Corso di laurea in Ingegneria e Scienze Informatiche - Università di Bologna

Corso di Programmazione ad Oggetti - A.A. 2013/2014

Relazione relativa al progetto del corso di Programmazione ad Oggetti Space - Java Space Invaders.

Componenti del gruppo: Mattia Capucci (Matricola 655780), Manuel Bottazzi (Matricola 654226).

1 - Analisi del problema

L'obiettivo del nostro progetto è stata la realizzazione di un'applicazione desktop in linguaggio Java che simulasse il funzionamento del famoso videogioco "Space Invaders" sviluppato da Toshihiro Nishikado nel 1978, in cui il giocatore impersona una navicella che si oppone all'avanzata di una squadra di alieni provenienti dalla parte alta dello schermo di gioco. Ogni alieno può occasionalmente sparare un colpo verso la navicella, se esso è l'ultimo della fila, e viene eliminato quando colpito da un missile della navicella. Il gioco termina quando uno degli alieni raggiunge la navicella oppure quando essa termina le vite a sua disposizione. Se il giocatore riesce ad eliminare tutti gli alieni presenti sullo schermo di gioco il livello è completato ed è possibile proseguire con un nuovo livello per incrementare il proprio punteggio.

In particolare è stato necessario gestire:

- Il funzionamento del gioco e la sua logica, in modo da permettere ad un utente di utilizzarlo.
- Una classifica dei migliori punteggi dei giocatori.
- Una serie di obiettivi sbloccabili eseguendo alcune particolare azioni all'interno del gioco.

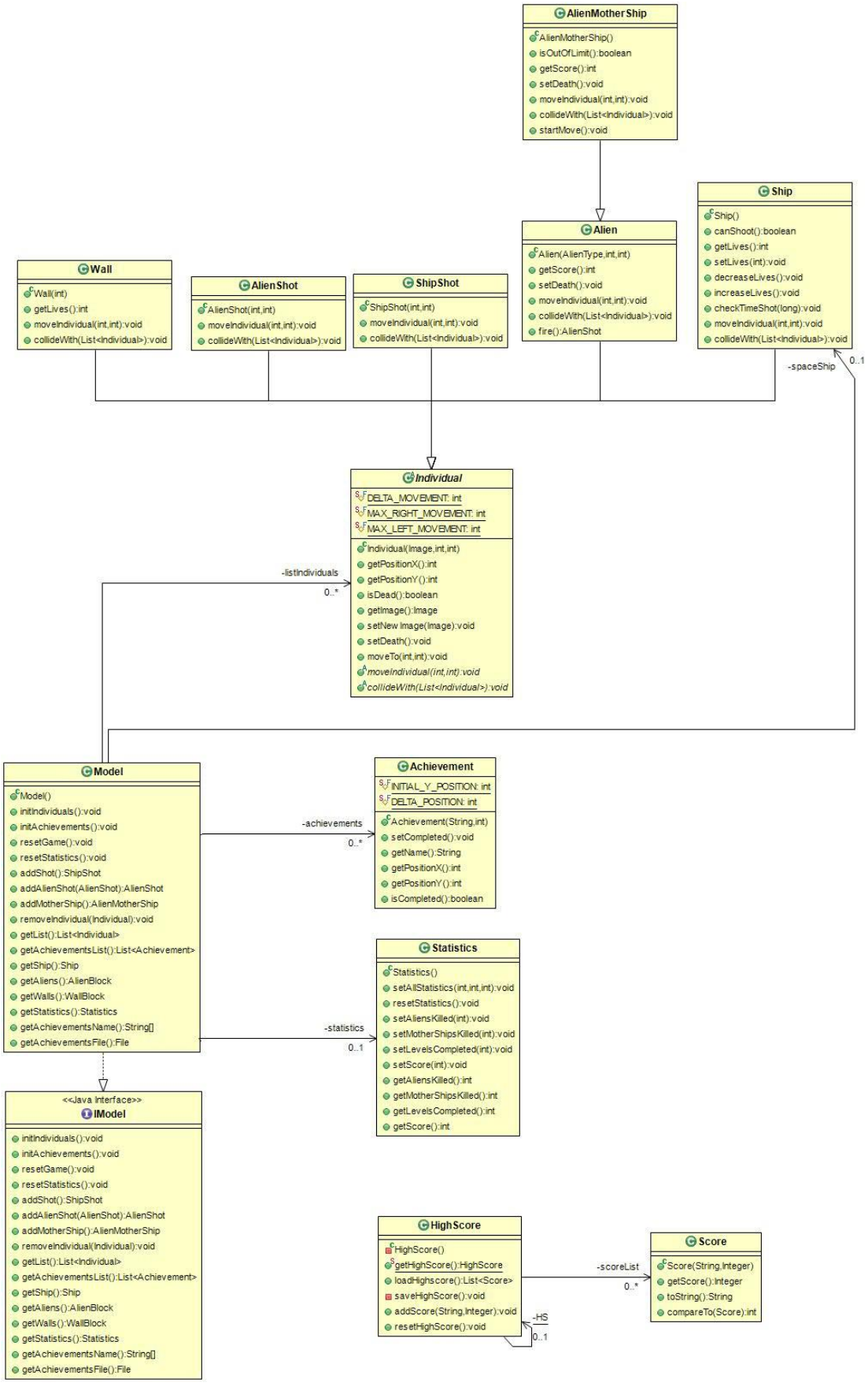
È inoltre necessaria la lettura / scrittura di file per salvare le informazioni generate da queste due ultime funzionalità, mentre per il funzionamento del gioco è stato necessario l'utilizzo di alcuni file multimediali (immagini e suoni) esterni.

2 - Progettazione architetturale

Durante la fase di progettazione abbiamo studiato il funzionamento del gioco e ipotizzato una possibile realizzazione con gli strumenti Java a nostra disposizione, sfruttando ampiamente il pattern Model-View-Controller. Questo ha permesso di mantenere una netta separazione tra i dati utilizzati, gli aspetti grafici e quelli legati alla logica dell'applicazione. In particolare risulta conveniente in una successione di livelli, in quanto occorre semplicemente resettare, con qualche lieve modifica, il *Model.* Inoltre, essendo pensata come applicazione interattiva, con un menù principale e una serie di bottoni nei vari pannelli, risulta comodo avere specifici controller che intercettano gli eventi della view, apportando, se necessario, modifiche al model e cambiare di conseguenza la view.

2.1 - Diagrammi UML dell'applicazione.

Diagramma UML delle classi relative alla parte di Model dell'applicazione:



Principali classi della parte del model:

IModel e **Model** - interfaccia e relativa implementazione dell'insieme dei dati usati dall'applicazione durante lo svolgimento del gioco.

Individual - classe astratta che definisce il generico individuo del gioco, ossia una qualunque entità disegnabile nel pannello di gioco. Da questa estendono quindi tutte le classi che modellano un'entità presente nel gioco (**Alien, Ship, Wall, Shot** e **AlienShot**):

- **Alien** modella un singolo alieno.
- **Ship** modella la navicella controllata dal giocatore.
- **Wall** modella i blocchi che compongono la barriera di protezione tra gli alieni e la navicella.
- **Shot** e **AlienShot** modellano i proiettili sparati rispettivamente dalla navicella e dagli alieni.

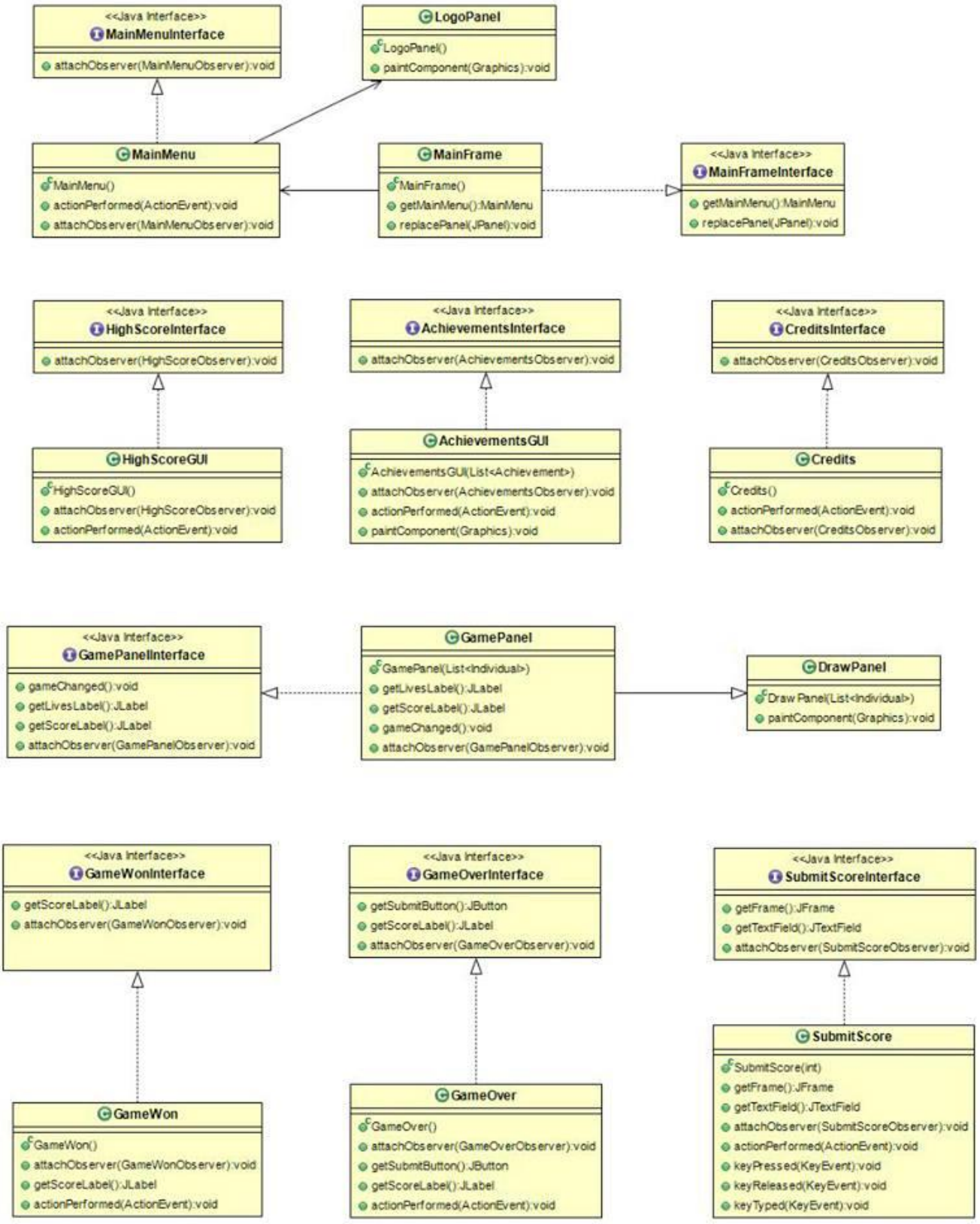
AlienMotherShip estende **Alien** e rappresenta la navicella bonus che passa saltuariamente nella parte alta dello schermo.

