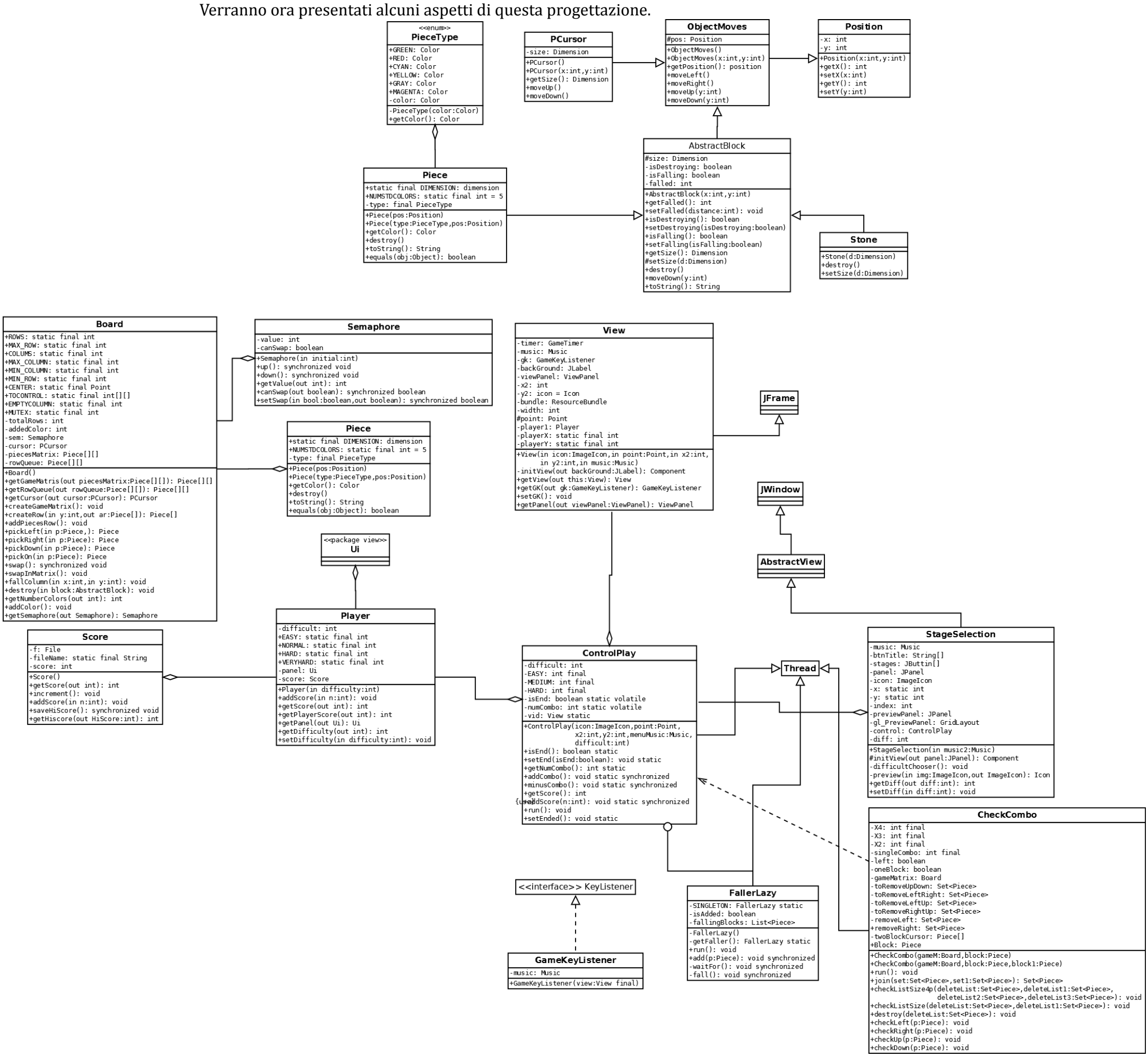
Un'altra funzionalità del gioco è la riproduzione della musica.
Infatti durante il caricamento iniziale ,all'avvio del menu principale e dei menu Credits e HowToPlay sarà riprodotta una musica che suonerà all "infinito" , sarà diversa per il menu di scelta dello stage e ogni stage avrà la sua musica,inoltre per le azione compiute dal giocatore(swap dei pezzi) verrà riprodotto un suono.

1.4 Conclusione

Visti i numerosi requisiti di gioco abbiamo pensato di realizzare un'applicazione GUI-based in modo da suddividere le varie opzioni in più schermate per poter permettere al giocatore una chiara visione e una miglior scelta.

2 Progettazione Architetturale

In questa parte della relazione verrà analizzata la progettazione per la parte dell'applicazione che concerne l'organizzazione e le relazioni tra le entità.
Per la progettazione si è deciso di usare il pattern **MVC** (Model View Controller) mediante il quale si riesce a mantenere un distacco tra la parte visiva , quella di modellazione dei pezzi di gioco e quella legata alla logica del gioco.
Verranno ora presentati alcuni aspetti di questa progettazione.



2.1.1 Presentraione package tetrisattck.model.player e tetrisattack.model.basic

Descrizione degli aspetti principali:

. Position: modella una posizione (x,y). Utile quando hai oggetti che si muovono in uno spazio bidimensionale per rappresentarne il modello.

.ObjectMoves: offre le basi per gli spostamenti ortogonali di un oggetto necessarie per il cursore(Pcursor) e per i blocchi(AbstractBlock).

.PCursor: modello del cursore su cui agirà l'utente per scambiare I blocchi.

.AbstractBlock: classe astratta che fornisce una base per pezzi(Piece) e pietre(Stone). Definisce cos'è in generale un blocco e le sue funzioni. In particolare deve immagazzinare le condizioni di stato del blocco:
se sta cadendo e se sì, di quanto è caduto(per gli spostamenti di riga in riga);
se sta per essere distrutto;

.Stone: non implementata. E' stata scritta nel caso in cui il tempo avesse permesso un'evoluzione del gioco.

.Piece: modello di un pezzo in TetrisAttack. Definisce la sua dimensione e il suo colore. E' importante che ogni pezzo sia facilmente equiparabile per un comodo controllo nel cercare le combinazioni.

.PieceType: enum che definisce le tipologie di colori a disposizione.

.Board: il modello del campo di gioco.

.Player: modello di un giocatore. Comprende il suo Score e il campo di gioco(board).

.Ui: la classe responsabile della visualizzazione del campo di gioco. Si tratta di un Jpanel utilizzato come canvas per il disegno dei pezzi. Comprende i metodi per agire sulla loro posizione nel tempo da parte del controller.

.Faller: Inner Class di ControlPlay. Il suo compito è gestire tutti i pezzi che stanno cadendo.

.GameKeyListener: KeyListener atti a ricevere gli Input da tastiera dell'utente.

2.2.1 Presentazione package ***tetrisattack.utility***

In questo package sono implementate le classi che modellano il cronometro di gioco
le musiche e la creazione degli ImageButton estensione della classe java.swing.JButton utilizzata nel GameMenu e nel EndGame.

Aspetti principali:

. GameTimer: Questa classe modella il cronometro di gioco,all'avvio di ogni partita calcolerà lo scorrere del tempo ricavando dal campo *diffTime(tempo corrente del sistema -startTime)*usando il metodo *synchronized inc()* per poi ricavare la stringa corrispondente che verrà visualizzata nella View.

. **ImageButton:**Questa classe modella un JButton usando un'immagine come icona,in questa classe verranno implementate anche alcune caratteristiche del JButton in modo da dover dichiarare solo il riferimento alla classe senza dover dichiarare tutti i metodi del JButton,inoltre verrà implementato il metodo *setBtnY()* a cui verrà passato un intero per settare la nuova posizione del JButton.

. Music: Questa classe ha il compito di gestire la musica di gioco sotto forma di Thread.

. Tutorial: Questa classe ha il compito di gestire il thread dell'animazione del tutorial

2.2.2 Presentazione package ***tetrisattack.view.menu***

In questo package sono implementate le classi legate alla GUI relativi ai menu di scelta.
Aspetti principali:

. AbstractView:E' la classe che fornisce lo scheletro per le classi GameMenu,StageSelection e HowToPlay implementata mediante il pattern Template Method le classi che estenderanno AbstarctView la useranno come base ovvero come container per i rispettivi Jpanel(mediante il metodo protetto *initView()*).

. SplashScreen: Questa classe modella uno SplashScreen di caricamento con un immagine di background,al completamento della JProgressBar(mediante il metodo *refresh()*)partirà la schermata GameMenu.

