



A.D. 1308
unipg
DIPARTIMENTO
DI INGEGNERIA

Tesina Finale di
Programmazione di Interfacce Grafiche e Dispositivi Mobili
Corso di Laurea in Ingegneria Informatica ed Elettronica – A.A. 2021-2022
DIPARTIMENTO DI INGEGNERIA

docente
Prof. Luca GRILLI

JHelicopter

applicazione desktop JFC/SWING



Studenti

326580
321306

Erigen
Leonardo
Ignazio

Partalli
Pagliochini

erigen.partalli@studenti.unipg.it
leonardoignazio.pagliochini@studenti.unipg.it

Sommario

1. Descrizione del Problema	3
Il Videogioco Flappy Bird.....	3
L'applicazione JHelicopter	4
2. Specifica dei Requisiti.....	4
3. Progetto	5
3.1 Architettura del Sistema del Software	5
3.2 Model.....	6
GameObject.....	7
Helicopter	8
Beam & BeamColumn.....	9
Coin & Coins	11
3.3 View	13
MainFrame	13
Scores.....	14
Skins	14
Pause.....	14
HelicopterPanel.....	14
GameOver.....	15
Sound	15
Sounds Package.....	15
3.4 Controller.....	16
ControllerData	17
UIController	18
GameController.....	19
CollisionController.....	19
GameKeyController.....	20
TimeController	20
AnimationController	23
3.5 Diagramma UML Completo	23
3.6 Problemi Riscontrati.....	24
4. Conclusioni e Sviluppi Futuri	25

1. Descrizione del Problema

L'obiettivo di questo lavoro è lo sviluppo di un'applicazione desktop, denominata JHelicopter, che realizza una versione modificata della grafica del rinomato videogioco "Flappy Bird".

L'applicazione sarà implementata utilizzando la tecnologia JFC/Swing in modo da favorire un'ampia portabilità su diversi sistemi operativi (piattaforme), riducendo al minimo eventuali modifiche al codice sorgente.

Di seguito sarà data una breve descrizione del gioco originale seguita da quella della versione realizzata.

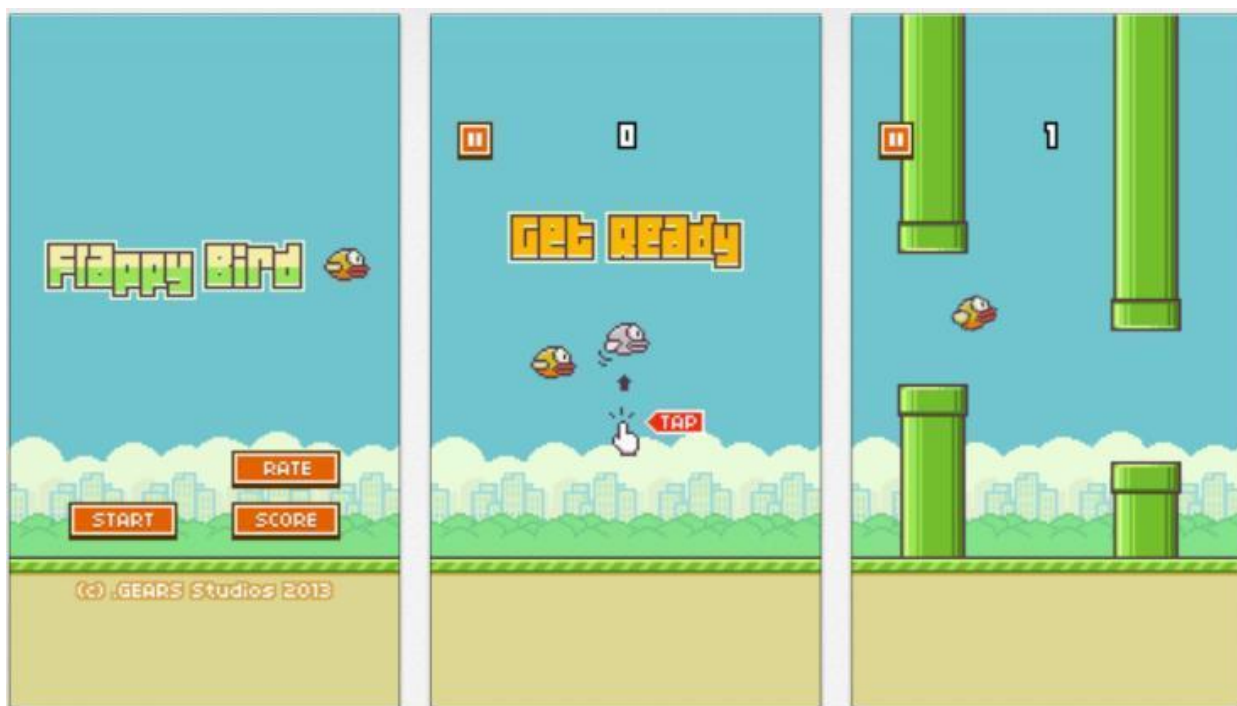
Il Videogioco Flappy Bird

Flappy Bird è un videogioco sviluppato dal programmatore vietnamita Dong Nguyen e distribuito a partire da maggio 2013 su AppStore e PlayStore; è stato un videogioco di gran successo grazie ad una dinamica di gioco intuitiva basata su uno stile simile a quello dei videogiochi anni Ottanta.

È un gioco a scorrimento continuo in cui lo scopo è quello di totalizzare il punteggio più alto possibile facendo volare un uccellino **attraverso una serie di tubi evitando di farlo scontrare contro essi o di farlo cadere a terra.**

In assenza di input da parte del giocatore, l'uccello cade verso il basso. Ad ogni pressione dello schermo corrisponde un solo battito delle ali che comporta un innalzamento minimo.

Nonostante i grandi introiti che il gioco generava, nel 10 febbraio 2014 quest'ultimo venne rimosso dagli store rimanendo però disponibile per chi lo aveva già scaricato.



Le tre schermate rappresentano rispettivamente la homepage di gioco, una spiegazione intuitiva dei comandi e il gioco in esecuzione

L'applicazione JHelicopter

L'applicazione JHelicopter si propone di offrire all'utente un'esperienza di gioco molto simile a quella del gioco originale Flappy Bird, ma modificandone l'estetica ed implementando suoni, animazioni e funzioni aggiuntive.

L'utente visualizzerà quindi, nella propria schermata, un elicottero che potrà comandare cliccando sullo schermo o utilizzando la barra spaziatrice in modo da evitare che questo vada ad impattare con delle travi da costruzione.

A differenza del gioco originale saranno presenti delle monetine che, nel caso in cui vengano raccolte dall'elicottero, faranno in modo che per otto secondi questo possa attraversare gli ostacoli.

2. Specifica dei Requisiti

L'applicazione JHelicopter che si intende realizzare dovrà soddisfare i seguenti requisiti.

Il programma deve presentare cinque schermate: la schermata di avvio, la schermata di gioco, un menu adibito alla scelta della skin, un menù di pausa ed una schermata di *GAME-OVER*

La schermata di avvio deve presentare un campo adibito all'inserimento del nome utente del giocatore e tre bottoni per avviare la partita, entrare nel menù della scelta delle skin e per accedere alla schermata relativa ai punteggi

Il sistema deve restituire un errore nel caso in cui si provi ad avviare una partita senza inserire un username

Il menu della scelta delle skin permetterà al giocatore di scegliere l'estetica del proprio elicottero cliccando su una delle tre disponibili.

La schermata dei record dovrà presentare i tre punteggi più alti raggiunti con il relativo username.

La schermata di gioco sarà occupata per la maggior parte dal campo di gioco e in alto sarà presente un contatore del punteggio, un campo in cui verrà visualizzato il record attuale, un contatore di otto secondi che verrà mostrato a schermo solo nel caso in cui vengano raccolte delle monete e un collegamento al menù di pausa.

Il menù di pausa deve presentare due opzioni: riprendere la partita o uscire dall'applicazione. Ogni volta che l'elicottero colpirà una trave, al di fuori del bonus concesso dalla moneta, verrà visualizzata a schermo una schermata di game-over.

La schermata di game-over riporterà il punteggio raggiunto, un'animazione nel caso in cui si sia superato il record, un tasto per tornare al menù principale ed un tasto per avviare una nuova partita.

L'applicazione dovrà contenere diversi effetti sonori:

Effetto sonoro dell'elicottero che esplode sbattendo sulla trave

Il suono dell'elicottero che supera una trave

Suono del record superato

Effetto sonoro della moneta che viene raccolta

Effetto sonoro dell'elicottero che accelera
 Musica di sottofondo durante la partita
 L'applicazione presenterà diverse animazioni
 Animazione dell'esplosione
 Animazione quando viene raccolta una moneta

3. Progetto

Di seguito viene riportata la struttura dell'applicazione realizzata, illustrandone prima l'architettura software per poi scendere nel dettaglio dei blocchi funzionali che la compongono.