HighScore - modella la classifica dei punteggi del gioco e gestisce le singole entry, definite dalla classe **Score,** composte da coppie nome – punteggio.

Achievements - modella il concetto di obiettivo.

Statistics - dati raccolti durante una sessione di gioco e utili per la gestione degli obiettivi.

Diagramma UML delle classi relative alla parte di view dell'applicazione.



Aspetti principali e classi di maggiore importanza della parte di View:

La parte di GUI dell'applicazione è stata creata in modo tale che per cambiare vista sia sufficiente scambiare i vari pannelli all'interno di un unico frame per tutta la durata dell'esecuzione. Questo frame è definito dalla classe **Mainframe**.
Per ogni classe è definita la rispettiva interfaccia.

Di seguito una breve descrizione dei vari pannelli utilizzati nell'applicazione:

- MainMenu e **LogoPanel** rappresentano i due pannelli visualizzati inizialmente all'avvio dell'applicazione.

In particolare **MainMenu** rappresenta il menu principale dell'applicazione e presenta bottoni per raggiungere le principali funzionalità dell'applicazione:

- un bottone per iniziare un nuovo livello di gioco.
- uno per visualizzare la classifica dei punteggi.
- uno per visualizzare gli obiettivi correnti.
- uno per visualizzare i crediti e i ringraziamenti.

LogoPanel è un pannello a sua volta contenuto in **MainMenu** che definisce il logo del gioco.

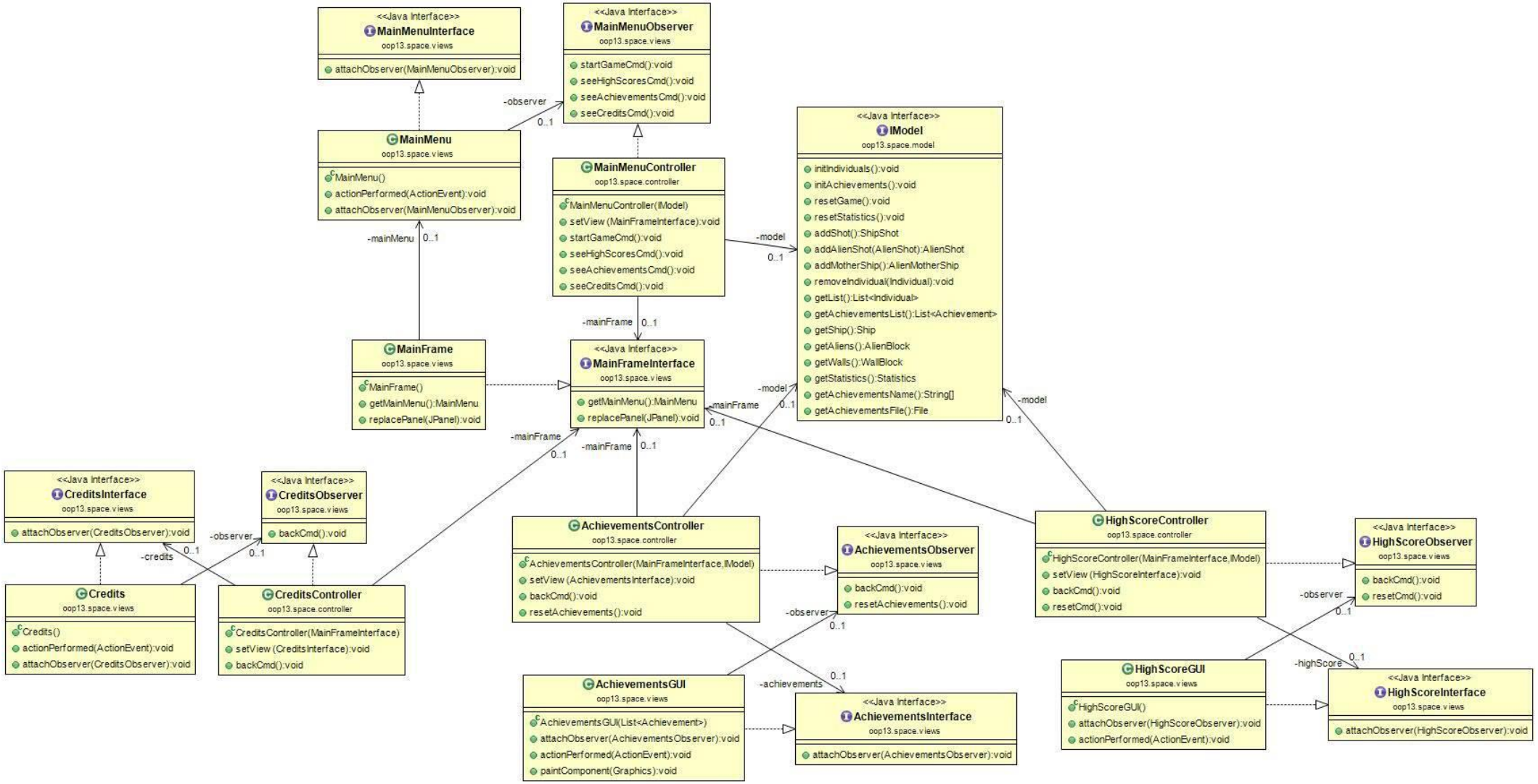
- HighScoreGUI definisce il pannello che visualizza la classifica dei punteggi del gioco.

- Achievements definisce il pannello che visualizza gli obiettivi del gioco e il loro stato attuale.

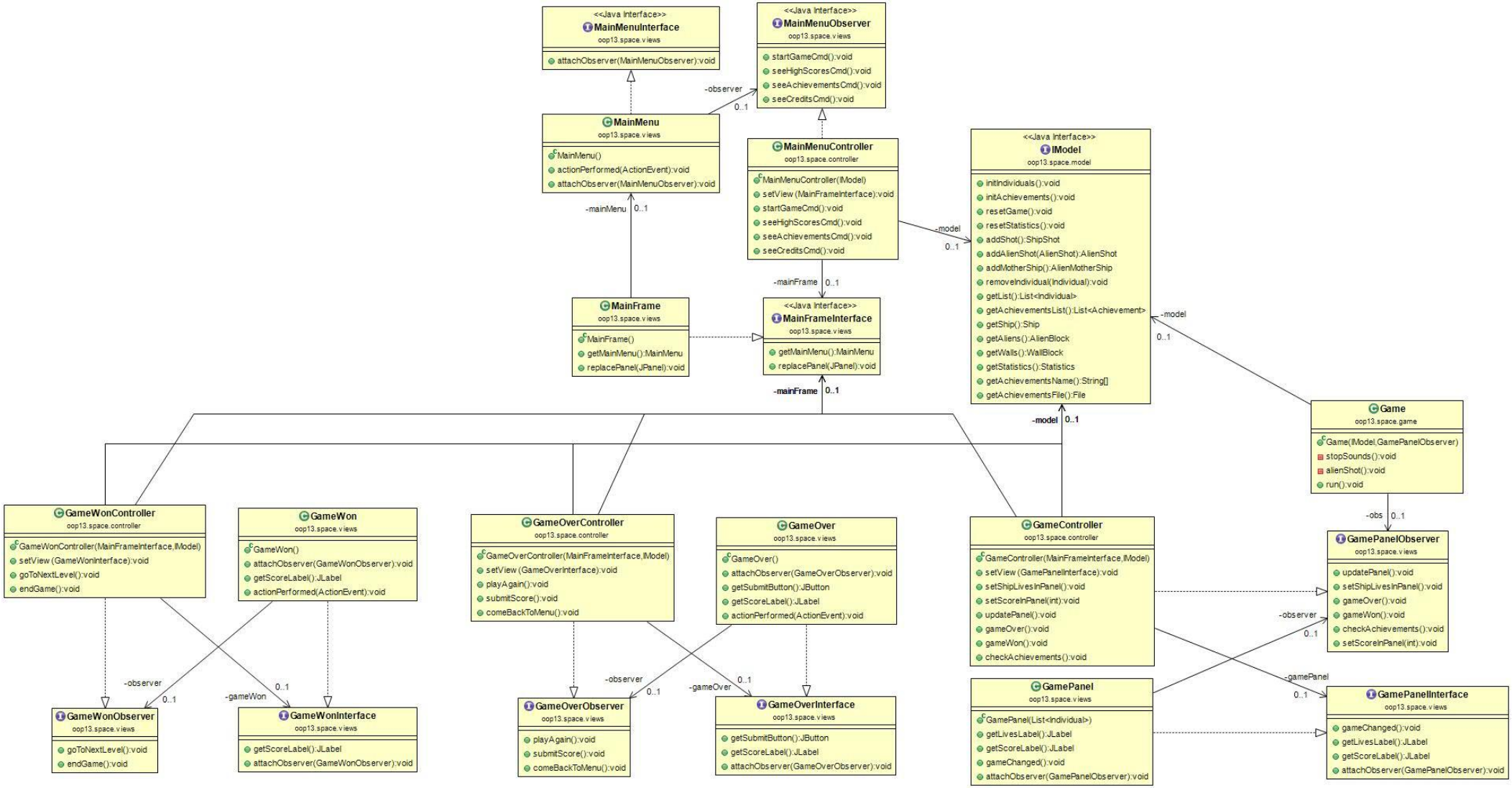
- Credits definisce il pannello che visualizza i crediti.

Esistono inoltre due pannelli che vengono visualizzati quando viene completato un livello (**GameWon**) e quando termina una partita (**GameOver**).

Diagramma UML delle classi relative al Controller dell'applicazione e alla sua interazione con le componenti di model e view. In particolare questo è lo schema relativo al menù principale:



Questo secondo schema è invece relativo ad una nuova partita:



Da questi due diagrammi risulta evidente che l'applicazione è gestita tramite pattern MVC:

I controller sono l'unica parte dell'applicazione che permette all'utente di raggiungere i dati contenuti dal Model, e ad ogni vista è associato un differente controller. In questo modo è il controller stesso a gestire le interazioni dell'utente con l'applicazione e con quella particolare vista.

Da questi diagrammi è inoltre facile notare che per ogni classe di View e per la classe Model è stata definita la relativa interfaccia e come i Controller interagiscano con quest'ultima invece che con l'effettiva implementazione della classe. In questo modo diventerà più facile in futuro un'eventuale rifattorizzazione o aggiunta di una nuova versione di una classe esistente, in quanto sarà sufficiente che la nuova classe implementi l'interfaccia appropriata perché tutto funzioni, invece che dover riscrivere parte dei controller o doverne aggiungere di nuovi.