. GameMenu: Questa classe ha il compito di modellare il menu principale dell'applicazione nella quale l'utente potrà scegliere tra 4 diverse opzioni,*New Game* che farà partire una nuova partita con lo stage predefinito,*StageSelection* dove l'utente potrà scegliere uno dei 6 stage,*HowToPlay* dove potrà apprendere come giocare e infine *Credits* dove potrà visualizzare i crediti di gioco.

. StageSelection: Questa classe modella il subMenu di scelta qui il giocatore potrà scegliere uno dei 6 stage di gioco e iniziare la nuova partita.

. HowToPlay: Questa classe modella il menu dove il giocatore potrà apprendere mediante un animazione quali tasti premere per giocare.

. Credits: Questa classe modella la visualizzazione dei crediti di gioco,mediante l'utilizzo di un Thread.

2.3.1Presentazione del package tetrisattack.control

Questo package contiene le classi responsabili del controllo dell'applicazione come:

.CheckCombo:un oggetto un oggetto "CheckCombo" è un thread che controlla la presenza di combo nella matrice di gioco, assegna I punti di eventuali combo presenti, e elimina la/le combo trovate.

.ControlPlay: un oggetto "ControlPlay" è un thread che fornisce metodi di concorrenza alla classe CheckCombo, controlla che il muro dei blocchi non sia arrivato al massimo della sua altezza e, se ci è arrivato, termina il gioco corrente.

.Faller (inner class di ControlPlay): un oggetto "Faller" è un thread che fa cadere I pezzi contenuti nella lista che ha come campo privato

. GameKeyListener: questa classe gestisce l'intero input da tastiera durante il gioco.

3. Organizzazione in Package

In questa sezione, verrà effettuata una analisi dell'organizzazione in package dell'applicazione:
in particolare verrà spiegato quali parti dell'applicativo ogni package modellerà e verrà brevemente descritto il funzionamento delle classi al loro interno, mentre nella sezione 5 verranno descritti in modo più dettagliato.

.tetrisattack.model.basic: Contiene il sorgente di tutti gli oggetti "primitivi": pezzi(Piece), pietre(Stone, non utilizzata!), cursore(PCursor), le relative sottoclassi ed i sorgenti a loro necessari. Le classi qui contenute (brevemente descritte nel capitolo 2) sono:

- *Position;*
- *ObjectMoves;*
- *PCursor;*
- *AbstractBlock;*
- *Stone;*
- *Piece;*

- *PieceType;*

.tetrisattack.model.player: contiente il sorgente Player e i suoi componenti: Board, Score. Le classi qui contenute sono:
- *Player:* Un oggetto "Player" è un oggetto che contiene la sua tavola dei pezzi(Ui) e una difficoltà.
- *Board:* Contiene la struttura dati principale, una matrice 12x6 rappresentatnte il modello del pannello, ove vi vengono gestiti i pezzi. La classe Board contiene una Inner class Semaphore per gestire la concorrenza.

-*Ui:* (vedi punto 2)
-*Faller:* Inner di controlPlay
-*Gamekey:* contiene i Listeners

. tetrisattack.main:E' il package principale contenente il main:
-*Main:*Contiene il solo metodo main che avvia l'applicazione.

. tetrisattack.utilities:E' il package che gestisce gli aspetti di utility per le view:

-*GameTimer:*Ha il compito di modellare lo scorrere del tempo di gioco,il metodo *inc()* incrementa il tempo,il metodo *getS()* recupera la stringa contenente la stringa del tempo di gioco.
-*ImageButton:*Ha il compito di modellare i pulsanti di scelta del GameMenu,il metodo *setY()* modifica la posizione del pulsante rispetto all'asse y.
-*Music:*Ha il compito di gestire il Thread della musica.
-*Tutorial:*Ha il compito di gestire il Thread che anima il tutorial per fare in modo che il giocatore possa apprendere come giocare,mediante il metodo *setOne()* vedrà quali pulsanti della tastiera deve premere per giocare mentre mediante il metodo *setTwo()* vedrà come scambiare di posizione due pezzi.

. tetrisattack.view.menu: Questo package contiene le classi che modellano la view dei menu:

-*AbstractView:*classe astratta che fornisce lo scheletro alle classi *GameMenu,StageSelection,HowToplay* mediante il pattern Template Method.
-*Credits:*classe che gestisce la visualizzazione dei crediti di gioco tramite Thread.
-*GameMenu:*classe che estende la classe *AbstractView* e implementa il metodo astratto *initView()*,ha il compito di gestire il menu principale.
-*HowToPlay:*classe che gestisce il tutorial e permette all'utente di capire come giocare,estende la classe *Abstractiew*,il pulsante start farà partire il tutorial mentre il pulsante back permetterà all'utente di tornare nel menu precedente.
-*SplashScreen:*classe che modella lo splashscreen iniziale,il metodo *refresh()* modificherà la barra di caricamento,una volta completata avvierà il *GameMenu*.
-*StageSelection:*classe che estende la classe *AbstractView*,implementa il menu di scelta mediante 6 pulsanti che fungono da anteprima dello stage, ogni **pulsante** inizializzerà una nuova partita,inoltre tramite un pop-up l'utente potrà selezionare la difficoltà di gioco.

. tetrisattack.view.game: Questo package contiene le classi che modellano la view di gioco:

-*View:*E' la view di gioco ,contiene il background scelto precedentemente nel menu *StageSelection* o mediante il pulsante *NewGame* nel menu principale per la visualizzazione delle info di gioco verrà usato un JPanel modellato dalla classe *ViewPanel*.
-*EndGame:*E' la view di fine gioco,qui l'utente potrà scegliere se fare una nuova partita(verrà riportato nel menu selezione dello stage)oppure uscire.

.tetrisattack.control: Questo package contiene le classi responsabili del controllo dell'applicazione come:

-*CheckCombo:un* oggetto un oggetto "CheckCombo" è un thread che controlla la presenza di combo nella matrice di gioco, assegna I punti di eventuali combo presenti, e elimina la/le combo trovate.
-*ControlPlay*: un oggetto "ControlPlay" è un thread che fornisce metodi di concorrenza alla classe CheckCombo, controlla che il muro dei blocchi non sia arrivato al massimo della sua altezza e, se ci è arrivato, termina il gioco corrente.
-*Faller (inner class di ControlPlay):* un oggetto "Faller" è un thread che fa cadere I pezzi contenuti nella lista che ha come campo privato
-*GameKeyListener:* questa classe gestisce l'intero input da tastiera durante il gioco.

4. Suddivisione del lavoro

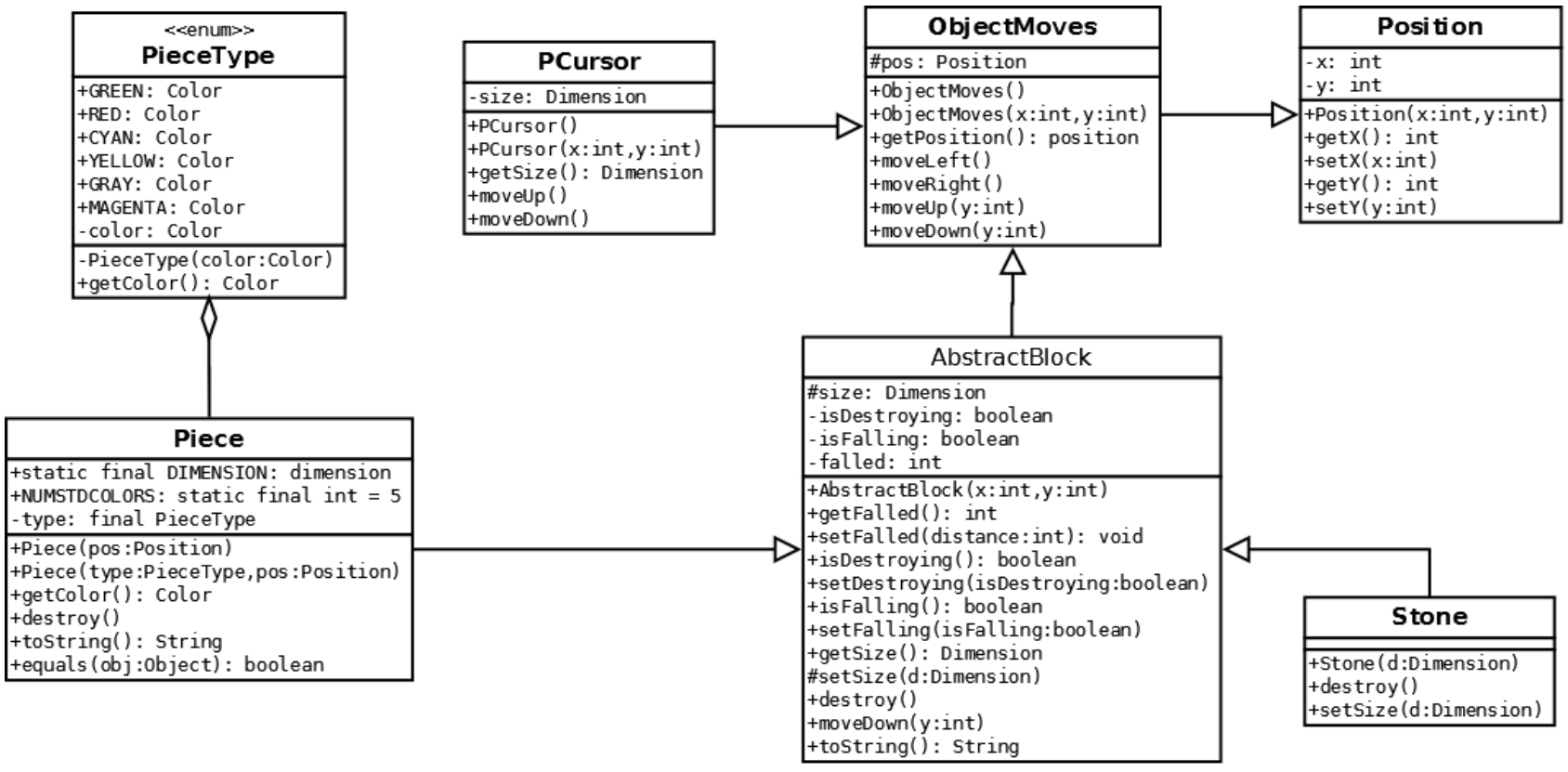
Nel contesto della realizzazione del progetto il lavoro tra i vari componenti del gruppo è stato suddiviso come segue:
- Riccardo Pezzolati si è occupato del package *"view"* e *"utilities".*
- Nicolas Perrori si è occupato del package *"model"*(sotto-package *"basic","player"*) inerenti alla creazione dei pezzi di gioco e relativo pannello di gioco e la classe Ui del package "view" e la gestione del cursore.
- Mauro Valmori si è occupato del package *"controller"* (ricerca delle combo) e alla parte *"model"* relativa al calcolo del punteggio.