3.1 Architettura del Sistema del Software

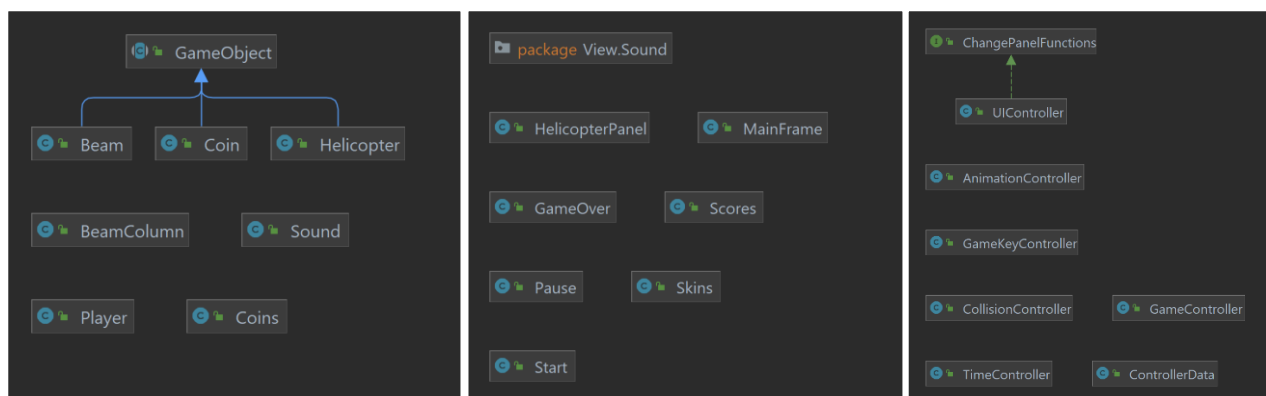
Per la realizzazione dell'applicazione è stato adottato un approccio MVC (Model, View, Controller) .

Il package Model si occupa di raggruppare i dati relativi agli elementi logici necessari al funzionamento del gioco (elicottero, travi, moneta, giocatore...).

Il package Controller raggruppa diverse classi relative alla gestione della logica fondamentale del gioco adibite all'esecuzione di istruzioni in funzione del verificarsi di eventi.

Il package View si occupa di rappresentare i dati all'utente raccogliendone l'input; questo è suddiviso in classi rappresentanti diverse schermate.

Con il seguente diagramma viene rappresentata l'architettura software.



3-a)

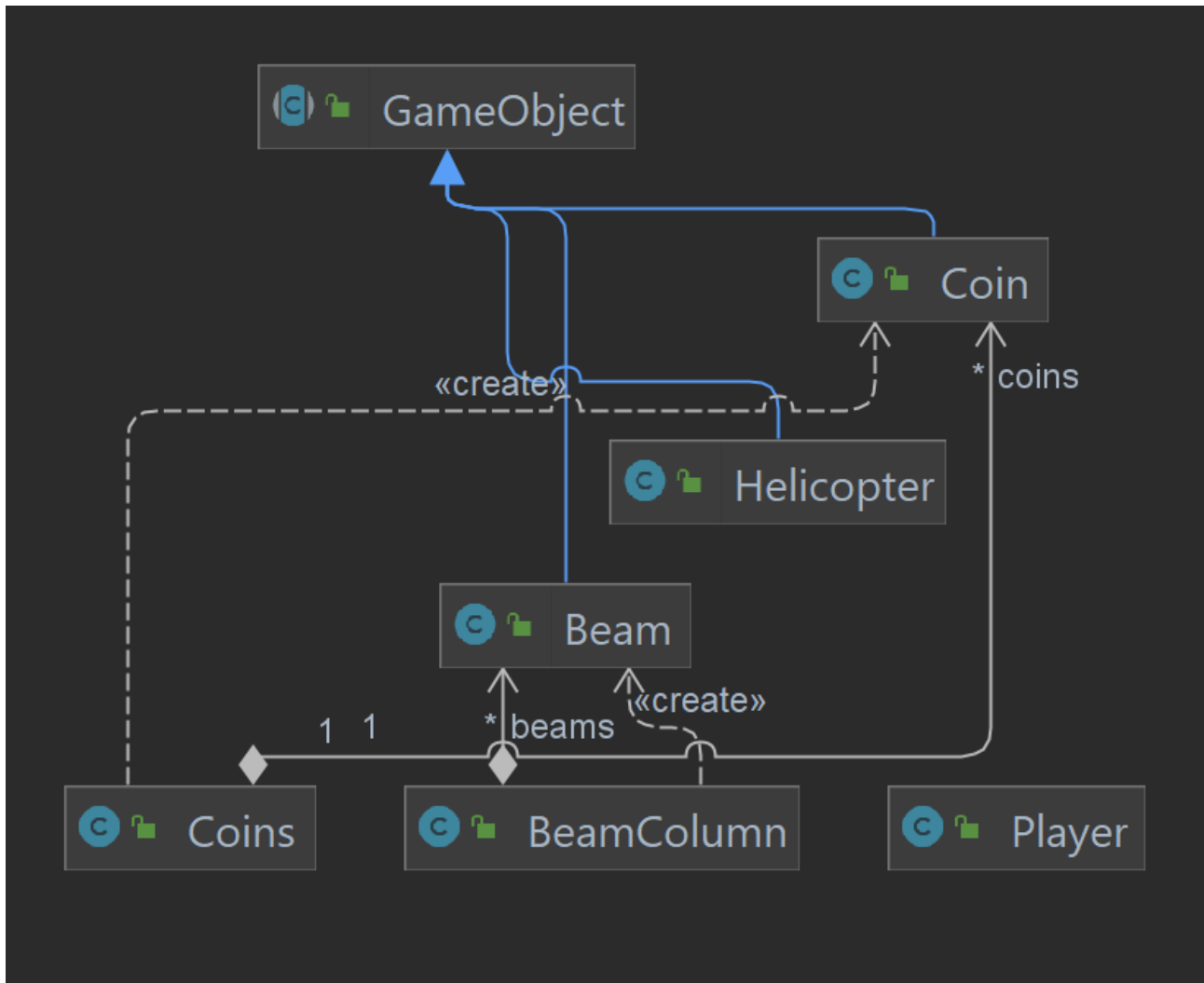
3-b)

3-c)

Nelle figure 1-a), 1-b) e 1-c) sono riportate le classi che compongono Model, View e Controller

3.2 Model

All'interno del package Model sono presenti nello specifico sette classi.



GameObject

La classe GameObject rappresenta un generico oggetto visualizzato a schermo durante l'esecuzione del gioco.

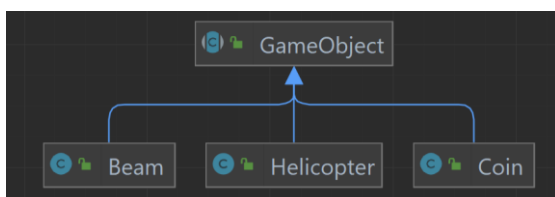
GameObject		
m	GameObject (int, int)	
f	dy	int
f	height	int
f	width	int
f	y	int
f	dx	int
f	x	int
f	image	Image

Ogni GameObject ha come attributi le coordinate all'interno dello spazio di gioco, la velocità di variazione di tali coordinate nel tempo, le dimensioni dell'oggetto ed un campo immagine rappresentante l'icona dell'oggetto.

GameObject presenta anche due funzione astratte:

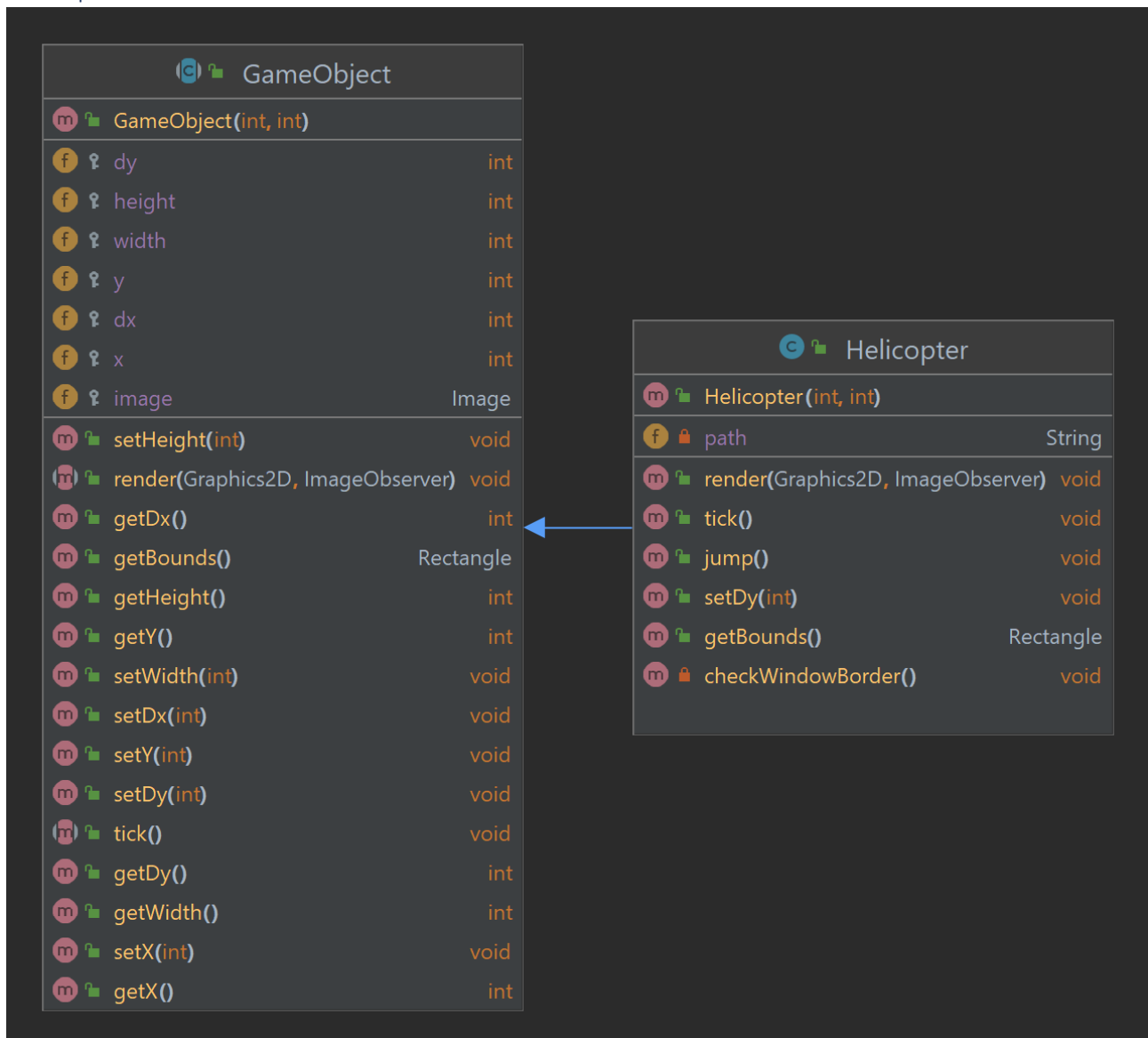
- **"tick()"** necessaria per definire le eventuali variazioni della dinamica dell'oggetto durante il tempo di gioco
- **"render()"** necessaria per visualizzare l'immagine.

GameObject		
m	setHeight (int)	void
m	render(Graphics2D, ImageObserver)	void
m	getDx ()	int
m	getBounds ()	Rectangle
m	getHeight ()	int
m	getY ()	int
m	setWidth (int)	void
m	setDx (int)	void
m	setY (int)	void
m	setDy (int)	void
m	tick ()	void
m	getDy ()	int
m	getWidth ()	int
m	setX (int)	void
m	getX ()	int



Ad estendere la classe GameObject sono presenti tre classi rappresentanti gli elementi base del campo di gioco.

Helicopter



La classe **Helicopter** rappresenta l'elicottero che il giocatore muove durante la partita.

Gli attributi dell'elicottero sono gli stessi di **GameObject** con l'aggiunta di un campo "path" adibito a contenere l'indirizzo dell'immagine da inserire come icona.

Oltre alla funzione "**tick()**", che nel caso dell'elicottero ricalcola la posizione e l'icona, sono presenti le funzioni di "**jump()**", la funzione "**checkWindowBorder()**" e "**render()**".

La funzione di "**jump()**", come riportato nel capitolo Controller (sotto capitolo **GameKeyController**), viene richiamata con pressione del tasto spazio e decrementa la velocità lungo l'asse y in modo da simulare un salto verso l'alto.

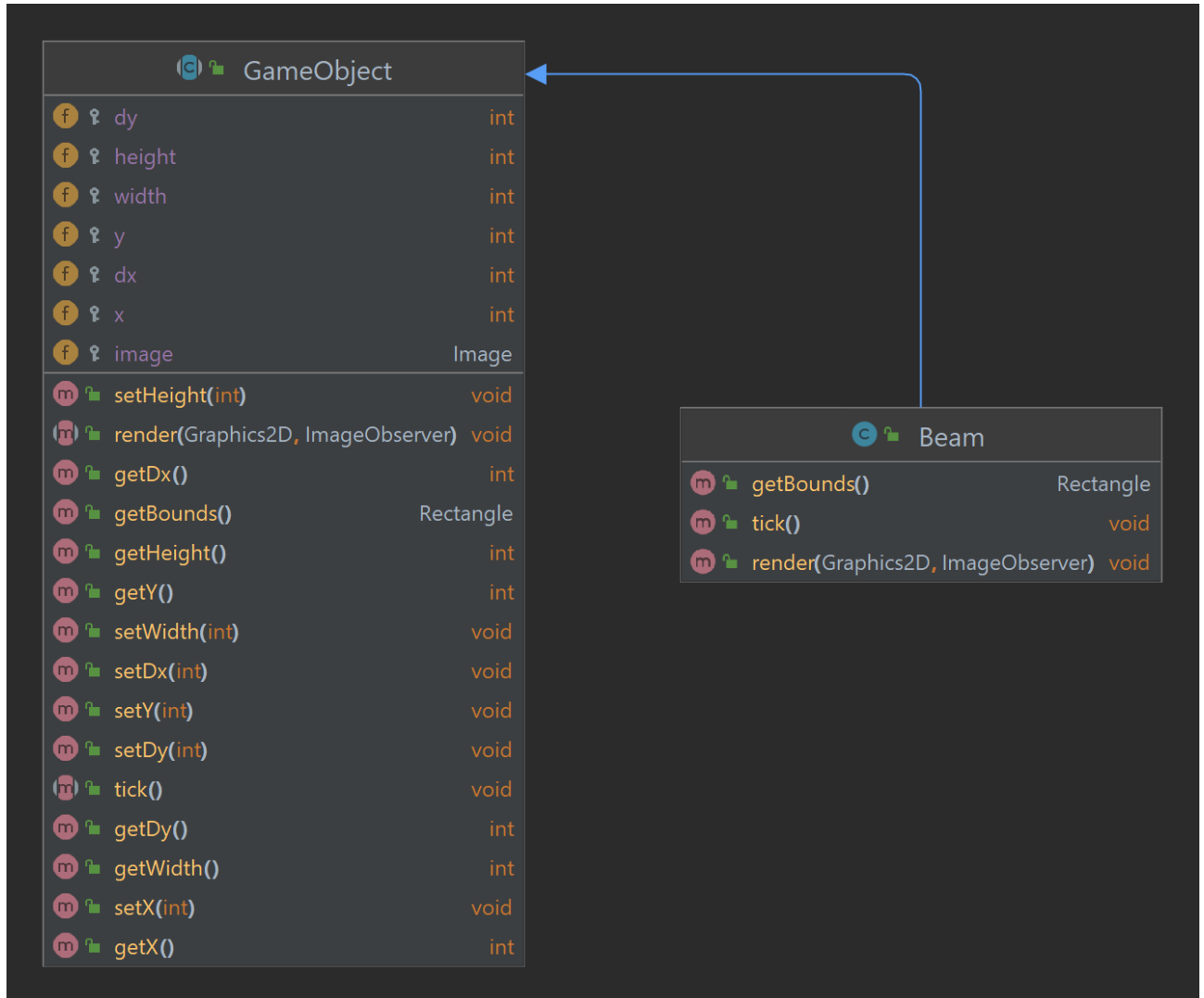
La funzione "**checkWindowBorder()**" viene richiamata all'interno della funzione "**tick()**" e ha lo scopo di evitare che l'elicottero esca dal campo visivo.

La funzione **"GetBounds()"** permette di ottenere un oggetto Rettangolo che delimita l'elicottero e viene utilizzato nel calcolo delle collisioni.

9

Beam & BeamColumn

Beam



La classe Beam rappresenta gli elementi costitutivi delle travi presenti all'interno del gioco. Anche la classe Beam estende la classe **GameObject()** con la differenza che la funzione **"tick()"** incrementa solamente l'ascissa dell'elemento; anche in questo caso è presente la funzione **"getBounds()"** utilizzata per il rilevamento di eventuali collisioni.

BeamCoulumn

BeamColumn		
f	speed	int
f	base	int
f	beams	List<Beam>
f	changeSpeed	int
f	random	Random
f	points	int
m	setPoints(int)	void
m	tick()	void
m	initColumn()	void
m	render(Graphics2D, ImageObserver)	void
m	getBeam()	List<Beam>
m	setBeam(List<Beam>)	void
m	getPoints()	int

La classe BeamColumn rappresenta le travi composte dalla sovrapposizione di elementi **Beam**.

Funzione fondamentale della classe **BeamColumn** è “**initColumn()**” la quale ha lo scopo di generare, attraverso un oggetto Random, la trave rappresentata da una lista di **Beam** facendo in modo che questi si sovrappongano l’uno all’altro senza intersecarsi e lasciando uno spazio vuoto pari a quattro volte l’altezza di un Beam per permette il passaggio dell’elicottero.