3 - Organizzazione in Package

L'applicazione è stata suddivisa in Package nella seguente maniera:

- icons:

Contiene i file immagine utilizzati dalle varie classi all'interno dell'applicazione. È stato utilizzato il formato standard .png e .gif per questo tipo di file.

- oop13.space.controller:

Contiene i sorgenti che incapsulano il comportamento dei controller del videogame.
Al suo interno sono presenti le seguenti classi:

- **AchievementsController** - controlla le interazioni con l'interfaccia grafica preposta alla visualizzazione degli obiettivi del gioco.
- **MainMenuController** - controller del menù principale dell'applicazione, gestisce la pressione dei bottoni che si trovano nella relativa GUI.
- **CreditsController** - controlla le interazioni con l'interfaccia grafica preposta alla visualizzazione dei crediti del gioco.
- **GameController** - controlla le interazioni della classe Game con la rispettiva interfaccia grafica (GamePanel) durante lo svolgimento della partita.
- **GameOverController** - controlla le interazioni con l'interfaccia grafica visualizzata al termine di una partita.
- **GameWonController** - controlla le interazioni con l'interfaccia grafica visualizzata quando il giocatore termina con successo un livello di gioco.
- **HighScoreController** - controlla le interazioni con l'interfaccia grafica preposta alla visualizzazione della classifica del gioco.
- **ShipController** - controlla le interazioni del giocatore con la tastiera per il controllo della navicella.
- **SubmitScoreController** - controlla le interazioni con la GUI per l'aggiunta di un nuovo punteggio in classifica.

- oop13.space.exceptions:

Contiene le classi che descrivono le possibili eccezioni personalizzate lanciate dall'applicazione durante l'esecuzione. Al suo interno è presente la seguente classe:

- **EmptyPlayerNameException** - eccezione lanciata dall'applicazione quando il giocatore inserisce una stringa nulla oppure una stringa contenente solo spazi nel form in cui viene richiesto un nome per aggiungere il punteggio alla classifica.

- oop13.space.main:

Contiene unicamente la classe **Main** dell'applicazione.

- oop13.space.game:

Contiene unicamente la classe **Game**, contenente un thread che gestisce la routine del livello di gioco.

- oop13.space.model:

Contiene i sorgenti delle classi necessarie ad implementare la parte di model dell'applicazione.
Al suo interno sono presenti le seguenti classi:

- **Achievement** - classe che modella un singolo obiettivo del gioco.
- **Alien -** classe che modella un singolo alieno.
- **AlienSquad -** classe che modella un'intera colonna di alieni.
- **AlienBlock -** classe che modella l'intero blocco di alieni e i relativi metodi di comportamento.
- **AlienMotherShip -** classe che modella la navicella bonus.
- **AlienType -** classe che modella le diverse tipologie di alieni.
- **AlienShot** e **ShipShot -** classi che modellano i due differenti tipi di colpi sparati dalle diverse entità.
- **HighScore** - classe che modella la classifica dei punteggi del gioco. Essa è stata costruita secondo il Pattern Singleton in modo da garantire l'unicità della sua istanza e la facilità di accesso.
- **Model** e **IModel** - classe e relativa interfaccia che modellano l'insieme dei dati utilizzati nel gioco.
- **Individual** - classe astratta che modella un generico individuo (entità disegnabile) appartenente al gioco.
- **Score** - classe che modella la singola coppia nome-punteggio che sarà poi inserita nella classifica.
- **Ship** - classe che modella la navicella del giocatore.
- **Statistics** - classe che modella i dati relativi alle statistiche della partita.
- **Wall** - classe che modella un singolo blocco della barriera che si trova sopra la navicella del giocatore.
- **WallBlock** - classe che modella la barriera, costituita da 3 singoli **Wall**.

- oop13.space.testing:

Contiene le classi usate durante la fase di testing dell'applicazione. Per alcune di esse è stata utilizzata la libreria esterna JUnit.
Al suo interno sono presenti le seguenti classi :

- **TestCollisionInFrame** - classe usata per testare le collisioni tra il colpo della navicella e i diversi tipi di individui presenti nel gioco.

- **TestCollisionsToShip** - classe usata per testare le collisioni tra i laser alieni e la navicella
- **TestModel** - test JUnit del model dell'applicazione
- **TestPanel** - classe utilizzata per visualizzare un pannello all'interno di un frame.

➢ oop13.space.utilities:

Contiene alcune classi di utilità generale all'interno dell'applicazione, principalmente legate alla gestione dell'I/O da file e all'utilizzo dei file multimediali. Al suo interno sono presenti le seguenti classi:

- **AudioPlayer** - classe definita tramite pattern Singleton che gestisce la riproduzione di file audio con funzionalità di un normale player audio (play, stop, …).
- **ImageLoader** - classe creata con pattern Singleton che gestisce la lettura da file dei file immagine.
- **ListIOManager** - classe che gestisce l'I/O da e su file di informazioni salvate in liste. All'interno dell'applicazione è usata per salvare i dati della classifica e degli obiettivi completati.
- **GameStrings** - classe che contiene le stringhe che vengono utilizzate maggiormente dalle altre classi.

➢ oop13.space.views:

Contiene le interfacce e le relative classi che descrivono l'aspetto grafico e le GUI dell'applicazione.
Al suo interno sono presenti le seguenti classi:

- **AchievementsGUI** e **AchievementsInterface** - classe e relativa interfaccia che definiscono il pannello che mostra la schermata degli obiettivi del gioco.
- **Credits** e **CreditsInterface** - classe e relativa interfaccia che definiscono il pannello che mostra la schermata dei crediti del gioco.
- **DrawPanel** - classe base che disegna su schermo una lista di individui.
- **GameOver** e **GameOverInterface** - classe e relativa interfaccia che definiscono il pannello che mostra la schermata relativa alla terminazione di una partita.
- **GameWon** e **GameWonInterface** - classe e relativa interfaccia che definiscono il pannello che mostra la schermata relativa al completamento di un livello.
- **GamePanel** e **GamePanelInterface** - classe e relativa interfaccia che definiscono il pannello che rappresenta un livello di gioco e le relative informazioni (Vite, punteggio attuale, …)
- **HighScoreGUI** e **HighScoreInterface** - classe e relativa interfaccia che definiscono il pannello che mostra la schermata della classifica dei punteggi del gioco.
- **LogoPanel -** classe che definisce il pannello contenente il logo principale del gioco.
- **MainFrame** e **MainFrameInterface** - classe e relativa interfaccia che definiscono il frame del gioco. Tutto lo svolgimento avviene all'interno di questo frame, in quanto gli eventi portano solo ad una sostituzione dei pannelli interni, mentre questo frame rimane invariato.
- **MainMenu** e **MainMenuInterface** - classe e relativa interfaccia che definiscono il pannello che rappresenta il menù principale del gioco.
- **SubmitScore** e **SubmitScoreInterface** - classe e relativa interfaccia che definiscono il frame per l'aggiunta di un nuovo punteggio in classifica. Questo è l'unico frame che viene generato dall'applicazione in aggiunta a quello principale.

- sounds:

Contiene i file audio utilizzati dalle varie classi all'interno dell'applicazione. Si è scelto il formato standard .wav per questo tipo di file.

4 – Suddivisione del lavoro all'interno del gruppo

Abbiamo scelto di suddividere il lavoro per aree tematiche invece che per package, in modo che ognuno di noi realizzasse una parte di model e di view con i relativi controller piuttosto che uno facesse, per esempio, tutte le GUI e l'altro tutto il Model. In questo modo è stata possibile una più equa divisione della mole di lavoro ed è inoltre stata una possibilità in più per ognuno di noi di applicare quanto studiato durante il corso in maniera più completa e con una visione d'insieme del problema.

In particolare i compiti sono stati divisi come segue:

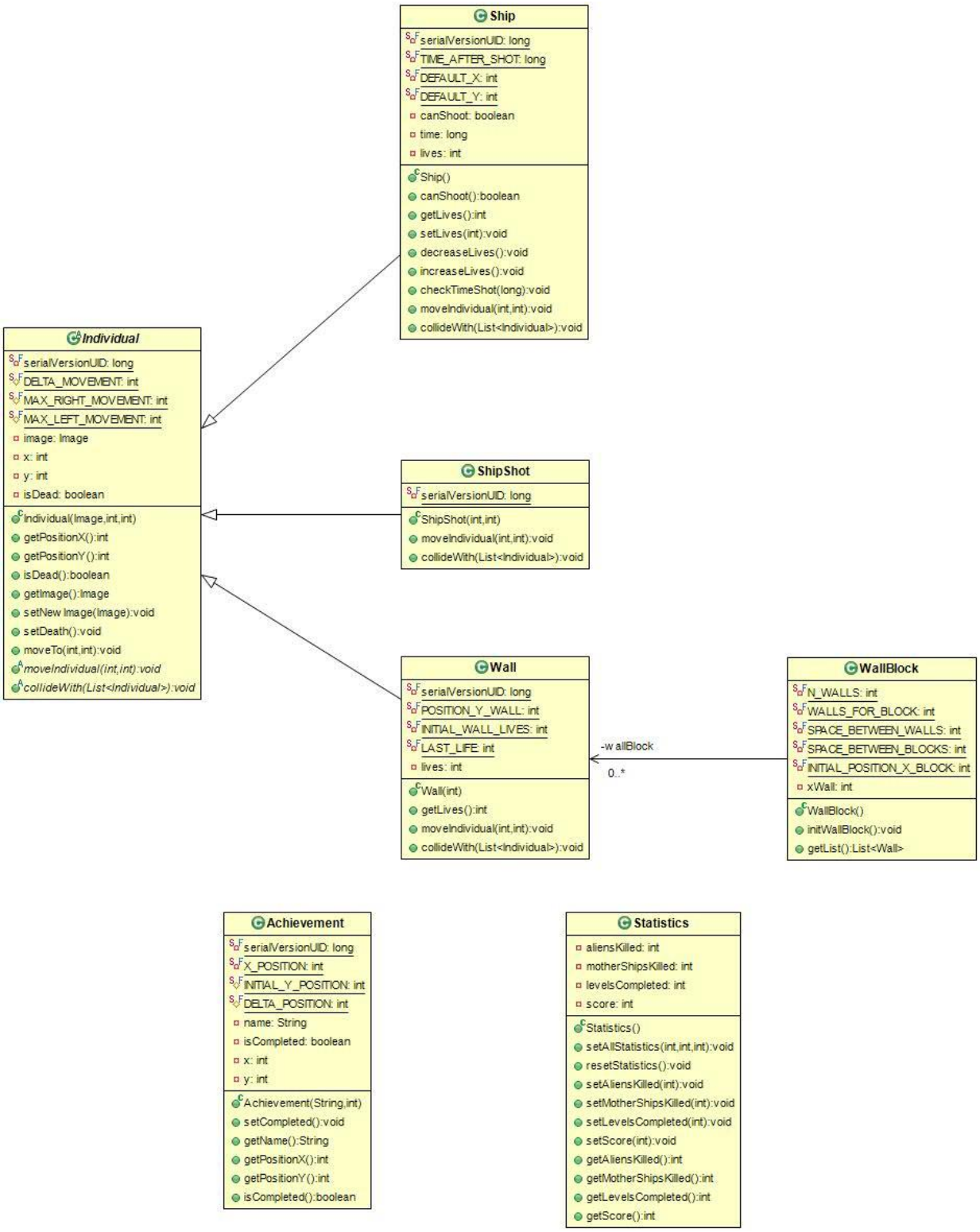
- Mattia Capucci ha realizzato la parte "umana" dell'applicazione, ossia le classi che descrivono la navicella comandata dall'utente, lo sparo lanciato dalla stessa e le barriere difensive nella quale può proteggersi. Inoltre ha realizzato la parte che gestisce gli obiettivi del gioco e il completamento di un livello. Di ogni classe ha realizzato (dove necessario) relativa interfaccia e/o GUI, nonché controller.