5.1 Progettazione di dettaglio: Parte di Nicolas Perroni

In questa sezione verrà discussa in dettaglio la parte del progetto di specifica competenza di Nicolas Perroni.
La presente sezione è divisa in 2 sotto-sezioni, ognuna spiega in dettaglio il funzionamento di uno dei 2 packages di cui il membro del gruppo è responsabile:tetrisattack.model.basic e tetrisattack.model.player.
Per ognuno di questi package verrà presentato uno schema UML dettagliato e una descrizione esauriente di ogni classe del package.

5.1.1. Analisi package tetrisattack.model.basic

In questa sezione si discuterà del package della model sulla creazione dei pezzi
Questo è lo schema UML dettagliato del package:



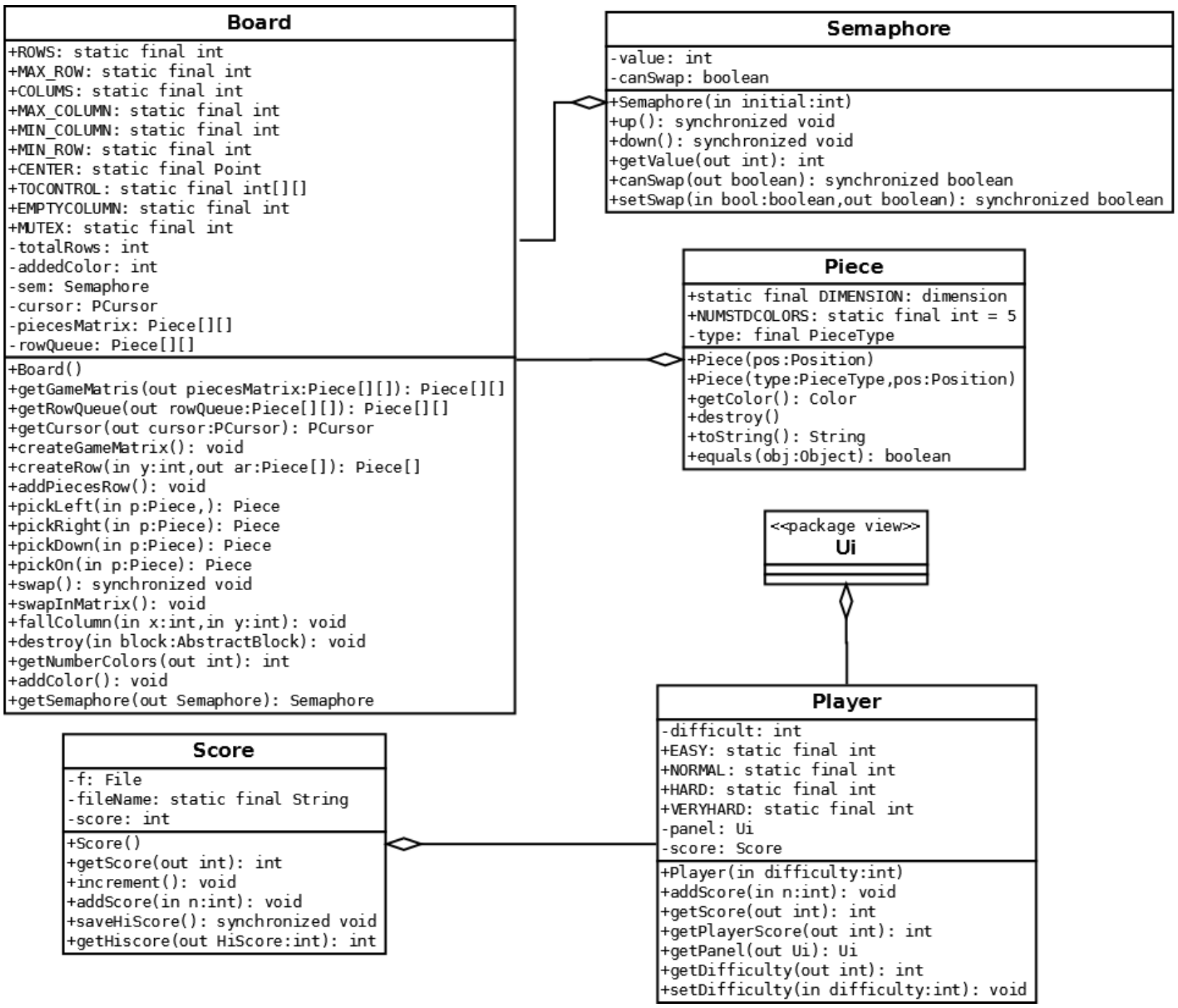
1.Position: Models an (x,y) position. Comprende I campi int x e int y ed i relativi getter/setter. Non sono state scelte strategie particolari per questa classe.
2.ObjectMoves: Contiene un istanza Position. Definisce tutti i metodi necessari per un modello di un oggetto che può essere spostate in un ambiente bidimensionale. E' da notare che gli spostamenti laterali sono sempre eseguiti con un passo lungo quanto lo spessore di un pezzo(Piece.DIMENSION.getWidth(){class Piece}), mentre gli spostamenti verticali sono da impostare secondo una lunghezza fornita come parametro in precisione Integer.
3.PCursor: Estende ObjectMoves. E' il modello del cursore gestito dal giocatore. Contiene due metodi per lo spotamento verticale con passo lungo quanto l'altezza di un pezzo(Piece.DIMENSION.getHeight()).
4.AbstractBlock: Estende ObjectMoves. E' la classe astratta utilizzata come base per le classi Piece e Stone. In particolare era stato scelto il pattern Template Method, poiché le classi differiscono semplicemente per metodo "destroy(), ovvero per il modo in cui dovevano essere distrutti poiché sulle pietre si può riflettere in un cambiamento della dimensione che invece per i pezzi è costante.
Il metodo setSize() è reso protetto poiché le istanze della classe Piece devono essere di dimensione fissa, mentre invece le istanze Stone possono variarla.
5.Stone: Classe non usata. Certe scelte sono state eseguite pensando che, se la situazione lo avesse permesso, si sarebbero potute implementare funzioni aggiuntive.
6.Piece: La classe Piece e la sua visualizzazione è stata forse il punto in cui si è focalizzato di più il lavoro di studio di fattibilità. Inizialmente si era pensato di trattare un pezzo come un estensione di JLabel che insieme ai LayoutManager(GridBagLayout su tutti) agevolavano molte operazioni come il rintracciamento di un blocco data la sua posizione(getComponent() , il suo spostamento tramite le GridBagConstraints fornite dal relativo LayoutManager, la sua visualizzazione. La decisione di abbondare questa comoda strada è dovuta all'animazione di caduta dei pezzi: Il GridBagLayout non permette un piazzamento assoluto dei componenti. Si è provato anche a creare un layout su misura che si sarebbe rivelata la soluzione più elegante e "attraente", ma è una cosa molto complicata che si sarebbe potuta rivelare solo una grande perdita di tempo e perciò abbandonata poco dopo. La soluzione adottata, anche perché permette una miglior separazione tra MVC, è quella di utilizzare un canvas per la visualizzazione dei pezzi e di trattare i pezzi da zero senza estendere JLabel perché non ne avremmo sfruttato bene le proprietà.
Un pezzo ha una dimensione costante e non può essere cambiata. Per la generazione casuale del colore si è fatto uso del metodo Math.random(). Sono presentati due costruttori: uno permette l'impostazione del colore del blocco, l'altro lo genera casualmente fra i 5 colori standard (non è quindi generato il colore Magenta).
7.PieceType: enum dei possibili colori dei pezzi.