All’interno della classe sono presenti tre attributi fondamentali quali la velocità di movimento della colonna (speed), il punteggio conseguito(point), ed un valore di punteggio oltre il quale la speed aumenta

di 1 (changeSpeed).

Il valore della speed è di sistema impostato a 5.

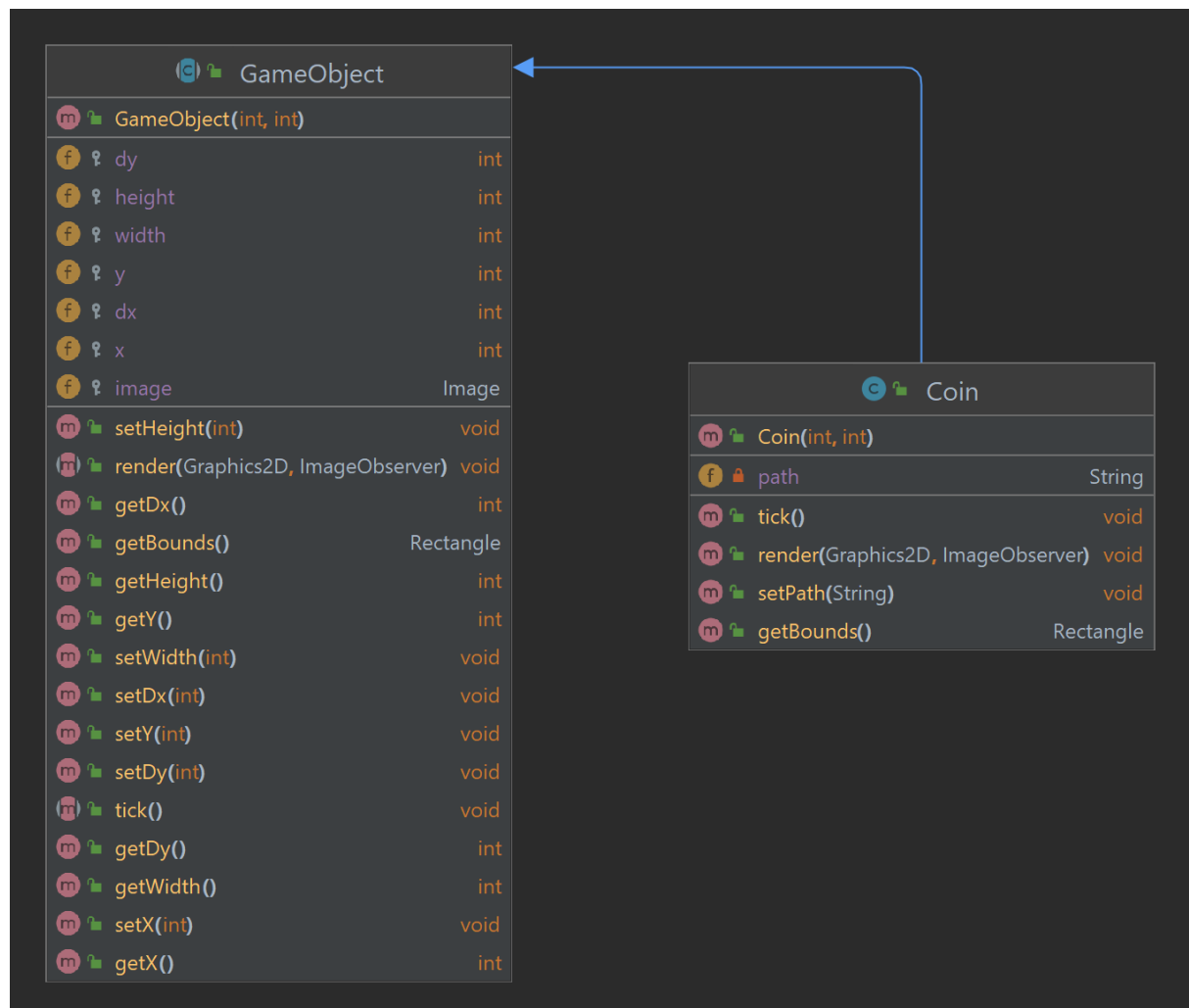
La changeSpeed ha un valore iniziale pari a 5 ed aumenta di 5 ogni qual volta il punteggio la eguagli.

Dunque la velocità aumenta di 1 ogni 5 punti.

La funzione “**tick()**” esegue “**Beam.tick()**” su tutti gli elementi della lista, elimina tutti gli elementi della lista se sono finiti al di fuori del campo visivo e nel caso in cui la lista sia vuota (la colonna è finita al di fuori del campo visivo) inizializza una nuova colonna ed incrementa la variabile point.

La funzione “**render()**” non fa altro che richiamare la funzione “**Beam.render()**” per tutti gli elementi della lista.

Coin



La classe **Coin** rappresenta la moneta e viene utilizzata dalla classe **Coins** che tiene conto delle monete che vengono visualizzate a schermo

La funzione **“tick()”** si occupa del movimento orizzontale e verticale (attivato solo per l’animazione a seguito della collisione con la moneta). La velocità di spostamento verticale viene impostata pari a quella delle travi.

La funzione **“setPath()”** ha lo scopo di poter modificare in corsa l’icona della moneta in modo da permettere l’esecuzione dell’animazione.

Infine **“getBounds()”** viene utilizzata per l’analisi delle collisioni.

Coins

Coins		
m	Coins()	
f	speed	int
f	coins	List<Coin>
f	random	Random
m	initCoin()	void
m	getCoins()	List<Coin>
m	render(Graphics2D, ImageObserver)	void
m	tick()	void

La classe Coins rappresenta tutte le monete presenti nel campo di gioco e ne tiene conto tramite l'utilizzo di una lista di Coin.

In corrispondenza del **tick()** setta la velocità della moneta pari al valore di velocità delle colonne e controlla che la moneta non abbia raggiunto un certo valore di x dopo il quale la moneta viene eliminata e ne viene inizializzata un'altra.

L'inizializzazione di una moneta è delegata alla funzione "**initCoin()**" che genera una moneta un valore randomico di y e la inserisce all'interno della lista.

Player

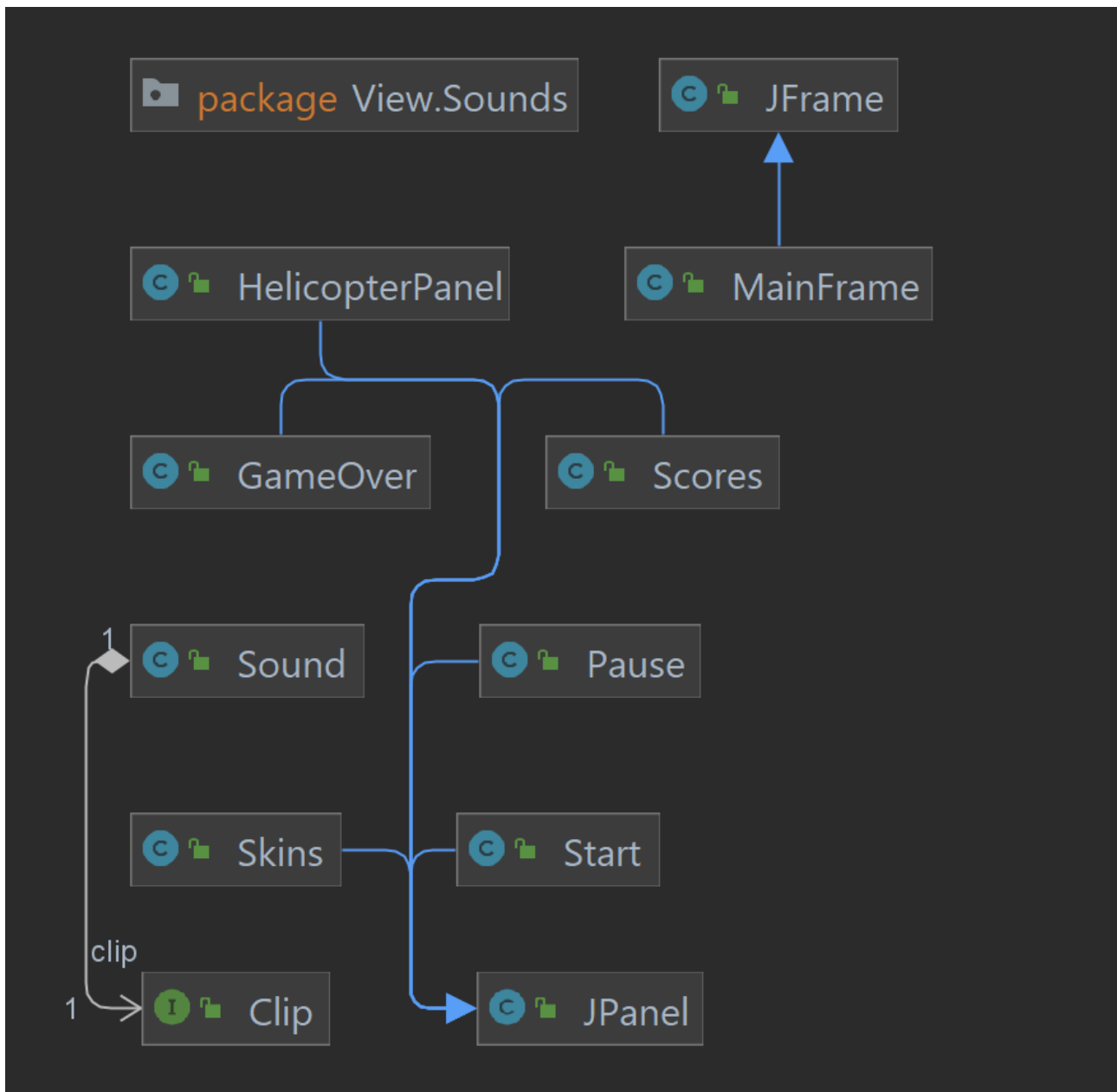
La classe player tiene conto di tutti i dati relativi al giocatore quali:

Username che viene inserito tramite la funzione "**setUsername()**" ad inizio partita.