- Manuel Bottazzi ha invece realizzato la parte "aliena" del gioco, ossia le classi che gestiscono e modellano il blocco di alieni e quanto ad esso correlato. Inoltre ha realizzato la parte che gestisce la classifica dei punteggi del gioco e la fine di una partita. Anche in questo caso di ogni classe ha realizzato (dove necessario) relativa interfaccia e/o GUI, nonché controller.

5.1 - Progettazione di dettaglio: Parte di Mattia Capucci

Per quanto riguarda il Model, il mio compito è stato quello di modellare il concetto di individuo del gioco e realizzare tre specializzazioni di quest'ultimo: la navicella comandata dall'utente, il colpo sparato dalla stessa e il muro di difesa sotto il quale la navicella può proteggersi dai colpi degli alieni. Infine ho modellato un generico obiettivo del gioco e la classe contenente le varie statistiche di una sessione di game.

Diagramma UML del model:



Individual è una classe astratta che modella un qualsiasi individuo del gioco. È caratterizzata da una posizione X e Y e da un'immagine. Infine un booleano ci informa se l'individuo è ancora vivo o meno. Contiene due metodi astratti: *moveIndividual()* e *collideWith()*. Il primo specifica come l'individuo deve muoversi durante il gioco mentre il secondo definisce le conseguenze che avvengono in seguito ad una collisione dell'individuo rispetto a tutti gli altri presenti nel gioco. Da questa classe estendono **Ship**, **ShipShot** e **Wall**.

Ship modella la navicella del gioco. E' caratterizzata da un numero di vite, inizialmente uguale a 3, e dal tempo in cui viene sparato un colpo. In base a questo valore, il metodo *checkTimeShot()* controlla che il tempo successivo ad uno sparo non sia inferiore a 550 millisecondi (=*TIME_AFTER_SHOT*): in ogni caso viene settato il valore del booleano *canShoot* che definisce la possibilità o meno della navicella di sparare. La navicella può muoversi solo lungo l'asse x, sia a sinistra che a destra ma non può superare i limiti massimi definiti dalla classe Individual che estende. Quando la navicella collide con un colpo lanciato da un alieno, le sue vite si decrementano tramite il metodo *decreaseLives()* e viene riprodotto il suono della navicella colpita. Quando le vite si esauriscono, la navicella è distrutta.

ShipShot modella il colpo lanciato dalla navicella. Può muoversi solo lungo l'asse y e viene distrutto quando raggiunge un valore di y minore o uguale a zero. Appena il colpo viene lanciato, viene riprodotto un suono caratteristico. Quando collide con un individuo, entrambi vengono eliminati.

Wall modella un muro di difesa per la navicella. Non ha movimento e ha due vite. Quando collide con qualsiasi colpo (lanciato o dagli alieni o dalla navicella) le sue vite vengono decrementate e la sua immagine principale viene modificata con un'immagine rappresentate un muro crepato. Quando le sue vite si esauriscono, il muro è distrutto e quindi non è più in grado di difendere la navicella da eventuali colpi alieni. Quando un qualsiasi alieno raggiunge il muro, indipendentemente dal numero delle vite, quest'ultimo viene distrutto.

WallBlock modella blocchi di muri. In particolare, crea tre blocchi ognuno dei quali contiene tre muri difensivi. È formata da una lista contenente tutti i muri, differenziati solamente per la posizione che hanno lungo l'asse delle ascisse.

Achievement modella un obiettivo del gioco. È caratterizzato da un nome che lo identifica, una posizione X e Y affinché possa essere rappresentato nella giusta posizione nell'apposito menù e infine contiene un booleano che ci indica se l'obiettivo è stato completato o meno. Un obiettivo può essere settato come completato tramite il metodo *setCompleted()*.

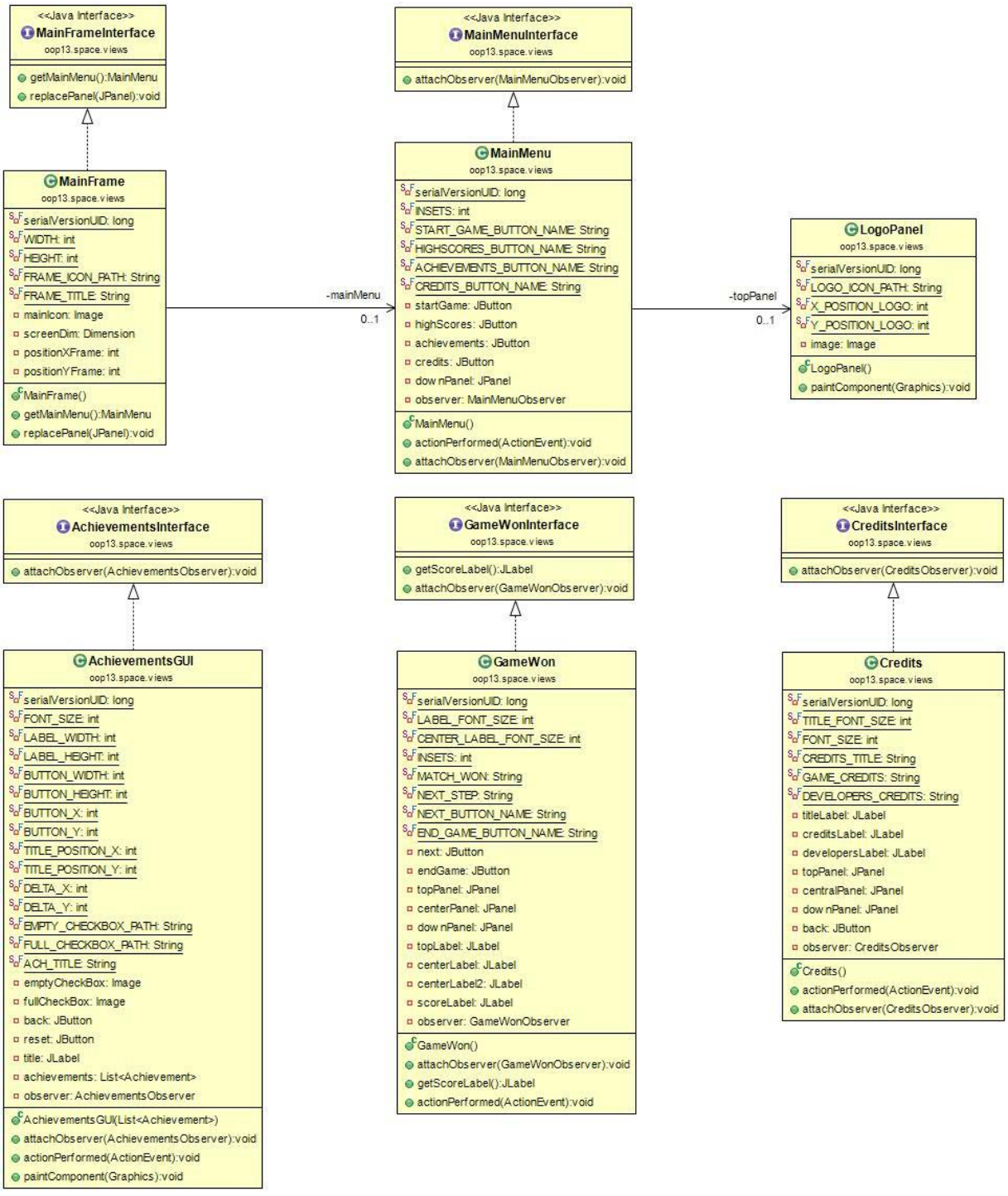
Statistics contiene tutte le statistiche ottenute in una sessione di gioco. Contiene solo campi di tipo intero e precisamente:

- → Numero di alieni uccisi
- → Numero di madri aliene uccise
- → Numero di livelli completati
- → Punteggio

È possibile settare ogni campo singolarmente oppure tutti insieme (eccetto il punteggio, poiché viene aggiornato automaticamente durante il game loop). A gioco terminato, il metodo *resetStatistics()* azzera tutte le statistiche del gioco.

Per quanto riguarda le views, ho modellato le GUI che mostrano il menù principale dell'applicazione, i crediti del gioco, gli obiettivi conquistati e infine la schermata mostrata quando si vince un livello di gioco.

Diagramma UML delle views:



MainFrame modella il frame principale dell'applicazione. La finestra ha dimensioni 600x600 (=*WIDTH* x *HEIGHT*) e attraverso il campo *screenDim*, ovvero le dimensioni dello schermo del pc, si ottengono la posizione X e Y del frame in modo tale da posizionarlo al centro dello schermo. Implementa l'interfaccia **MainFrameInterface** che presenta due metodi: *getMainMenu()* e *replacePanel()*. Il primo restituisce il menù principale dell'applicazione mentre il secondo, preso in ingresso un pannello, lo sostituisce a quello corrente.