1 - GREEN(Color.GREEN),
2 - RED(Color.RED),
3 - CYAN(Color.CYAN),
4 - YELLOW(Color.YELLOW),
5 - GRAY(Color.GRAY),
6 - MAGENTA(Color.MAGENTA);

La tipologia 6(Magenta) è resa disponibile solo quando la difficoltà è settata su difficile (HARD), come ulteriore difficoltà per il giocatore.

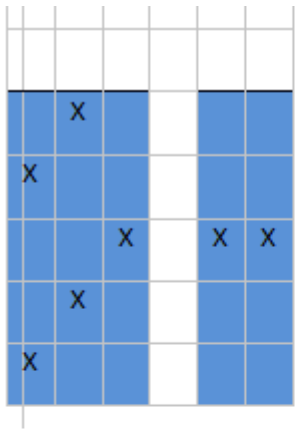
5.1.2. Analisi package tetrisattack.model.player

In questa sezione si discuterà del package della creazione della matrice di pezzi
Questo è lo schema UML dettagliato del package:



1.Board: Board è il modello della tavola di gioco. Utilizza una matrice di pezzi 12x6 come struttura dati per creare il modello della tavola di gioco. Scopo importante nello gestire la matrice sta nella sua Inner Semaphore. Infatti la race condition generata dai numerosi threads che agiscono sui pezzi, rendono necessaria una struttura di controllo. Come descritto nella parte iniziale di questo testo, i thread sono tutti mutualmente esclusivi, quindi la struttura adottata risulta un Semaforo MUTEX gestito appunto dalla Inner. I metodi che agiscono sullo stato del semaforo sono: Up() (signal) e Down(wait). Down() mette in attesa di un invocazione del metodo Up() il thread invocante, fornendo così un accesso controllato. Un particolare accorgimento è stato fatto per il metodo swap() nella classe Board che necessita di un ulteriore controllo per evitare che compia due scambi consecutivi senza che il Thread CheckCombo riesca a controllare il risultato del primo swap. Questo creerebbe inconsistenza nello stato della matrice che perderebbe il riferimento ai blocchi da distruggere.
Metodi principali:
- swap(): scambia di posto due pezzi. Sono necessari molti controlli per evitare errori: infatti i pezzi che stanno cadendo o che stanno per distruggersi non devono essere scambiati. Inoltre se un pezzo viene spostato sopra una posizione vuota, questo dovrà cadere(ad opera del FallerLazy). Ad ogni chiamata di swap che abbia successo(cioè non fermata dalla variabile di controllo che impedisce il "doppio-swapping") viene invocato il thread CheckCombo per controllare la presenza di combinazioni.
- addPiecesRow() : Questo metodo viene invocato quando si vuole aggiungere una riga nuova nella matrice di gioco. Questo metodo sposta tutti le righe verso le posizione più basse della matrice di un posto pe creare spazio alla nuova riga. La prima riga della coda rowQueue(uma matrice 2x6 che serve nella creazione di nuove righe e per non lasciare spazio vuoto nella parte inferiore durante l'avanzamento del muro, infatti viene disegnata subito sotto dopo la matrice principale).
- CreateGameMatrix(): Inizializza la matrice di partenza con blocchi di colore casuale. Onde evitare spiacevoli combinazioni già create dal caso, vengono effettuati controlli in punti strategici. La figura seguente mostra la matrice virtuale in partenza con le caselle occupate dai pezzi colorate in blu. Mentre le caselle marcate in figura con una "X" sono i pezzi da controllare per evitare qualsiasi combinazione possibile. Ricordando che le combinazioni minime sono da 3 blocchi, quelle risultano essere le posizioni più strategiche, ovvero l'insieme con il minor numero di pezzi da controllare. L'algoritmo sostituisce nelle posizioni segnalate i pezzi fino a quando non viene eliminata ogni corrispondenza con i blocchi vicini. Non è molto funzionale, è molto migliorabile.
Score:

3.Player: Contiene le costanti(pubbliche) per indicare il livello di difficoltà che poi riceverà come parametro obbligatorio del proprio costruttore. Quando la difficoltà è impostata su HARD viene comunicato all'istanza di Board che è necessario aggiungere il colore Magenta attraverso il metodo addColor().
4.GameKeyListener: implementa keyListerner. Le scelte dei tasti sono state fatte in base a quella che è stata ritenuta la comodità