Score, ovvero il punteggio ottenuto dal giocatore che viene inserito tramite la funzione "**setScore()**" a fine partita.

La funzione più importante della classe è la funzione "**insertPlayer()**" che tramite una funzione ausiliaria "**insertAt()**" permette di inserire i dati del giocatore all'interno del documento contenente i punteggi; questo avviene solo nel caso in cui il punteggio conseguito superi almeno uno di quelli registrati, sovrascrivendo quindi Username e Score del player con punteggio minore presente nel file

Player		
m	Player()	
f	username	String
f	score	int
m	setUsername(String)	void
m	getScore()	int
m	InsertPlayer()	void
m	getUsername()	String
m	insertAt(int)	void
m	setScore(int)	void



All'interno del package View sono presenti 8 classi ed un package relativo ai vari effetti sonori

MainFrame

Il MainFrame è una classe che estende JFrame ed ha come unico metodo il costruttore che setta alcuni parametri come la dimensione, imposta la dimensione del frame non modificabile, ecc.

Start

La classe Start estende la classe JPanel e presenta bottoni ed altre componenti swing come da requisiti di progetto.

Scores

Scores	
Scores()	
backButton	JButton
username5	JLabel
point2	JLabel
username1	JLabel
username3	JLabel
point1	JLabel
point4	JLabel
username2	JLabel
username4	JLabel
point5	JLabel
point3	JLabel
initComponents()	void
max()	int

La classe scores estende anch'essa JPanel e riporta attraverso l'ausilio di JLabel i dati contenuti all'interno del file "punteggi.csv".

All'interno della classe scores è implementata una funzione d'utilità che permette di ottenere il punteggio massimo presente all'interno del file di punteggi

Skins

Skins	
Skins()	
skinPath	String
backButton	JButton
recoverySkinPath	String
skinButton2	JButton
skinButton3	JButton
textLabel	JLabel
skinButton1	JButton
skin1()	void
initComponents()	void
skin3()	void
jButton2ActionPerformed(ActionEvent)	void
skin2()	void

La classe Skins estende JPanel e presenta tra le altre cose 3 bottoni che avviano 3 metodi differenti atti a modificare la skin di gioco.

I seguenti metodi sono skin1(), skin2() e skin3() che modificano il valore degli attributi skinPath e skinRecoveryPath.

Pause

La classe Pause estende la classe JPanel e presenta bottoni ed altre componenti swing come da requisiti di progetto.

HelicopterPanel

HelicopterPanel	
HelicopterPanel()	
proxylImage	ProxyImage
background	Image
beamColumn	BeamColumn
coins	Coins
gameKeyController	GameKeyController
helicopter	Helicopter
paint(Graphics)	void
getHelicopter()	Helicopter
getBeamColumn()	BeamColumn
getCoins()	Coins

L'HelicopterPanel è il pannello all'interno del quale si svolge effettivamente il gioco

La classe HelicopterPanel estende JPanel e sovrascrive il metodo paint() permettendo di inserire un background e disegnando o meno componenti in funzione del parametro isRunning presente all'interno della classe ControllerData trattata più avanti.

In aggiunta, presenta alcuni metodi per restituire gli oggetti Coins, BeamColumn ed Helicopter dichiarati al suo interno.

GameOver

La classe GameOver estende la classe JPanel e presenta bottoni ed altre componenti swing come da requisiti di progetto.

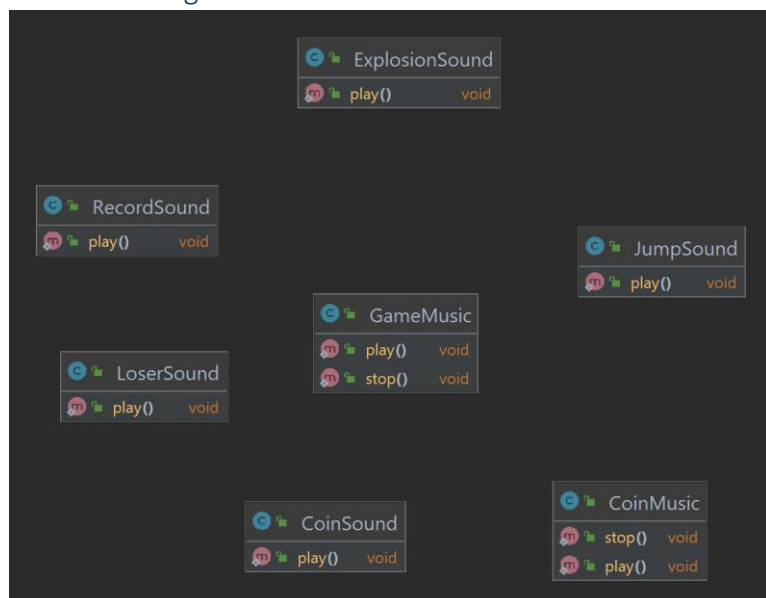
Sound

Icona	Nome	Return Type
	Sound	
	Sound()	
	clip	Clip
	stop()	void
	loop()	void
	setFile(String)	void
	getClip()	Clip
	play()	void

La classe sound è una classe sulla quale si basano tutte le classi relative ai vari effetti sonori e presenta diversi metodi per far partire, stoppare, mettere in loop un certo file sonoro.

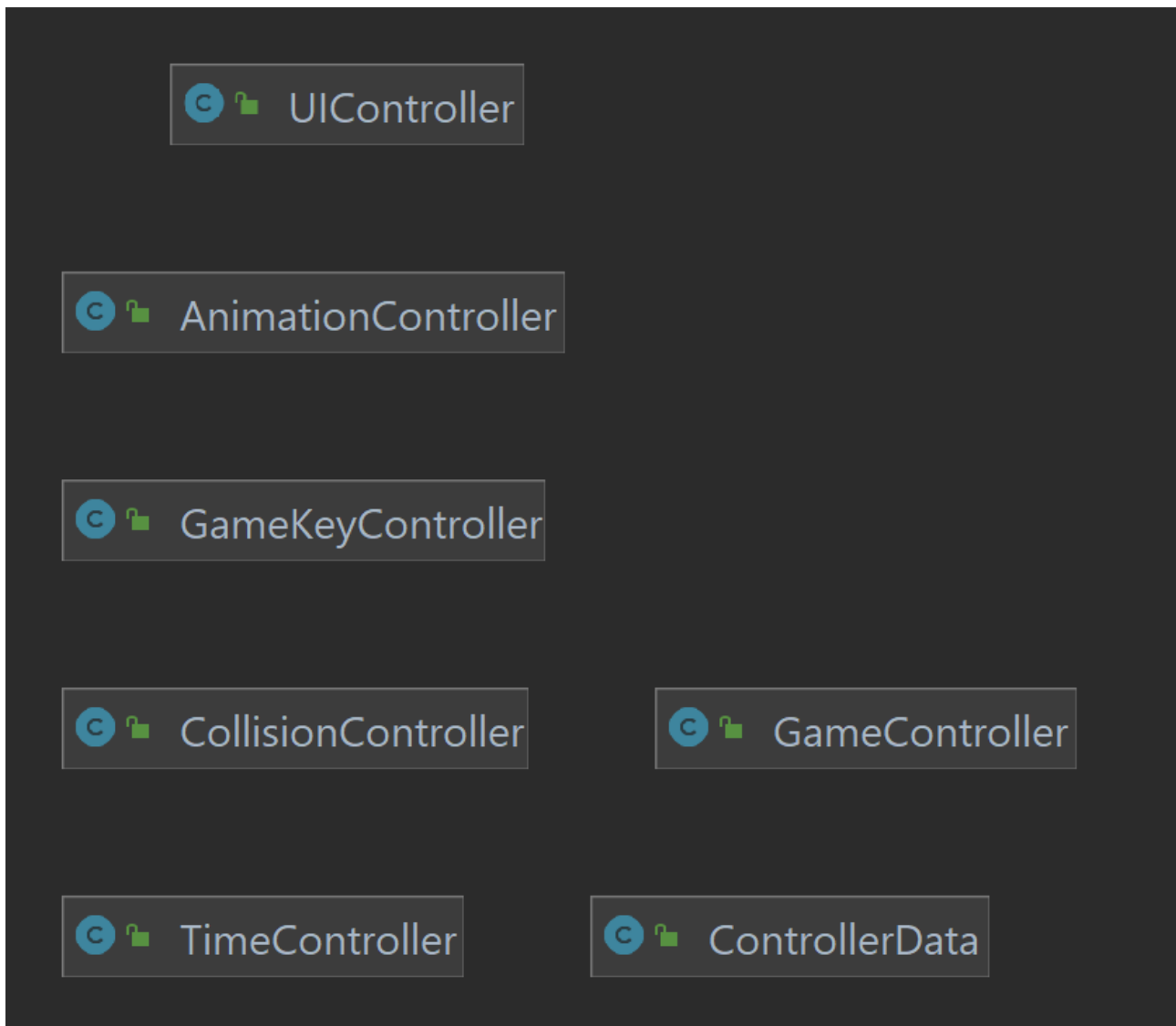
Questo è possibile mediante l'utilizzo di un oggetto di tipo Clip

Sounds Package



All'interno del package sound sono presenti delle classi relative ai vari effetti sonori (una per effetto). Utilizzare una classe per suono è una scelta sicuramente ridondante ma permette di rendere il codice più snello e meno complesso.