All'avvio dell'applicazione nel MainFrame è contenuto il **MainMenu**: esso è formato nella parte superiore da un altro pannello, il **LogoPanel**, che contiene un'immagine (il logo di space invaders) e nella parte inferiore da una serie di 4 bottoni in cascata che permettono rispettivamente di:

→ Cominciare una nuova partita
→ Visualizzare i dieci migliori punteggi
→ Visualizzare gli obiettivi completati e non
→ Visualizzare i crediti del gioco

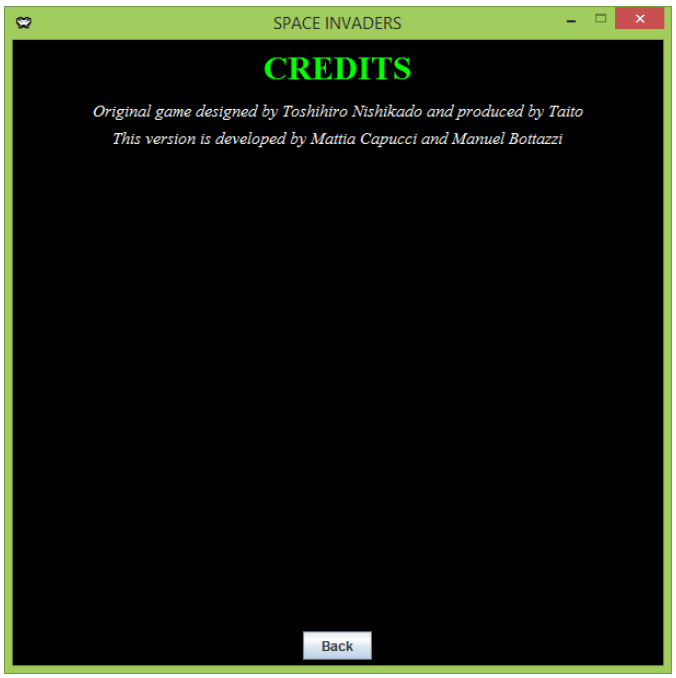
È stato realizzato tramite un *BorderLayout* mentre il pannello inferiore, contenente i bottoni, è realizzato tramite *GridBagLayout*.

AchievementsGUI modella un pannello raffigurante gli obiettivi conquistati o meno durante il gioco. Il suo costruttore prende in ingresso la lista degli obiettivi e li rappresenta nel seguente modo: se sono completati, disegna una casella con la spunta verde, altrimenti una casella vuota. Ricava inoltre dall'obiettivo il suo nome e lo inserisce in una label, posizionandola accanto alla casella. I due bottoni presenti, inoltre, permettono rispettivamente di tornare al menù principale e di resettare gli obiettivi conquistati, in modo tale da permettere al giocatore di mettere nuovamente alla prova le sue abilità nella conquista degli obiettivi. La lista di tali obiettivi è caricata da file (*achievements.data*): se tale file non esiste viene creato automaticamente alla pressione di "*Start Game*" o "*Achievements*" nel MainMenu.

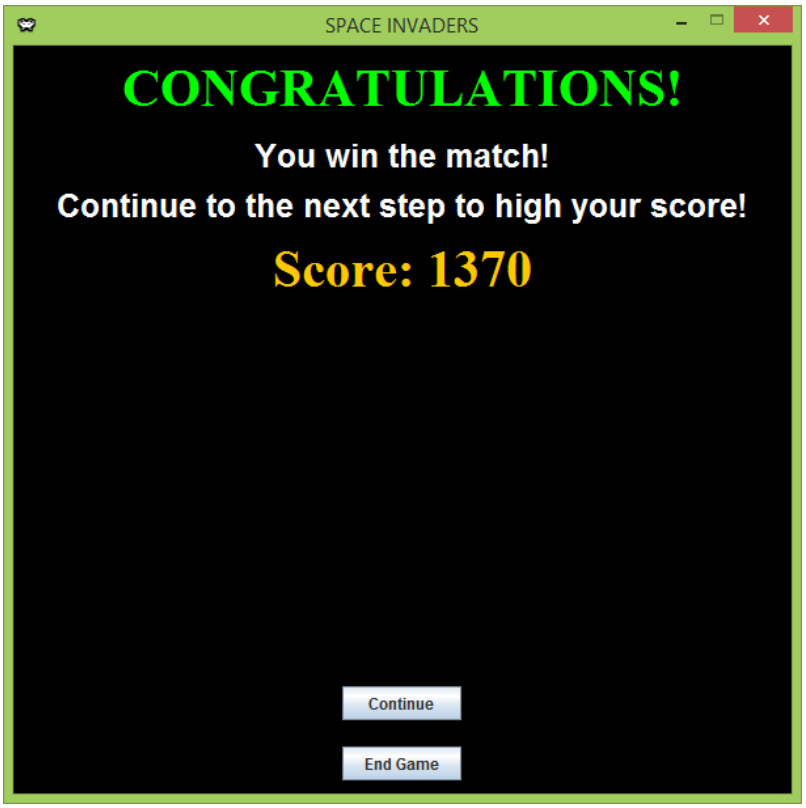
Gli obiettivi presenti nell'applicazione sono i seguenti:

- ➔ 100 ALIENS KILLED: uccidi 100 alieni in una sessione di gioco
- ➔ SURVIVOR: completa il primo livello senza perdere alcuna vita
- ➔ 3 LEVELS COMPLETED: completa 3 livelli di gioco
- ➔ 5 MOTHERSHIPS KILLED: uccidi 5 madri aliene in una sessione di gioco

Credits modella un pannello che mostra i crediti del gioco. È formato da altri tre pannelli, posizionati tramite *BorderLayout*. Il primo contiene il titolo ed è posizionato nella parte superiore del pannello. Seguono poi le label contenenti i crediti del gioco e infine, nella parte inferiore, è posizionato il bottone per tornare al menù principale. Tutti questi pannelli sono realizzati tramite *FlowLayout*.



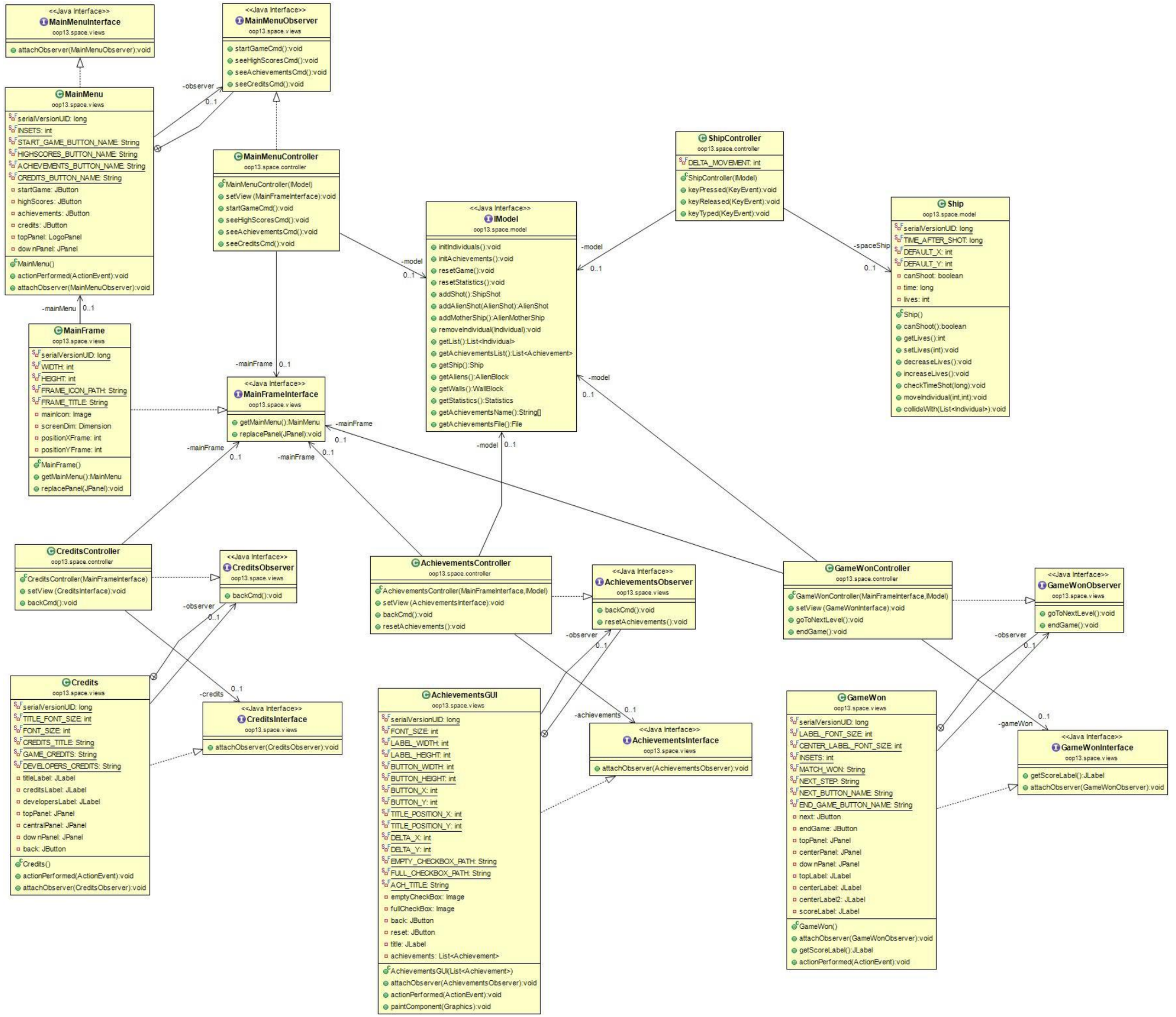
GameWon modella la schermata che appare quando si vince un livello di gioco. È formata da tre pannelli posizionati tramite *BorderLayout*: il primo contiene la label con il titolo ed è posizionato nella parte superiore; segue il secondo che contiene le label che mostrano testo per il giocatore e il punteggio; infine nel terzo pannello sono contenuti due bottoni ed è posizionato nella parte inferiore. Mentre i primi due sono realizzati tramite *FlowLayout*, il terzo è progettato tramite *GridBagLayout* e i due buttoni permettono rispettivamente di continuare la partita (passando al livello successivo) e di terminare la partita.



Implementa l'interfaccia **GameWonInterface** e il metodo *getScoreLabel()* permette di ritornare la label dove viene visualizzato il punteggio dell'utente. Viene utilizzato in particolare dal controller di gioco, che avrà il compito di settare nella label il punteggio ottenuto in una sessione di game.

Tutte le views descritte (fatta eccezione per MainFrame e LogoPanel) implementano la rispettiva interfaccia facendo l'override del metodo *attachObserver()* che permette di collegar loro un osservatore.

Diagramma UML relativo all'interazione tra la parte di view, model e controller:



Come detto in precedenza, il tutto è realizzato tramite il pattern Model-View-Controller. Inoltre è stato utilizzato il pattern **Observer**: ad ogni view viene registrato un osservatore con il compito di gestire un evento generato dalla stessa (per esempio la pressione di un bottone della GUI). Infine i rispettivi controller implementano direttamente le azioni da compiere in risposta ad un evento.

ShipController specifica le azioni della navicella alla pressione di alcuni pulsanti della tastiera. Nello specifico, ShipController implementa *KeyListener* e permette di muovere la navicella a sinistra se viene premuta la freccia sinistra, a destra se viene premuta la freccia destra e infine permette di far lanciare un colpo alla navicella premendo la barra spaziatrice. Agisce sul *Model* modificando la posizione della navicella di un valore *DELTA_MOVEMENT* = 10 o aggiungendo un colpo di navicella alla lista degli individui.