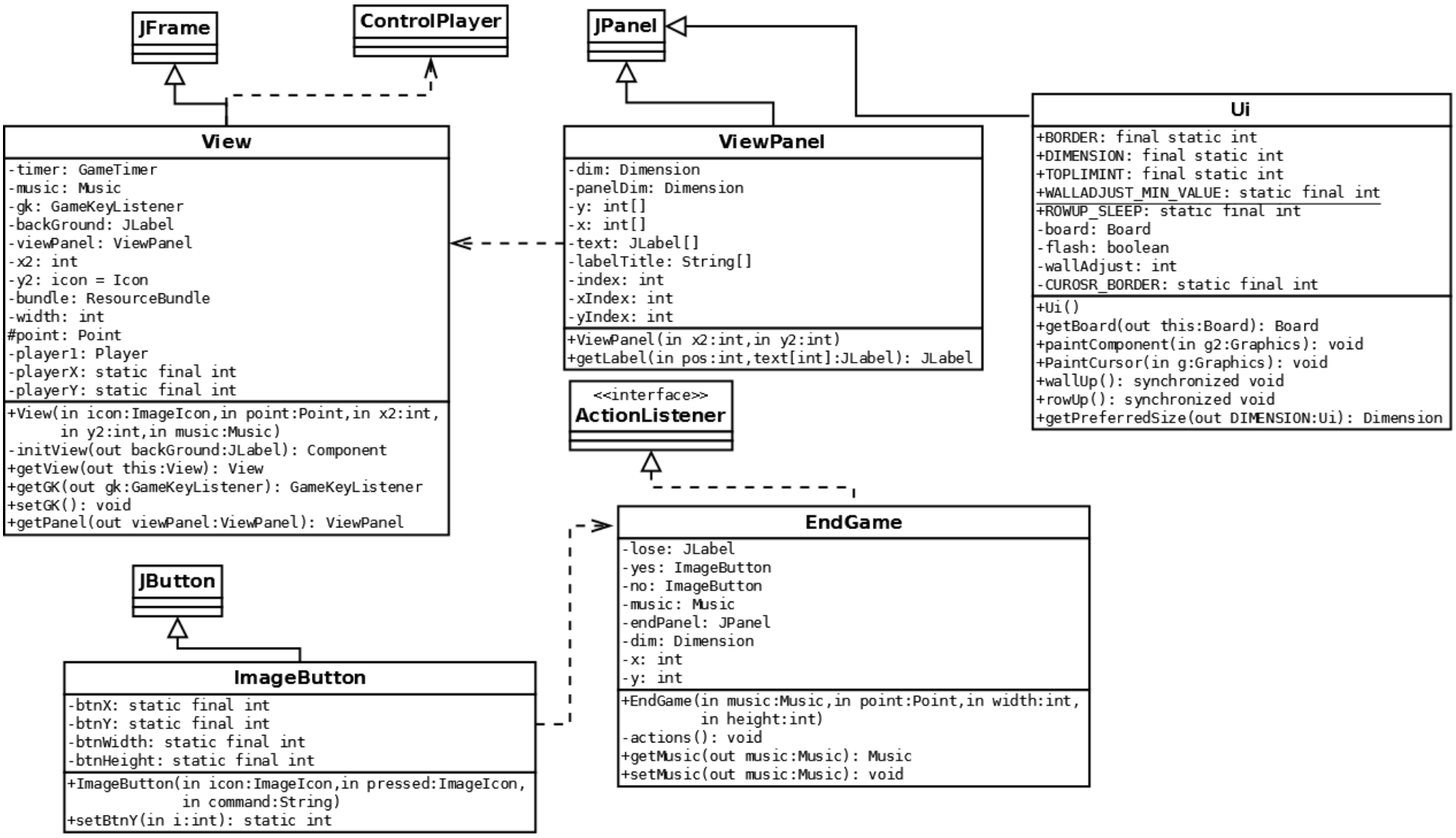
massima per l'utente. In Java i Listener in questione sono gestiti dall'EventDispatchThread(EDT) e alcuni dei metodi che vengono chiamati alla pressione di tasti quali "spazio"(per il metodo swap(){ classe Board} responsabile dello scambio di pezzi) e "c"(per il metodo rowUp(){classe Ui} per alzare il muro fino alla comparsa della riga successiva di pezzi) sono soggetti a criticità: lo scambio dei pezzi è controllato da un semaforo e l'alzarsi della riga è soggetto a Thread.Sleep() per crearne l'animazione facendo salire i blocchi un po' alla volta. Queste cose potrebbero rallentare l'azione dell'EDT che potrebbe causare un ritardo dell'aggiornamento dell'interfaccia grafica con la conseguente perdita di reattività. Per questo motivo vengano creati thread anonimi incaricati di occuparsi di questi compiti.
In particolare si è scelto di attivare l'azione di rowUp() nel momento del rilascio(e non della pressione) del tasto "c" per evitare che una pressione prolungata accidentale del tasto sviluppi più avanzamenti dei pezzi di quelli voluti.
5.Faller: Inner class dell'outer ControlPlay. La scelta di renderlo Inner Class è dovuta alla possibilità di vedere ed agire sui campi del ControlPlay. Faller è stato pensato come l'unico thread gestore della caduta dei pezzi a cui più thread faranno riferimento. Questo lo sposa perfettamente con il Pattern SINGLETON, ed infatti così è stato realizzato. Inizializzato in modo "lazy", non viene istanziato fino al momento del bisogno. Il Faller agisce(richiamando il metodo "fall(Piece, (int) distance)" nella Classe Board) su tutti i pezzi presenti nel suo buffer fino a quando la loro caduta non è terminata, rimuovendoli poi dalla lista ed invocando un'istanza anonima di CheckCombo su di loro per controllare la presenza di combinazioni. Quando non sono presenti pezzi, il Faller entra in stato di wait() per evitare di occupare la CPU inutilmente controllando la presenza o meno di elementi nel Buffer. Il thread viene poi svegliato quando un pezzo viene aggiunto alla lista tramite il metodo pubblico add(Piece) da un altro Thread. Questo è il classico problema di concorrenza dell' Unbounded Buffer da gestire attraverso una corretta mutua esclusione fra scrittori e lettori. Un'ulteriore difficoltà di concorrenza è che, mentre i pezzi cadono, si deve bloccare l'avanzamento del muro,effettuata dal ControlPlay, agendo sui suoi metodi addCombo(), minuscombo() che sviluppano una sorta di semaforo contatore sull'avanzamento del muro.
6.Ui: trattato come un canvas. Vengono estrapolate le dimensioni di ogni blocco, il suo colore, la sua posizione per poi disegnarlo attraverso il motodo fillRect(). I pezzi vengono disegnati iniziando dalla parte bassa del pannello per arrivare poi fino alla sommità. Questo perché è più facile che ci sia pezzi alla base rispetto che in cima. Una particolare attenzione va prestata alla variabile wallAdjust. Essa, che altro non è che un int, è usata dal metodo paintComponent() nella classe Ui come modificatore della posizione verticale dei pezzi nel tempo secondo la formula: posizione_dove_disegnare_il_pezzo = posizione_pezzo – wallAdjust
più la posizione dove disengare è bassa, più il pezzo si troverà in un posizione più alta sul pannello. L'incremento della variabile wallAdjust è fornito dal ControlPlay eseguendo il metodo wallUp().
In questo modo il pezzo ad ogni chiamata viene ridisegnato più in alto. Questo procedimento viene applicato fino a quando lo spostamento offerto da wallAdjust è tale da permettere un inserimento di una nuova riga in gioco. Da qui nasce la necessità di avere dei pezzi "accodati" alla matrice per disengare l'avanzamento della riga successiva. Quando quindi lo spazio è sufficiente per far comparire la nuova riga wallAdjust viene azzerato e viene aggiunta duna riga con la chiamata al metodo "addPiecesRow()".

5.2 Progettazione di dettaglio: Parte di Riccardo Pezzolati

In questa sezione verrà discussa in dettaglio la parte del progetto di specifica competenza di Riccardo Pezzolati.
La presente sezione è divisa in 3 sotto-sezioni, ognuna spiega in dettaglio il funzionamento di uno dei 3 packages di cui il membro del gruppo è responsabile:tetrisattack.utility e tetrisattack.view..game,tetrisattack.view.menu.
Per ognuno di questi package verrà presentato uno schema UML dettagliato e una descrizione esauriente di ogni classe del package.

5.2.1. Analisi package tetrisattack.view.game

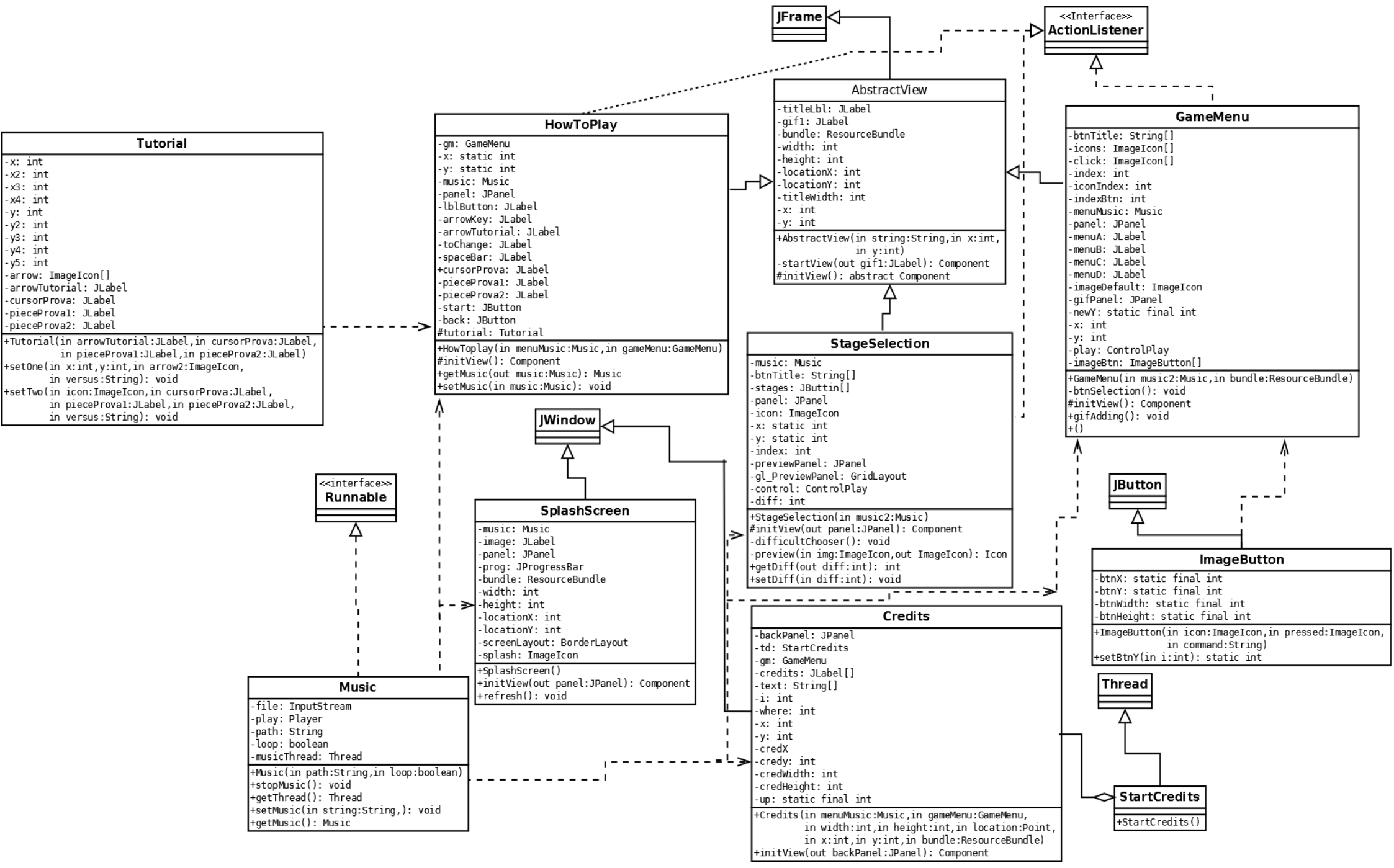
In questa sezione si discuterà del package della view di gioco
Questo è lo schema UML dettagliato del package:



Il package in questione è relativo alla vista del gioco: compito principale delle classi al
suo interno è dunque quello di rappresentare graficamente il cuore del gioco.
Per fare ciò si è creato il frame principale del gioco (View),che conterrà il background scelto dall'utente che fa da container per il pannello viewPanel,istanza della classe ViewPanel, dentro il quale è contenuta l'interfaccia delle informazioni relative al gioco,tempo,miglior punteggio e punteggio corrente,queste informazioni sono modellate utilizzando delle JLabel, all'interno della view di gioco verrà aggiunto un oggetto della classe Player che modellerà il pannello di gioco con i pezzi e il cursore, la classe GameKey del package tetrisattack.controller intercetterà le azioni del giocatore per muovere il cursore e swappare i pezzi selezionati.
Qualora il giocatore perdesse la partita verrà caricato il frame di fine partita,con il quale l'utente potrà scegliere se iniziare una nuova partita oppure uscire dal gioco.

Segue la descrizione dettagliata di ogni classe del package:
1. View: Classe che modella la vista della JLabel di sfondo(scelto precedentemente dall'utente)
e della matrice dei pezzi creata dal model,le informazioni quali lo scorrere del tempo,il punteggio massimo,il punteggio corrente e la difficoltà verranno creati nella classe ViewPanel e aggiunto alla vista,la seguente classe verrà inizializzata dalla classe ControlPlay del package tetrisattack.control che a sua volta verrà inizzializata o dal GameMenu qualora si decidesse di iniziare con le impostazioni di default o dallo StageSelection.
2. ViewPanel: Classe che modella il panel contente le informazioni di gioco,il tempo verrà calcolato dalla classe GameTimer,il punteggio verrà calcolato dalla classe Score che modificherà il testo della label corrispondente,per la posizione delle Label è stato utilizzato il metodo setBounds() in modo che posizionasse le scritte “modificabili” esattamente sotto le scritte presenti nello stage.
3. EndGame: Classe che modella il pannello di fine gioco,contente lo sfondo di Game Over e due ImageButton con i quali il giocatore potrà scegliere se iniziare una nuova partita o uscire dall'applicazione.

5.2.2. Analisi package tetrisattack.view.menu
In questa sezione si discuterà del package “menu” della view.
Questo è lo schema UML dettagliato del package:

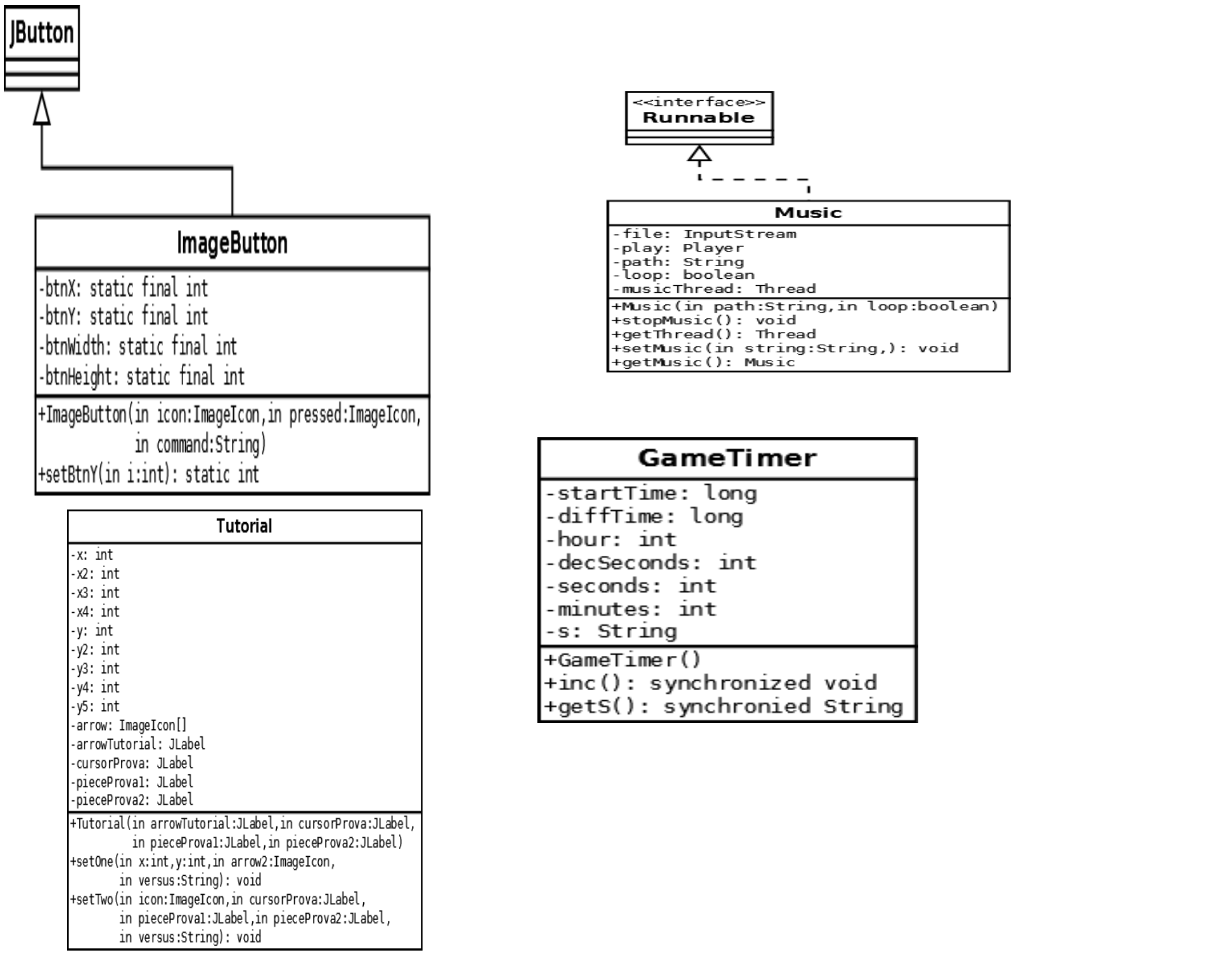


Il package in questione è relativo alla vista dei vari menu di gioco: compito principale delle classi al suo interno è dunque quello di rappresentare graficamente questi menu. Per fare ciò si è creato la Jwindow AbstarctView contente una GIF di sfondo e una GIF rappresentate il titolo,questa classe mediante il template method fornirà la base per le classi GameMenu,StageSelection e HowToPlay che implementeranno il metodo initView()in base alle rispettive caratteristice.
Il menu GameMenu conterrà 4 ImageButton rappresentanti le 4 possibili scelte dell'utente,per poter scegliere lo stage di gioco(e di conseguenza la musica)è stato creato il menu StageSelection, una volta selezionato lo stage sarà possibile selezionare la difficoltà di gioco,nel menu HowToPlay l'utente potrà imparare tramite tutorial animato come giocare,per la visualizzazione dei crediti di gioco dovrà selezionare il menu Credits che si comporrà di un Jpanel con sfondo nero dove scorreranno verso l'alto i vari crediti di gioco.
In fine all'avvio dell'applicazione si avvierà uno splashscreen contente un immagine e una JprogressBar alla fine del caricamento di quest'ultima si avvierà il GameMenu.
Le seguenti classi utilizzano un file denominato View_it.properties per settare le caratteristiche comuni tra le quali: la dimensione,la posizione della finestra rispetto allo schermo,il valore della X,Y usate come coordinate,quindi in ogni costruttore viene passata la variabile di riferiemento per non dover richiamare ogni volta il file ResourceBundle.
Segue la descrizione dettagliata di ogni classe del package:
1.SplashScreen: Classe che modella lo splashscreen iniziale,è composta da un Jpanel contenente un immagine di sfondo alla cui estremità inferiore è stata aggiunta una JProgressBar mediante il BorderLayout il cui completamento verrà gestito dal metodo refresh()(ogni 115 ms modifica il valore % della JPogressBar);
2.AbstractView: Classe astratta che modella la base per i menu GameMenu,StageSelection e HowToPlay,la GIF di sfondo servirà da container per i pannelli dei menu ,i quali implementeranno il template method a seconda del loro compito.
3.GameMenu: Classe che modella il menu principale,è composto da due Jpanel uno per i pulsanti e l'altro per le GIF che completano la grafica del menu,i 4 pulsanti sono istanze della classe ImageButton del package tetrisattack.utility,alla pressione del pulsante l'icona si modificherà per rappresentare la pressione su quest'ultimo, la gestione delle icone verranno implementate nel metodo btnSelection(),i listeners dei pulsanti forniranno le seguenti azioni,NewGame avvierà la partita con stage e difficoltà di default,StageSelection avvierà il menu di scelta dello stage,HowToPlay avvierà il menu tutorial mentre Credits avvierà la visualizzazione dei crediti di gioco,per questi due ultimi alla chiamata di ciascun costruttore verrà passato un riferimento alla classe GameMenu in modo tale da poterla richiamare senza l'avvio di una nuova istanza.
4.StageSelection: Classe che modella il menu di scelta dello stage,tramite il metodo initView() nel panel principale sono stati aggiunti 6 pulsanti che rappresentano altrettanti stage selezionabili ,le icone verranno modificate mediante il metodo preview()che prenderà in ingresso l'immagine dello stage e restituirà la sua versione ridimensionata per il pulsante per fornire un'anteprima dello stage,alla pressione di ogni pulsante apparirà un JOptionPane contente la richiesta di settare la difficoltà,una volta scelta verrà caricata la View(inizializzata da ControlPlay) e partirà il gioco gestito dal controller.
5.HowToPlay: Classe che modella la visualizzazione del tutorial,è composto da un panel principale in cui sono presenti una rappresentazione del pulsanti di gioco(frecce della tastiera, barra spaziatrice e tasto “c”) due pezzi del gioco e il cursore,inoltre sono presenti due pulsanti,con uno si potrà tornare nel menu principale mentre con l'altro si avvierà l'animazione del tutorial(gestita dalla classe Tutorial del package tetrisattack.utility).