All'interno delle varie classi è presente sempre il metodo play, mentre il metodo stop è presente solo in effetti sonori di durata maggiore.



All'interno del package controller sono presenti 7 classi che si occupano di gestire aspetti diversi della dinamica di gioco

All'interno della classe ControllerData sono raggruppate le variabili che rappresentano lo stato del videogioco. Tali variabili sono necessarie per il funzionamento delle altre classi presenti nel package.

Di seguito vengono riportate e spiegate singolarmente.

ControllerData		
	<code>explosionMstimer</code>	<code>int</code>
	<code>endingGame</code>	<code>boolean</code>
	<code>xSpeed</code>	<code>int</code>
	<code>ySpeed</code>	<code>int</code>
	<code>beamCollision</code>	<code>boolean</code>
	<code>coinMstimer</code>	<code>int</code>
	<code>exceedColumn</code>	<code>boolean</code>
	<code>cointime</code>	<code>double</code>
	<code>xSpeedRecovery</code>	<code>int</code>
	<code>score</code>	<code>int</code>
	<code>isRunning</code>	<code>boolean</code>
	<code>highScore</code>	<code>int</code>
	<code>coinIntersect</code>	<code>boolean</code>
	<code>coinPath</code>	<code>String</code>
	<code>paused</code>	<code>boolean</code>
	<code>ySpeedRecovery</code>	<code>int</code>

- **xSpeed**: riporta la velocità di tutte le componenti che si muovono sull'asse orizzontale.
- **ySpeed**: riporta la velocità di tutte le componenti che si muovono sull'asse verticale.
- **xSpeedRecovery**: variabile d'appoggio utilizzata per memorizzare il valore di xSpeed nel caso in cui il gioco debba essere freezato.
- **ySpeedRecovery**: variabile d'appoggio utilizzata per memorizzare il valore di ySpeed nel caso in cui il gioco debba essere freezato.
- **paused**: variabile booleana che viene impostata a true nel caso in cui il gioco venga messo in pausa.
- **endingGame**: variabile booleana che viene impostata a true nel caso in cui il gioco stia finendo (utilizzata per la gestione delle animazioni)
- **beamCollision**: variabile booleana che viene impostata a true a seguito di una collisione con una trave.
- **isRunning**: variabile booleana che identifica se il gioco è in esecuzione.
- **coinIntersect**: variabile booleana che viene impostata a true a seguito di una collisione con una moneta
- **exceedColumn**: è una variabile booleana impostata a true nel caso in cui venga superata una trave.
- **coinPath**: è un campo Stringa all'interno del quale è riportato il percorso dell'icona della moneta (funzionale all'animazione di rotazione)
- **Score**: variabile di tipo int che tiene conto del punteggio.
- **highScore**: variabile di tipo int che riporta il punteggio massimo conseguito nelle partite precedenti
- **explodingMstimer e coinMstimer**: sono due variabili int che rappresentano due timer in millisecondi necessari per l'esecuzione delle due animazioni presenti nel gioco
- **coinTime**: variabile utilizzata per tenere conto della durata dell'effetto della moneta.

All'interno del codice della classe controllerData sono presenti, in fine, delle funzioni di utilità per ottenere e settare i valori delle variabili sopracitate.

UIController

La classe **UIController** si occupa di controllare tutte le operazioni legate alla visualizzazione dei vari pannelli all'interno del **mainFrame**.

All'interno della classe vengono dichiarate ed inizializzate tutte le classi viste nel package view relative a schermate di gioco e una componente mainFrame all'interno della quale queste vengono visualizzate.

UIController	
player	Player
start	Start
scores	Scores
skins	Skins
mainFrame	MainFrame
gameOver	GameOver
helicopterPanel	HelicopterPanel
pause	Pause
backFromPause ()	void
scores()	void
helicopter ()	void
setPanel (JFrame, JPanel, JPanel)	void
resume()	void
gameOver()	void
start ()	void
playAgain ()	void
backFromSkins ()	void
skins()	void
backFromScores()	void
backFromGameOver()	void
pause ()	void

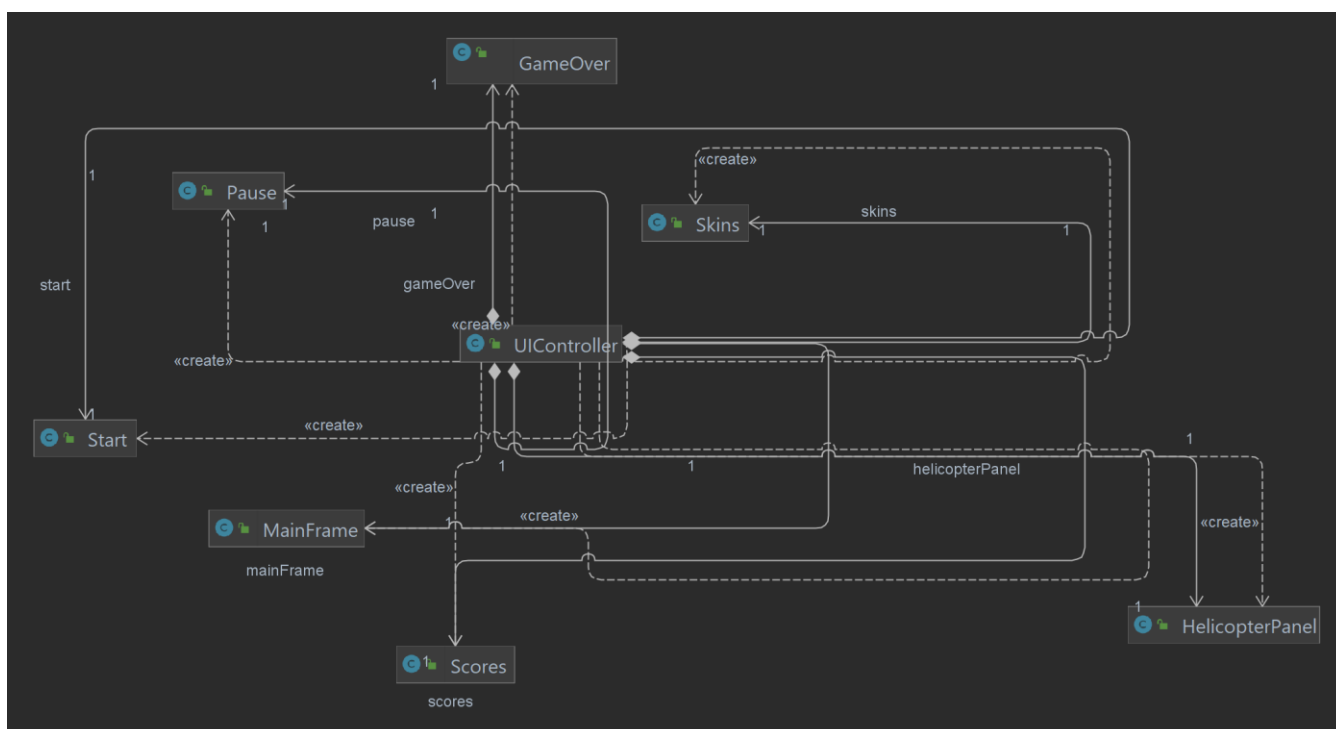
Tutte le funzioni che comportano un cambio di schermata sfruttano la funzione privata **setPanel()** che, attraverso i metodi *frame.add* e *frame.remove*, permette di cambiare JPanel visualizzato all'interno del frame.

La funzione *start()* viene eseguita dalla classe main e carica la prima schermata (ovvero quella del menù di start).

Tutte le altre operazioni vengono eseguite a seguito del verificarsi di condizioni particolari:

- *gameOver()* viene richiamata dalla funzione *endGame()* presente all'interno della classe *GameController*
- *pause()* viene richiamata, a seguito della pressione del tasto p durante il gioco, dalla classe *GameKeyController*.