GameWonController gestisce gli eventi a seguito di una vittoria di un livello di gioco. Nello specifico, gestisce la pressione di due bottoni del pannello: se viene premuto il tasto "Continue" il pannello corrente viene sostituito con un nuovo pannello di gioco, mantenendo il punteggio ottenuto nel livello precedente e aggiungendo una vita alla navicella; se viene premuto il tasto "End Game" il pannello corrente viene sostituito con una *GameOverGUI*, *che* permette di uscire dalla sessione di gioco. Agisce sul *Model* inizializzando gli individui di gioco e aggiornando il numero di vite della navicella.

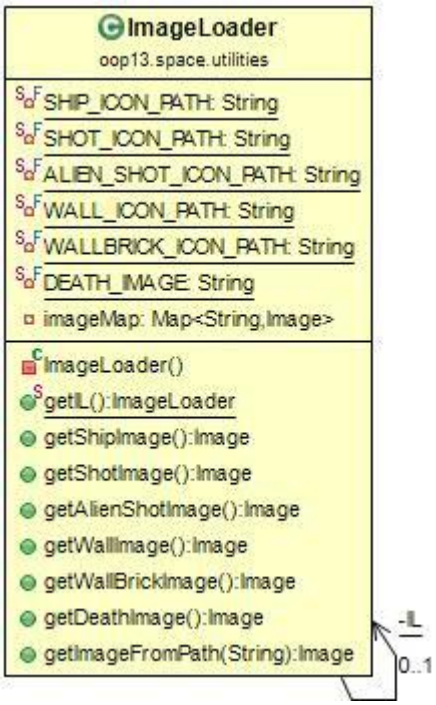
MainMenuController gestisce gli eventi del menù principale. Nello specifico, gestisce la pressione dei quattro bottoni di cui si compone il pannello: se viene premuto "Start Game" il pannello corrente viene sostituito con un pannello di gioco; se viene premuto "HighScores" il pannello corrente viene sostituito con *HighScoreGUI*, mostrando i migliori punteggi ottenuti da ogni giocatore; se viene premuto "Achievements" il pannello corrente viene sostituito con *AchievementsGUI,* mostrando gli obiettivi completati - e non - dal giocatore; se viene premuto "Credits" il pannello viene sostituito con la view *Credits* mostrando i crediti del gioco. Il controller agisce sul *Model* resettando il gioco e le statistiche, inizializzando tutti gli individui e gli obiettivi.

AchievementsController gestisce gli eventi di *AchievementsGUI.* Nello specifico, gestisce la pressione di due bottoni nel pannello: se viene premuto il tasto "Back" il pannello corrente viene sostituito con quello rappresentante il menù principale; se viene premuto il tasto "Reset" vengono resettati gli obiettivi. In *AchievementsGUI* l'effetto è subito visibile: infatti, le spunte verdi che rappresentano gli obiettivi completati vengono sostituiti da un riquadro vuoto. Agisce sul *Model* eliminando il file contenente gli obiettivi completati e sostituendolo con uno vuoto.

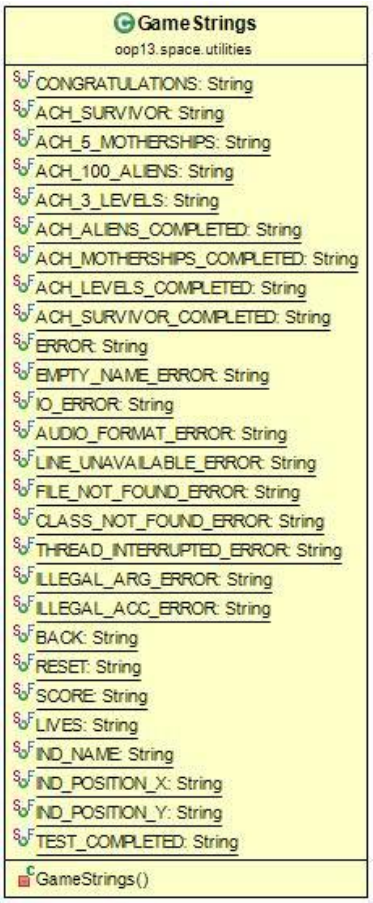
CreditsController gestisce gli eventi di *Credits* view. Nello specifico, gestisce la pressione di un solo bottone nel pannello: se viene premuto il tasto "Back" il pannello corrente viene sostituito con quello rappresentate il menù principale. Non agisce sul *Model*, in quanto la GUI ha il solo compito di mostrare a video alcune informazioni sul gioco e sui suoi creatori.

oop13.space.utilites: ImageLoader e GameStrings

ImageLoader è una classe realizzata tramite **Singleton** pattern, in modo tale che ne venga realizzata un'unica istanza e che sia accessibile da tutte le altre classi. Contiene una mappa che associa una stringa (il path dell'immagine) ad un'immagine. La mappa contiene tutte le immagini presenti in una sessione di gioco (fatta eccezione per le icone degli alieni che vengono gestite dalla classe *AlienType*) e per ognuna di esse un *getter*. Tutte le altre immagini non presenti in ImageLoader vengono gestite dal metodo *getImageFromPath()* che prende in ingresso una stringa rappresentate il path, aggiunge l'immagine associata a quel path alla mappa e la ritorna.



GameStrings è una classe finale che contiene le stringhe che vengono utillizzate maggiormente dalle altre classi. In particolare, abbiamo stringhe riguardati gli obiettivi di gioco, possibili errori, stringhe visualizzabili in un qualsiasi JComponent e stringhe utilizzate nelle classi di test. È formata solo da campi pubblici, statici e finali e da un costruttore privato.



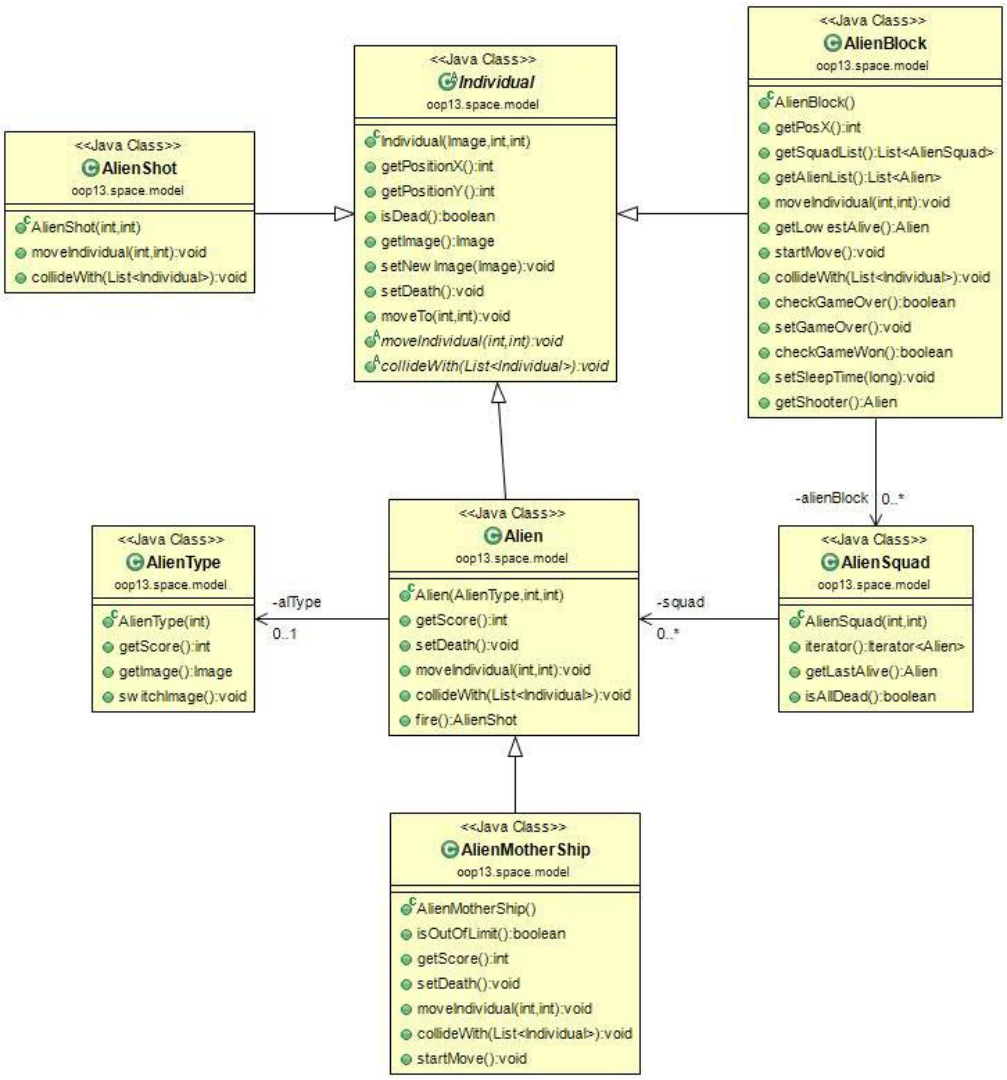
5.2 - Progettazione di dettaglio: Parte di Manuel Bottazzi

1) Model

Per quanto riguarda la parte di Model dell'applicazione il mio compito è stato quello di realizzare le classi relative ai vari tipi di alieni (classi **AlienType**, **Alien, AlienMothership**) e quelle relative a loro aggregazioni più o meno grandi (**AlienSquad** e **AlienBlock**) necessarie poi per gestire la logica di movimento e di azione all'interno del gioco. Ho inoltre realizzato la classe che modella il colpo sparato dagli alieni (**AlienShot**).

Inoltre ho costruito le classi necessarie a gestire la classifica dei punteggi del gioco (classi **Score**, che gestisce un generico punteggio, e **HighScore**).

Diagramma UML della parte di model relativa alle classi "Aliene":



AlienType è una semplice classe che prende come parametro un numero intero tra 1 e 4. Questo permette di definire il tipo di alieno desiderato e quindi il suo set di icone associate e il suo punteggio (che varia in base al tipo di 10 in 10).
Il metodo *switchImage()* permette lo scambio tra le due immagini del set di ogni alieno modificando un

campo booleano interno che identifica lo stato corrente. Questo permette l'alternarsi delle immagini durante il movimento, infatti l'immagine dell'alieno mostrata in fase di gioco proviene da questa classe ed è restituita dal metodo *getImage()*.

Alien definisce una singola entità Alieno ed estende da **Individual**. Essa è caratterizzata da un oggetto della Classe *AlienType*, che ne definisce i parametri caratteristici (immagini e punteggio). Il suo costruttore vuole in aggiunta a questo oggetto anche due interi, che rappresentano rispettivamente le coordinate X e Y in cui l'alieno dovrà essere disegnato.
Un alieno che si muove (tramite *moveIndividual()*) cambia la propria immagine ad ogni movimento, e quando viene colpito da un proiettile della navicella riproduce un suono associato alla sua morte e viene eliminato (grazie al metodo *setDeath()* ereditato da *Individual*).
Il metodo *fire()* è invocato quando l'alieno deve sparare un colpo verso la navicella. Esso genera un oggetto della classe *AlienShot* nella posizione corrente dell'alieno.