6.Credits: Classe che modella la visualizzazione dei crediti di gioco, la classe si compone di un panello con sfondo nero dentro al quale, tramite un'istanza della classe privata StartCredits(estensione di Thread)scorreranno dal basso verso l'alto le scritte contententi gli autori del gioco e i siti dove sono state reperite le risorse,al termine del thread si ritornerà nel menu principale.

5.2.3. Analisi package tetrisattack.utility

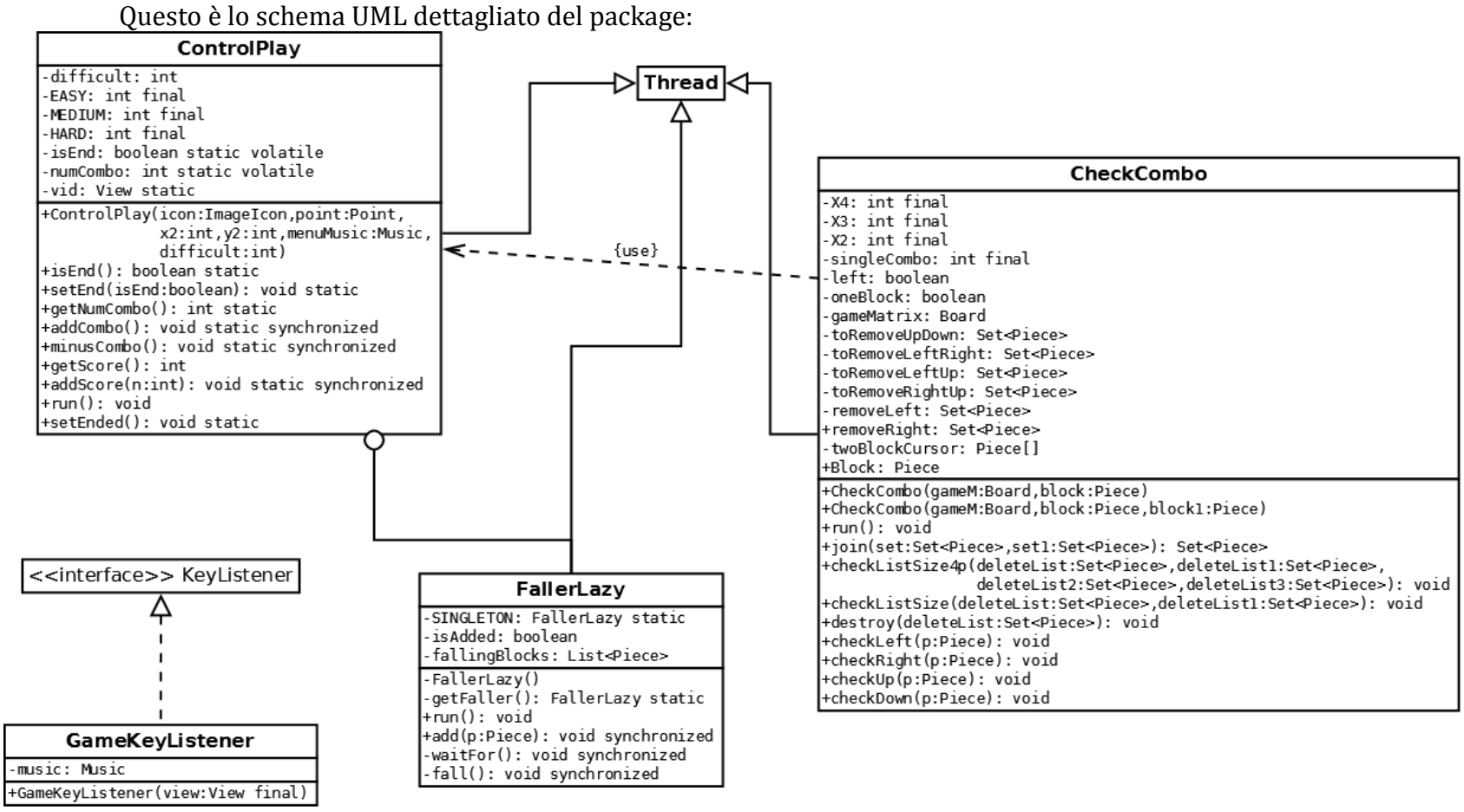
In questa sezione si discuterà del package “utility” della view.
Questo è lo schema UML dettagliato del package:



Il package in questione è relativo alle classi di utilità della view,come il calcolo dello scorrere del tempo,il thread per il tutorial,i pulsanti per il menu e la musica di gioco.
Segue la descrizione dettagliata di ogni classe del package:
1.GameTimer: La classe ha il compito di calcolare il tempo di gioco,nel metodo inc() viene calcolato il tempo suddiviso in ore,minuti,secondi e decimi di secondo e poi inserito in una stringa il metodo getter permetterà alla label della view di venire modificata in tempo reale.
2.ImageButton: Classe che modella i pulsanti del GameMenu e di EndGame,in ingresso nel costruttore viene data un immagine e la sua versione pressed,all'interno sono definiti i metodi per i pulsanti come setBounds() che indica la loro dimensione e posizione,l'eliminazione del bordo e l'area di pressione,l'altro costruttore prenderà in ingresso anche una String contenente l'ActionCommand per i pulsanti dell'EndGame menu,all'interno della classe il metodo setBtnY()servirà per modificare la posizione verticale dei button.
3.Music: Classe che ha il compito di gestire la musica di gioco,ogni menu avrà la sua musica,cosi come ogni stage e anche le azioni di gioco produrranno un suono specifico(movimento del cursore, swap di due pezzi,pezzi in caduta e combo di pezzi)il costruttore accetta in ingresso il percorso della canzone e un valore booleano,se settato a true una volta conclusa la canzone questa ripartirà da capo,mentre se false una volta conclusa si interromperà il thread,è presente oltre ai metodi setter e getter un metodo per interrompere il thread chiamato stopMusic().
4.Tutorial: Classe che ha il compito di gestire il thread del tutorial,nel costruttore verranno utilizzati come parametri di ingresso 4 Jlabel rappresentanti uno una freccia che indicherà quale tasto premere per modificare la posizone del cursore,le altre tre rappresentano il cursore e due pezzi del gioco,ogni 1,5 secondi verrà chiamato il metodo setOne che prenderà in ingresso la nuova icona della freccia ,la sua nuova posizione e la direzione del cursore, in base a queste informazioni indicherà ogni volta quale pulsante si dovrà premere e si potrà vedere l'azione corrispondente,il metodo setTwo simulerà lo scambio di posizione dei pezzi,infine se premuto il pulsante start si potrà ripetere il tutorial oppure premere il tasto back per ritornare nel menu principale.