Tutte le altre funzioni vengono eseguite successivamente alla pressione di bottoni presenti all'interno delle varie schermate.



GameController

GameController		
	<code>endGame()</code>	<code>void</code>
	<code>closeGameFromPause()</code>	<code>void</code>
	<code>getPlayer()</code>	<code>Player</code>
	<code>restartGame()</code>	<code>void</code>
	<code>resumeGame()</code>	<code>void</code>
	<code>stopGame()</code>	<code>void</code>

All'interno della classe GameController sono presenti diversi metodi che vengono richiamati al verificarsi di determinate condizioni durante il gioco.

- 1) **restartGame()** viene richiamato alla pressione del tasto "Invio" sulla schermata di gioco, setta il parametro *isRunning* a true, inserisce l'username del player e resetta alcuni parametri che potrebbero essere stati modificati nella partita precedente.
- 2) **endGame()** viene richiamato a seguito dell'intersezione tra un elicottero ed una trave ed inizializza un timer di un secondo durante il quale si svolge l'animazione di esplosione dell'elicottero. A seguito del timer viene eseguita l'effettiva conclusione del gioco in cui viene impostata a *false* la variabile *ControllerData.isRunning*, viene settato il punteggio del giocatore pari al punteggio conseguito ed in funzione di quest'ultimo viene riprodotto il suono di vittoria o di sconfitta.
- 3) **stopGame()** è una funzione utilizzata per freezare il gioco. Prevede che vengano impostate a zero le due variabili relative al movimento delle componenti e che venga impostata a *true* la variabile *ControllerData.paused*.
- 4) **resumeGame()** è la funzione adibita a riprendere la partita a seguito della schermata di pausa e non fa altro che resettare le velocità ai valori di recovery (pari a quelli precedenti al menù di pausa) ed impostare il valore di *ControllerData.paused* pari a *false*.
- 5) **closeGameFromPause()** è il metodo che viene inizializzato quando si torna al menu start dal menu di pausa e si occupa di resettare alcune variabili in vista della partita successiva.

In fine il metodo *getPlayer()* è un metodo di utilità per poter ottenere la variabile player da altre classi.

CollisionController

CollisionController		
	<code>rectHelicopter</code>	<code>Rectangle</code>
	<code>rectBeam</code>	<code>Rectangle</code>
	<code>rectCoin</code>	<code>Rectangle</code>
	<code>checkBeamExceed()</code>	<code>void</code>
	<code>checkCollision()</code>	<code>void</code>

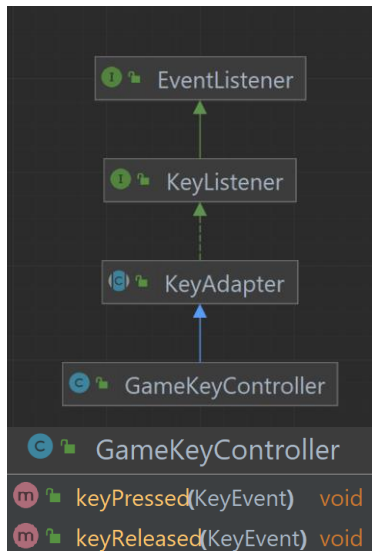
La classe CollisionController si occupa di effettuare operazioni di verifica su condizioni dipendenti dai rettangoli che iscrivono i vari elementi di gioco.

In particolare, la funzione **checkCollision()** si occupa di individuare l'eventuale intersezione del rettangolo che delimita l'elicottero con il rettangolo delimitante la trave (o meglio con tutti i rettangoli che delimitano ogni *beam*) e con il rettangolo delimitante la moneta.

Nel caso dell'intersezione con la trave, se il valore di *ControllerData.coinIntersect* è pari a *false*, viene settato a *true* il valore di *ControllerData.beamCollision* mentre nel caso di collisione con la moneta viene settato a *true* il valore di *ControllerData.coinIntersect*.

La Funzione **checkBeamExceed()** controlla se l'elicottero supera la trave e nel caso in cui ciò avvenga riproduce un suono e imposta a *true* il valore di *ControllerData.exceedColumn*.

GameKeyController



La classe game key controller estende la classe **keyAdapter** e si occupa di gestire tutti gli input da tastiera necessari per lo svolgimento del gioco.

La classe sovrascrive il metodo **keyPressed()** inserendo le seguenti istruzioni a seconda del tasto premuto:

- **ENTER**: esegue `gameController.restartGame()`
- **SPAZIO**: riproduce il suono del salto
- **P**: richiama `gameController.stopGame()` sopracitata e il metodo presente in `UIController` per passare alla schermata di pausa

Sovrascrive poi il metodo **keyReleased()** che nel caso in cui venga rilasciato il tasto **SPAZIO** esegue l'istruzione *jump* presente in `Helicopter`.

TimeController

La classe time controller è forse la classe più importante per il funzionamento del gioco in quanto regola tutte quelle azioni che devono essere fatte in maniera ciclica nell'arco della partita.

Attraverso il Timer, inizializzato attraverso la funzione `start`, permette di svolgere ripetutamente il seguente codice ogni 15 millisecondi (valore impostato in fase di progettazione):

```

Toolkit.getDefaultToolkit().sync();
if (ControllerData.isIsRunning()) {
    //////////////////////////////////////
    UIController.helicopterPanel.getCoins().tick();
    UIController.helicopterPanel.getHelicopter().tick();
    UIController.helicopterPanel.getBeamColumn().tick();
    if (!ControllerData.isExceedColumn())
        CollisionController.checkBeamExceed();

    if (ControllerData.isBeamCollision()) {
        GameController.endGame();
        ControllerData.setBeamCollision(false);
    }
    if (!ControllerData.isEndingGame())
        CollisionController.checkCollision();
    ControllerData.setScore(UIController.helicopterPanel.beamColumn.getPoints());
    if (ControllerData.isCoinIntersect() && !ControllerData.isPaused()) {
        ControllerData.incrementCoinMstimer();
        AnimationController.CoinRotation(ControllerData.getCoinMstimer());
        ControllerData.setCovertime(ControllerData.getCovertime() - 0.015);
    } else if (!ControllerData.isCoinIntersect() && !ControllerData.isPaused()) {
        ControllerData.setCoinMstimer(0);
        ControllerData.setCovertime(8);
    }
}
  
```

```

    if (ControllerData.getCointime() <= 0)
    {
        ControllerData.setCoinIntersect(false);
        CoinMusic.stop();
        GameMusic.play();
    }

    if (ControllerData.isEndingGame())
    {
        ControllerData.incrementExplosionMstimer();
        AnimationController.Explosion(ControllerData.getExplosionMstimer());
    }
    UIController.helicopterPanel.repaint();
}

```

Per spiegarlo in maniera più efficiente, il codice riportato sopra verrà suddiviso in sezioni:

```
if (ControllerData.isIsRunning()) {...}
```

Tutto il codice riportato all'interno dell'operazione **actionPerformed()** è sottoposto ad una prima clausola *if* che si assicura che il gioco sia in esecuzione.

```

UIController.helicopterPanel.getCoins().tick();
UIController.helicopterPanel.getHelicopter().tick();
UIController.helicopterPanel.getBeamColumn().tick();

```

Prima di tutto il gioco applica il metodo **tick()** (responsabile del movimento) a tutti gli elementi di gioco.

```

if (!ControllerData.isExceedColumn())
    CollisionController.checkBeamExceed();

```

Dopo di che lancia l'operazione per il controllo del superamento di una colonna solo nel caso in cui quella colonna non sia già stata superata

Questo è necessario in quanto essendo il movimento delle colonne discreto e non continuo, non è stato possibile definire il superamento della colonna in corrispondenza dell'istante in cui l'ascissa dell'elicottero supera l'ascissa della colonna; quindi è stato definito nell'istante in cui l'ascissa dell'elicottero è maggiore di quella della colonna e questo genererebbe dei problemi dovuti alla ripetizione dell'istruzione non inserendo la parte di codice appena vista.

```

if (ControllerData.isBeamCollision()) {
    GameController.endGame();
    ControllerData.setBeamCollision(false);
}

```

Questa parte di codice si occupa di eseguire l'istruzione di **endGame()** solo se il valore di *beamCollision* sia impostato a *true*, ovvero solo dopo una collisione.

```

if (!ControllerData.isEndingGame())
    CollisionController.checkCollision();

```

Questa parte di codice si occupa di eseguire la ricerca delle collisioni solo nel caso in cui non ne sia già stata riscontrata una.

```
ControllerData.setScore(UIController.helicopterPanel.beamColumn.getPois());
```

Questa riga di codice si occupa di aggiornare il valore di punteggio visualizzato a schermo.

```
if (ControllerData.isCoinIntersect() && !ControllerData.isPaused()) {
    ControllerData.incrementCoinMstimer();

    AnimationController.CoinRotation(ControllerData.getCoinMstimer());
    ControllerData.setCovertime(ControllerData.getCovertime() - 0.015);
} else if(!ControllerData.isCoinIntersect() &&
!ControllerData.isPaused()) {
    ControllerData.setCoinMstimer(0);
    ControllerData.setCovertime(8);
}
```

Questa parte di codice si occupa della gestione del funzionamento della moneta: nel caso in cui sia avvenuta una collisione con la moneta e il gioco non sia stato messo in pausa incrementa il valore di **coinMsTimer**, richiama la funzione adibita all'animazione passando la variabile appena citata come parametro e diminuisce di 15 millisecondi il **coinTime** (valore che identifica il tempo rimanente allo scadere degli effetti della moneta). Nel caso in cui il gioco non si trovi negli 8 secondi successivi alla collisione con la moneta i valori dei timer vengono resettati.

```
if (ControllerData.getCovertime() <= 0)
{
    ControllerData.setCoinIntersect(false);
    CoinMusic.stop();
    GameMusic.play();
}
```

Questa parte di codice si occupa di riportare il gioco alla normalità appena il valore di **coinTime** raggiunge lo zero.




```
if (ControllerData.isEndingGame())
{
    ControllerData.incrementExplosionMstimer();

    AnimationController.Explosion(ControllerData.getExplosionMstimer());
}
```

Questa parte di codice permette di eseguire l'animazione di esplosione dell'elicottero tramite l'ausilio di un timer (**explosionMsTimer**).