AlienShot modella appunto un colpo sparato. Esso si muove solo lungo l'asse Y dalla posizione in cui è generato fino ad una posizione limite (quella impostata è 550) incrementando la propria posizione, muovendosi quindi verso il lato basso dello schermo. Il colpo viene eliminato quando:

1 - Raggiunge la sua posizione limite
2 - Collide con un muro di protezione
3 - Collide con la navicella
4 - Collide con un colpo sparato dalla navicella (entrambi vengono eliminati)

AlienMotherShip è un'estensione di Alien che definisce la navicella bonus che passa nella parte alta dello schermo. Ad essa corrisponde il parametro '4' in *AlienType*. Viene generata in momenti casuali durante il gioco ed attraversa tutto lo schermo di gioco da sinistra a destra nella parte alta, mentre la sua posizione lungo l'asse Y non cambia mai. Quando raggiunge il limite destro dello schermo essa viene eliminata senza conferire alcun punteggio al giocatore. Il suo movimento è gestito da un Thread separato da quello principale del gioco.
Il punteggio associato cambia ogni volta ed è definito casualmente (grazie alla funzione *Math.random()*) con un valore compreso tra 50 e 450.
Per tutto il tempo in cui questa navicella è in gioco essa riproduce un audio caratteristico che termina immediatamente nel momento in cui essa è eliminata.

AlienSquad modella una singola colonna di alieni, composta da 5 alieni delle tre diverse tipologie disponibili. Il metodo *getLastAlive()* restituisce l'alieno ancora vivo più in basso nella colonna.
È stata definita questa entità per semplificare le operazioni di scelta dell'alieno che deve sparare un colpo e di controllo della posizione più bassa dell'intero blocco di alieni.

AlienBlock definisce l'intero blocco di alieni che costituiscono il nemico di una singola partita. È costituita da una lista di *AlienSquad* e da una serie di variabili di controllo. Per uniformità al gioco originale sono state inserite 11 colonne di alieni per blocco.
È stato necessario definire questa classe per permettere di gestire il movimento e il controllo della posizione dell'intero blocco invece che del singolo alieno.

La classe presenta alcuni metodi per il controllo della posizione:

- *getPosX()* restituisce il limite esterno del blocco lungo l'asse X. Questo valore dipenderà dalla direzione di spostamento corrente e la posizione limite destra o sinistra sono calcolate con due metodi privati (*getLeftX* e *getRightX*) che aggiornano due variabili intere della classe (*rightX* e *leftX*).
- *getLowestAlive()* restituisce l'alieno più in basso sull'asse Y. In caso di più alieni nell'ultima riga restituirà il primo.
  Questo ci permette di conoscere la posizione Y più bassa dell'intero blocco, ma si è scelto di ritornare l'intero alieno e non solo la sua posizione su Y per poter effettuare controlli aggiuntivi, come spiegato in seguito.

Il metodo più importante di questa classe è senza dubbio l'implementazione del metodo *moveIndividual().* In esso infatti, oltre al movimento vero e proprio, sono definiti tutti i vincoli di movimento del blocco.

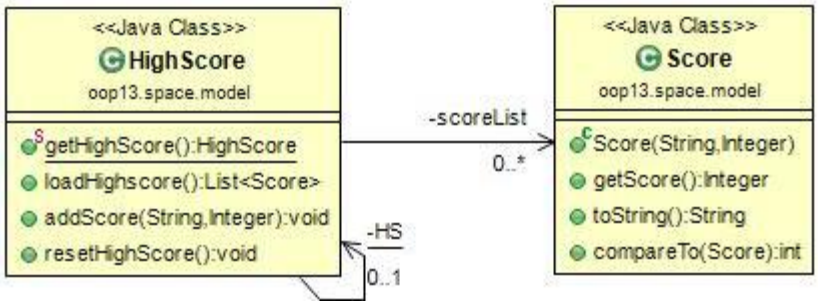
Esso contiene un thread che gestisce il movimento, in modo da poterlo gestire separatamente dal thread principale del gioco, in cui oltre al metodo *run(),* sono definiti due metodi privati che rispettivamente aggiornano il valore della posizione lungo Y del blocco (*updatePosY()*) e controllano la direzione corrente (*checkDirection()*). Il secondo metodo in particolare controlla la posizione lungo X del blocco rispetto al limite che deve raggiungere, e se è in quella posizione inverte la direzione e richiama il primo metodo per far scendere il blocco alla riga successiva.

Il metodo run() contiene un ciclo che mantiene attiva l'esecuzione fintantoché il livello corrente è attivo. All'interno di questo ciclo ho il controllo di posizione e direzione con i metodi citati sopra, e l'effettivo spostamento degli alieni che compongono il blocco nella posizione corretta che essi dovranno assumere. In particolare, se è cambiata la posizione su Y tutti gli alieni del blocco verranno mossi alla riga successiva a quella in cui si trovano, altrimenti verranno mossi al passo successivo a destra o a sinistra a seconda della direzione. Il movimento infatti non è continuo ma a passi di 50 per rimanere fedeli all'originale.
All'interno di questo ciclo avviene anche il controllo sul termine del livello, che termina quando l'ultimo alieno viene ucciso (*levelClear = true;* → livello vinto) o la navetta termina le vite (livello perso, in questo caso viene settato il booleano *gameOver = true***;** con il metodo *setGameOver()*). In entrambi i casi il ciclo termina.

Altri metodi di interesse sono:

- *startMove()* è un metodo che semplicemente incapsula *moveIndividual()* per essere richiamato in maniera semplice dall'esterno. Infatti esso deve essere richiamato una sola volta, e i parametri che gli devono essere passati sono i valori iniziali standard.
- *setSleepTime*() permette di impostare il tempo (in millisecondi) tra un movimento e l'altro del blocco e di aumentare quindi la velocità di discesa e di conseguenza la velocità del gioco. Nella nostra implementazione è stato fatto solo un esempio di questa funzionalità quando rimane un solo alieno vivo, ma sarebbe possibile in implementazioni future aumentare la velocità in maniera graduale in base per esempio al livello corrente o al numero di alieni uccisi.
- *getShooter()* restituisce, tra tutti i possibili candidati, un alieno che può sparare.

Diagramma UML delle classi di model relative ai punteggi.



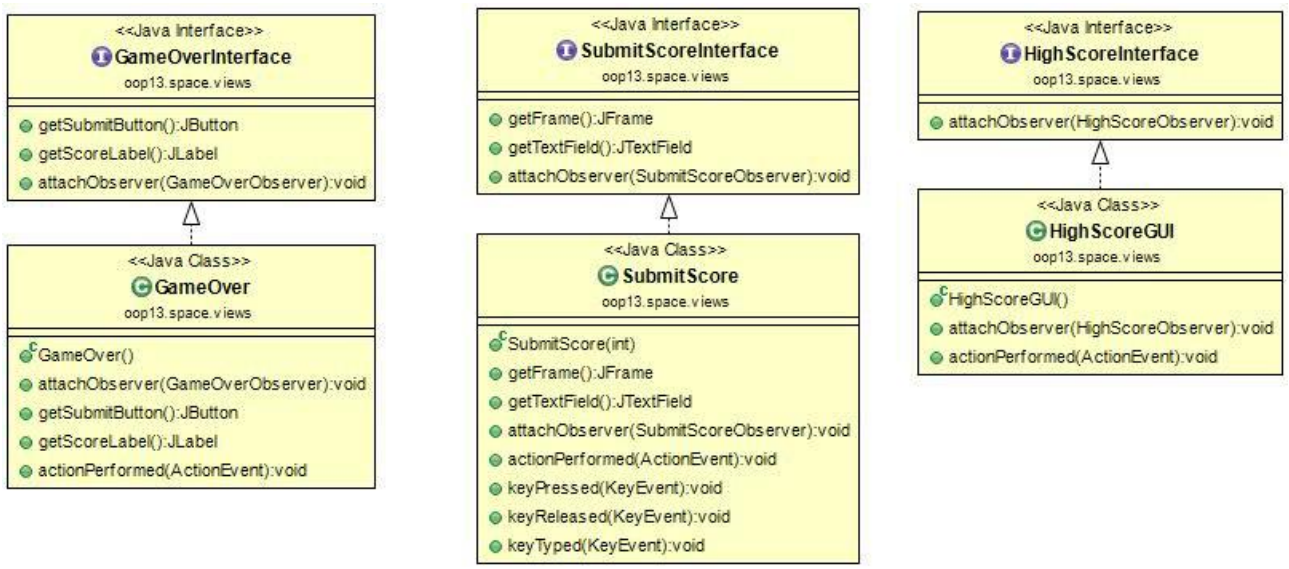
Score è una semplice classe che rappresenta una singola entry nella classifica, ed è composto da una coppia nome – punteggio. Stabilisce anche il criterio di ordinamento che dovranno avere all'interno della classifica (dal più alto al più basso) grazie alla ridefinizione del metodo *compareTo()*.

HighScore definisce la classifica del gioco, ed è stata realizzata tramite pattern *Singleton*, in modo tale da avere un'unica istanza possibile, ma accessibile da tutte le altre classi. Essa legge e scrive i dati su un file esterno con metodi appositi *(loadHighScore()* e *SaveHighScore()*) sfruttando un'istanza della classe *ListIOManager*, descritta in seguito. Permette inoltre di aggiungere un nuovo punteggio alla lista di quelli presenti (grazie al metodo *addScore()*) e di resettare la lista corrente (metodo *resetHighScore()*).

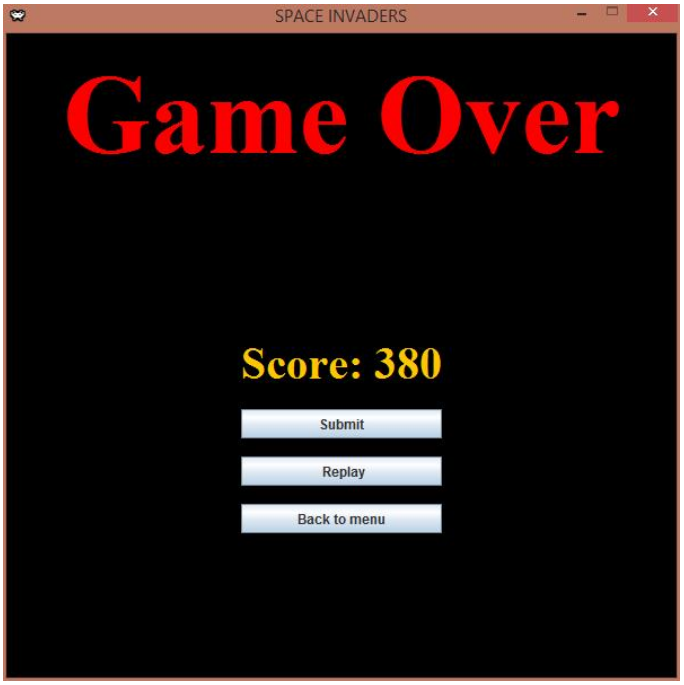
2) <u>View</u>

Ho realizzato le classi e le relative interfacce che definiscono le GUI della schermata di game over, della classifica e la relativa maschera per inserire un nuovo punteggio in classifica.