5.3 Progettazione di dettaglio: Parte di Mauro Valmori

In questa sezione verrà discussa in dettaglio la parte del progetto di specifica competenza di Mauro Valmori.
La presente sezione è divisa in una sotto-sezione,che spiega in dettaglio il package di cui il membro del gruppo è responsabile:tetrisattack.control.
Per questo package verrà presentato uno schema UML dettagliato e una descrizione esauriente di ogni classe del package.
Questo è lo schema UML dettagliato del package:



1.CheckCombo: la presenza di due costuttori (con parametri diversi) garantisce di poter operare sia su un singolo blocco che su due blocchi adiacenti (blocchi contenuti nel curosre). All'interno della classe toviamo dei campi come: toRemoveUpDown, toRemoveLeftRight, toRemoveLeftUp, toRemoveRightUp, removeLeft, RemoveRight. toRemoveUpDown e toRemoveLeftRight sono dei Set<Piece> che vengono usati solo nel caso in cui la classe sia invocata su un solo pezzo, mentre toRemoveLeftUp, toRemoveRightUp, removeLeft, RemoveRight sono Set<Piece> che vengono usati solo nel caso in cui la classe sia invocata sui I due pezzi contenuti nel cursore. Nel caso ci sia un solo pezzo, I pezzi trovati in su e in giù rispetto al pezzo in esame (controllati dai metodi checkUp e checkDown) vengono aggiunti al Set toRemoveUpDown mentre I pezzi trovati a destra e sinistra del pezzo in esame (controllati dai metodi chekLeft e checkRight) vengono aggiunti al Set toRemoveLeftRight. Successivamente viene chiamato il metodo checkListSize, con I due Set riempiti come input, che effettua un controllo sulla "lunghezza" dei set in input. Se entrambi I due Set risualtano maggiori o uguali a 3 allora unisce I set tramite il metodo join all'interno della medesima classe, Una volta uniti I set si procede con l'assegnamento del punteggio (moltiplicando il numero dei pezzi coinvolti per la costante X2) e chiama il metodo destroy che provvede ad eleminare I pezzi, mentre nel caso in cui solo uno dei set risulti "positivo" (maggiore o uguale a 3) allora assegna 2 punti e chiama il metodo destroy. Diversamente accade nel caso in cui la classe venga invocata sui due pezzi contenuti nel cursore. I metodi "checcher" vengono invocati su un pezzo per volta e in base al flag "left" (oppurtanamente settato di volta in volta) questi aggiungono i pezzi trovati alle rispettive liste (toRemoveLeftUp e removeLeft nel caso in cui left sia settato a true, toRemoveRightUp e removeRight diversamente). Viene poi chiamato il metodo checkListSize4p che controlla tutte e quattro I set (in maniera analoga a checkListSize), se tutte e quattro I set risultano "positivi" allora "joina" I set (mediante il metodo join), procede con l'assegnamento dei punti moltiplicando il numero dei pezzi coinvolti per la costante X4 (numero dei set "positivi") e così via per tutti gli altri casi. Se uno solo dei set risulta positivo allora viene assegnato il numero dei blocchi coinvolti come punteggio a meno che questi non siano 3, in questo caso viene assegano come punteggio la costante singlecombo. Alla fine dell'assegnamento dei punti provvede a chiamare il metodo destroy sul Set finale eventualmente "joinato".
2.ControlPlay: questa classe gestisce la difficoltà del gioco (velocità con cui il muro si avvicina alla sua altezza massima) e la concorrenza con cui I thread CheckCombo eliminano eventuali combo (bloccando temporaneamente l'ascesa del muro). Nel metodo run() troviamo un ciclo che si ripete fino a che il muro non è arrivato alla sua altezza massima. Questa classe fondamentalmente, nel metodo run, fa alzare il muro dei pezzi (se il numero dei thread checkCombo è uguale a 0) con la velocita dettata dalla difficoltà.

6. Testing

Per il testing dell'applicazione, si è preferito procedere con un testing "manuale" delle operazioni man mano che venivano implementate. Ogni qualvolta veniva rilevato un qualche tipo di "bug" nell'applicativo, esso veniva (possibilmente) rimosso. Il testing finale dell'applicazione è stato svolto seguendo alcuni passi:

1. Si è provveduto all'avvio del gioco,nel menu iniziale abbiamo selezionato il pulsante Credits;
2. Alla fine dello scorrere dei crediti siamo tornati automaticamente al menu iniziale cosi come ci aspettavamo,una volta li abbiamo cliccato sul pulsante HowToPlay;
3. Nel menu HowToPlay abbiamo fatto partire il tutorial,alla fine del quale abbiamo riscontrato che tutto quello che ci aspettavamo facesse è stato effettuato positivamente cosi come il ripristino delle condizioni iniziali per poi riavviare il tutorial altre 2 volte,infine premendo su back siamo tornati alla schermata precedente;
4. Abbiamo avviato poi il menu StageSelection scegliendo il primo pulsante in alto e settando la difficoltà su EASY;
5. Abbiamo testato i pulsanti di gioco e riscontrato che il cursore si spostava esattamente come precedentemente settato cosi come lo scambio di posizione dei pezzi;
6. Abbiamo poi testato la pressione sul key "c" per testare lo spostamento del muro verso l'alto e come ci aspettavamo alla "nascita" della nuova riga dal basso ha smesso di muoversi;
7. Poi abbiamo testato le varie combo che la partita ci ha "permesso" di effettuare e abbiamo riscontrato un bug,capitava che un pezzo risultasse in "distruzione" nonostante non fosse affiancato in alto,in basso o ai lati da pezzi di ugual colore in seguito a combo e distruzioni di gioco,una volta sistemato(errori dovuti ai semafori)non si è più ripresentato;

8. Infine abbiamo lasciato che il muro salisse fino alla sommità e da come ci aspettavamo il timer di gioco si è fermato ed è apparsa la schermata di fine gioco,abbiamo cliccato su pulsante “yes” e abbiamo scelto un nuovo stage e difficoltà MEDIUM e ricominciato a giocare;
9. Abbiamo riprovato a giocare e questa volta a cercare di aggiornare il punteggio massimo,ad ogni superamento di quest'ultimo la label corrispondente si è aggiornata in tempo reale per poi infine perdere di nuovo;
10. Infine abbiamo testato l'avvio di tutti gli stage rimasti e difficoltà HARD,all'ultimo stage abbiamo perso e nella schermata di fine cliccato su “no” e una volta riavviato il gioco abbiamo iniziato una nuova partita mediante il pulsante New Game.

7.Note finali

Seguono alcune note finali relative al progetto.

7.1 Descrizione del flusso di lavoro

Il flusso di lavoro risulta poco lineare a causa di alcune scelte errate e di una “spensierata” divisione dei compiti.
Un cambiamento abbastanza radicale si è verificato nella parte di Model che, partito inizialmente considerando i blocchi come JLabel gestite dal GridBagLayout(che avrebbe risparmiato notevoli sviluppi implementativi già offerti da queste classi) ha abbandonato l’idea in favore di un approccio più libero e dinamico offerto dal Canvas. Questo ha portato ad una moderata perdita di tempo di lavoro nella parte di modelling con una lieve ripercussione(in perdita di tempo) nella parte di controlling.
La divisione “spensierata” invece è dovuta ad una non adeguatamente approfondita conoscenza del Pattern MVC per buona parte del tempo che ha portato ad una notevole criticità che ha reso difficile comprendere i compiti esatti di ognuno, portando ad una divisione errata e pure uno squilibrio fra le parti. Appena rinvenuti gli errori, il non aver colmato questo squilibrio è una volontà di chi ne aveva diritto e dovere.
Nel complesso credo che, nonostante i problemi precedenti, il risultato sia un buon lavoro.

7.2 Evaluation del risultato finale

In quest’ultima sezione verrà data una valutazione finale al progetto.
Tutti i membri sono concordi nell’affermare che la creazione di questo progetto è stata un’esperienza unica ed estremamente appagante, soprattutto visti i risultati: il videogame creato è divertente e vario con la possibilità data all'utente finale di scegliere lo stage di gioco, la difficoltà con cui giocare,apprendere come giocare in modo unico,visualizzare i crediti di gioco.
L’impegno profuso nella realizzazione del progetto si può evincere dalle numerose funzioni implementate, e a come si sia cercato di fare in modo che lo scorrere del gioco risultasse interessante per l'utente e tutti i casi(le combo di pezzi,la caduta di quest'ultimi al seguito di spostamenti o combo precedenti) venissero gestiti correttamente. Nonostante qualche piccola difficoltà durante lo sviluppo, logicamente legata al fatto che la creazione di un videogame era una cosa nuova per tutti i membri, il prodotto finale è ben al di sopra delle aspettative iniziali dei membri, che dunque valutano il proprio lavoro in maniera **positiva**.