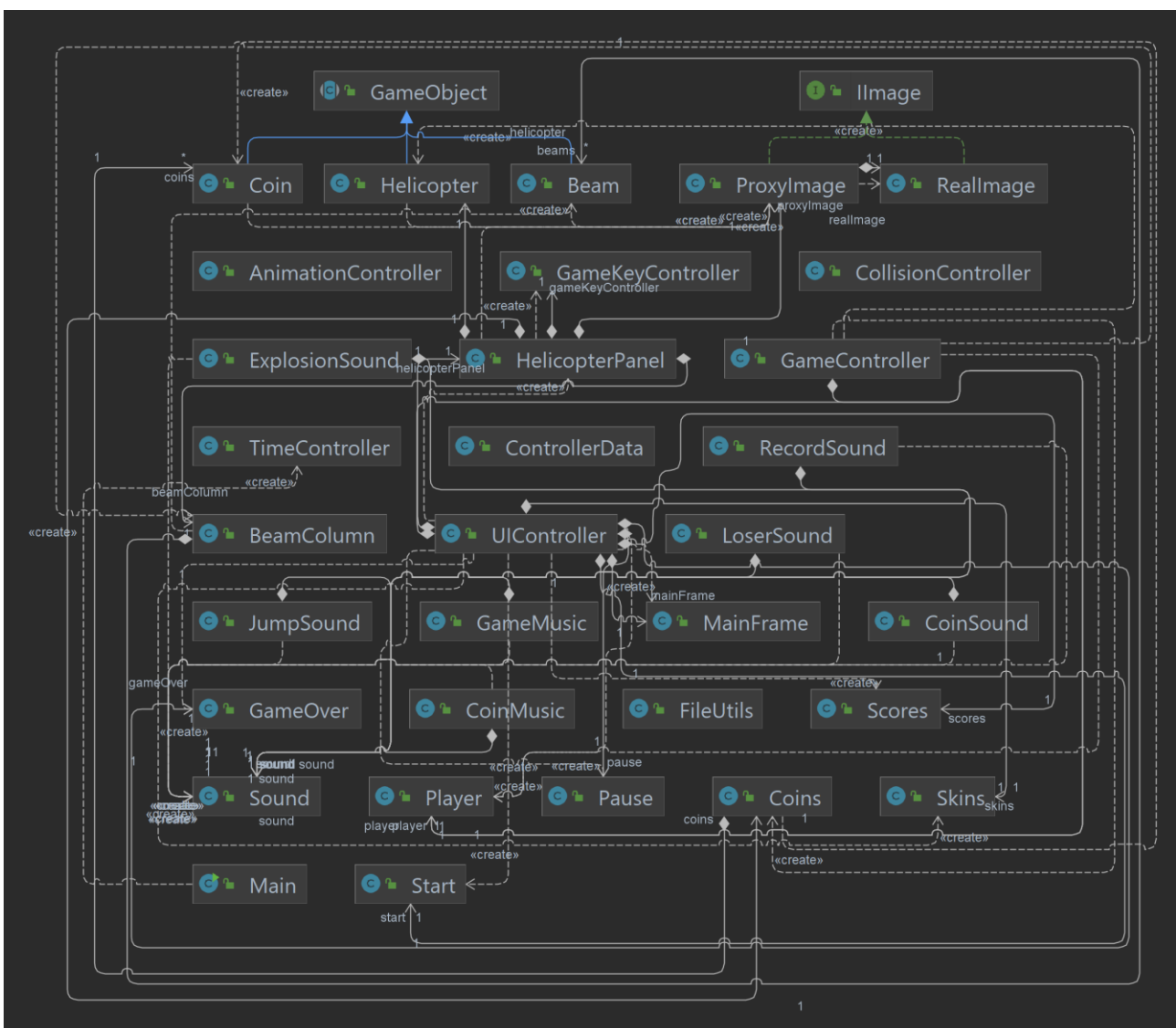
```
UIController.helicopterPanel.repaint();
```

In più ogni 15 millisecondi viene ridisegnata l'interfaccia di gioco in modo da poter apprezzare graficamente le modifiche logiche.

	AnimationController	
	Explosion(int)	void
	CoinRotation(int)	void

La classe contiene solamente le due operazioni che regolano le animazioni presenti all'interno del gioco e sfruttano l'aspetto temporale della variabile che viene loro passata come riferimento per fare in modo di modificare, in modo da ottenere un'animazione il più possibile appagante dal punto di vista grafico, l'icona delle varie componenti; viene generata dunque un'esplosione nel caso dell'elicottero, e una rotazione nel caso della moneta.

3.5 Diagramma UML Completo



1) Gestione del funzionamento dell'animazione d'esplosione

La prima difficoltà con cui ci siamo trovati a dover fare i conti è stata l'animazione dell'esplosione.

In un primo momento abbiamo provato ad implementare l'animazione facendo in modo che il programma stimasse la possibilità della collisione prima dell'effettivo impatto in modo da lasciare il tempo al sistema per l'esecuzione dell'animazione.

Dal punto di vista dello sviluppo del codice non abbiamo trovato grosse difficoltà ma ci siamo accorti che la durata dell'animazione di esplosione, essendo pari a mezzo secondo, imponeva di iniziare l'animazione con eccessivo anticipo.

Tale dinamica risultava tutto sommato accettabile nel caso in cui l'impatto con la trave avvenisse orizzontalmente (fig. 1) ma generava problemi nella valutazione dell'impatto verticale (fig. 2); questo perché risulta frequente che la distanza tra la trave e l'elicottero, quando questo passa nello spazio libero, decrementi molto velocemente e quindi, a volte, l'animazione di esplosione effettuata con il suddetto anticipo veniva inizializzata anche quando la collisione non avveniva realmente.

Per ovviare a questo problema abbiamo deciso di tornare sui nostri passi ed inizializzare l'animazione solo successivamente all'effettiva collisione, fermando il gioco per il tempo necessario.

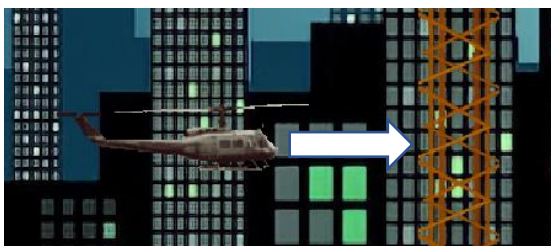


Fig 1



Fig 2

2) Necessità di ridurre ad icona l'interfaccia di gioco

Fin dalle prime fasi di progettazione è emerso un bug per cui, per far funzionare gli input da tastiera nella schermata di gioco, era necessario ridurre ad icona e riportare alle condizioni normali il programma.

Questo Bug abbiamo verificato che si palesa solo nel caso in cui la schermata di gioco non venga caricata per prima ma a seguito di un'altra schermata.

Senza andare a studiare il codice di funzionamento delle operazioni di ridimensionamento della finestra, che avrebbe portato via molto tempo, abbiamo aggiunto al codice due comandi che automatizzino il processo di riduzione ad icona.

```
mainFrame.setState(Frame.ICONIFIED);  
mainFrame.setState(Frame.NORMAL);
```


4. Conclusioni e Sviluppi Futuri

Il progetto è stato svolto senza eccessive difficoltà grazie alla presenza di numerosissime pagine sul web in cui è stato possibile riscontrare buona parte degli errori che abbiamo commesso nel corso della progettazione.

Il programma da noi realizzato rispecchia perfettamente i requisiti che ci eravamo posti.

Un possibile perfezionamento potrebbe contemplare degli sfondi per i JPanel che abbiamo generato attraverso l'utilizzo delle ide NetBeans che avrebbero donato al gioco maggiore coerenza grafica ma che abbiamo omesso per mancanza di tempo.