Diagramma UML delle views.



GameOver è una classe che estende da *JPanel*, ed è posta nel *MainFrame* quando una partita termina con esito negativo (la navicella termina le vite oppure gli alieni raggiungono il livello della navicella).



La classe è costituita da due sotto-pannelli, uno contenente il titolo, visualizzato nella parte superiore, e l'altro contenente una *Jlabel* con il valore del punteggio ottenuto dalla partita appena terminata più i tre bottoni della GUI, con cui è possibile compiere le seguenti operazioni:

- Aggiungere il punteggio corrente alla classifica.
- Ricominciare una nuova partita.
- Tornare al menù principale.

Il sotto-pannello del titolo è stato costruito con un *FlowLayout*, mentre il sotto-pannello centrale con un *GridBagLayout***.** Il tutto è organizzato con un *BorderLayout*.

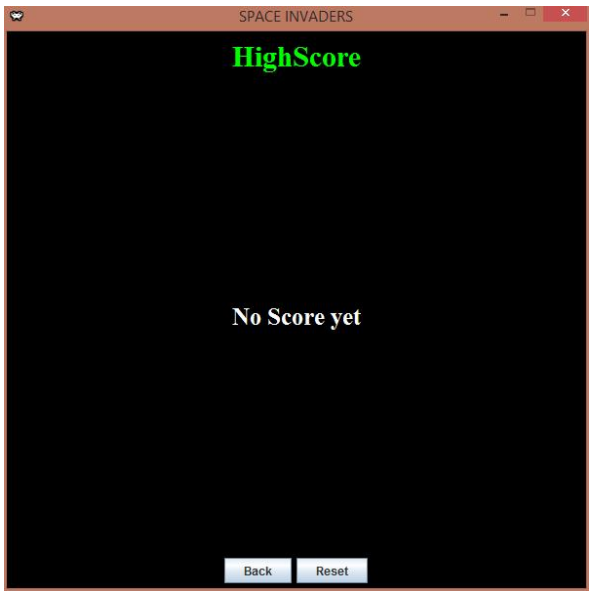
SubmitScore è la maschera che permette di aggiungere un punteggio appena ottenuto alla classifica.
A differenza delle altre GUI questa è definita come un *Frame* separato, in modo che possa comportarsi come una finestra a comparsa nel centro dello schermo.

La classe è costituita di un *JPanel* principale, organizzato con *GridBagLayout*, contenente una *Label* con il titolo e un *JTextField* in cui è possibile inserire il nome del giocatore da associare al punteggio per la

creazione di un nuovo *Score* da aggiungere in classifica.
In aggiunta a questo c'è un secondo *JPanel*, organizzato questa volta con *FlowLayout*, che contiene i due bottoni della GUI. Essi permettono rispettivamente di:

- Salvare il nome appena inserito e di aggiungere di conseguenza il nuovo *Score* alla classifica
- Tornare alla view precedente (chiudo la finestra senza fare nulla). La pressione di questo bottone è analoga alla pressione del pulsante di chiusura in alto a destra.

HighScoreGUI è la classe che definisce la presentazione grafica della classifica dei punteggi del gioco.
Quando la classe viene istanziata essa legge da file (*highscore.data*) la classifica corrente, sotto forma di una lista di *Score*.
Per ognuno di questi *Score* viene generata una *JLabel* contenente le informazioni dei punteggi e viene a sua volta salvata in un opportuna lista.



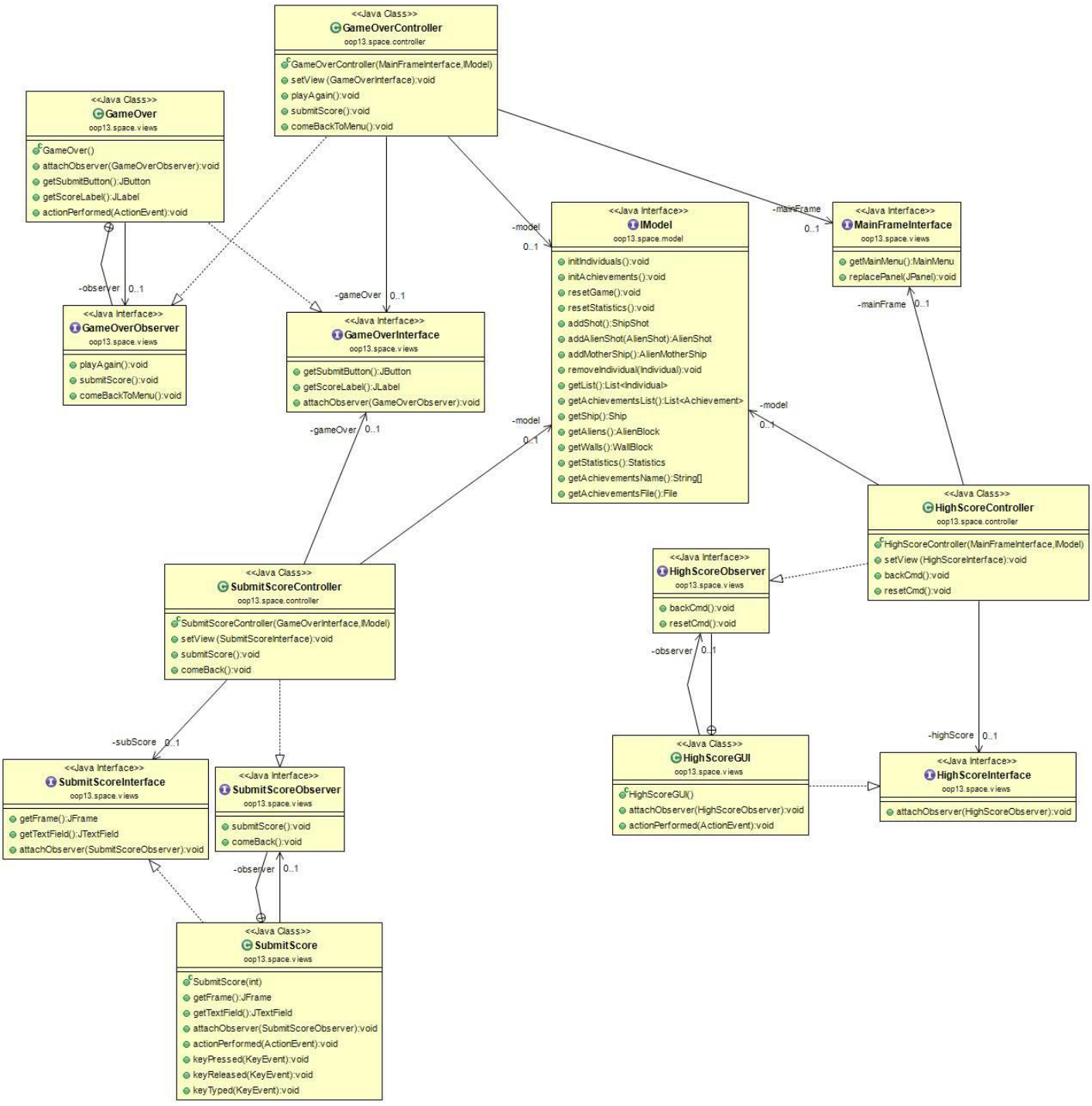
La classe è un'estensione di *JPanel* che viene posta nel *MainFrame* alla pressione del corrispondente bottone del menù principale. Essa è formata da tre sotto-pannelli posizionati tramite *BorderLayout*:
il primo contiene la *JLabel* con il titolo ed è posizionato nella parte superiore.

Nel secondo vengono visualizzate le *JLabel* della lista generata precedentemente, organizzate secondo *GridBagLayout.* Il numero di *Score* visualizzate varia in base a quante sono memorizzate nel file, fino ad un massimo di *MAX_DISPLAYED_SCORE* elementi (10 nella nostro caso). Nel caso nel file fossero memorizzati più di 10 *Score* verranno visualizzati i 10 migliori. Invece, nel caso in cui il file sia ancora vuoto, verrà visualizzata un'apposita *JLabel* che indica che ancora non sono stati inseriti punteggi.

Il terzo e ultimo sotto-pannello (realizzato con *FlowLayout*) contiene i due bottoni, che permettono rispettivamente di tornare al menù principale e di resettare la classifica.

3) Controller

Diagramma UML relativo ai controller e alla loro interazione con view e model.



Come già detto l'applicazione è stata realizzata con uso pervasivo del pattern Model-View-Controller e con l'ausilio del pattern **Observer**: ad ogni view viene registrato un osservatore con il compito di gestire un evento generato dall'interfaccia stessa e controllare le azioni da compiere in risposta ad esso.

GameOverController gestisce gli eventi che si presentano quando una partita termina con una sconfitta del giocatore, ed in particolare controlla la possibile pressione di tre bottoni:

il bottone *"Submit"* genera un nuovo frame di tipo *SubmitScore* (che viene visualizzato a comparsa come una finestra separata) e il relativo controller.
Il pulsante *"Replay"* provoca un reset dei punteggi e delle statistiche (quindi interagisce con il model) e conseguentemente la sostituzione della view corrente con un nuovo livello di gioco.
Il pulsante *"Back to menu"* permette di tornare al menù principale del gioco tramite la sostituzione del pannello corrente.

SubmitScoreController gestisce le interazioni dell'utente con la maschera per l'aggiunta di un punteggio in classifica, ed in particolare controlla la possibile pressione di due bottoni:

Il tasto *"OK"* comporta la creazione di una nuova entry per la classifica, data dal punteggio della partita appena terminata (letta dal *model*) e dal nome utente inserito nella apposita *JTextField* e la conseguente aggiunta del nuovo punteggio alla classifica. La pressione di questo tasto lancia un'eccezione *EmptyNameException* nel caso il nome inserito sia nullo o composto da soli spazi. In questo caso viene visualizzato un messaggio di errore all'utente ed è possibile ripetere l'operazione.
Inoltre per evitare che lo stesso punteggio venga salvato più volte durante una singola partita la pressione di questo bottone, in caso di esito positivo del salvataggio, disabilita il pulsante *"Submit"* della schermata di Game Over.
Il bottone *"Back"* non fa altro che chiudere la finestra corrente, senza che il tasto *"Submit"* della schermata di Game Over sia disabilitato. È analogo alla pressione del normale pulsante di chiusura in alto a destra.

HighScoreController gestisce le interazioni con la schermata della classifica, accessibile dal menù principale, ed in particolare controlla la possibile pressione di due bottoni:

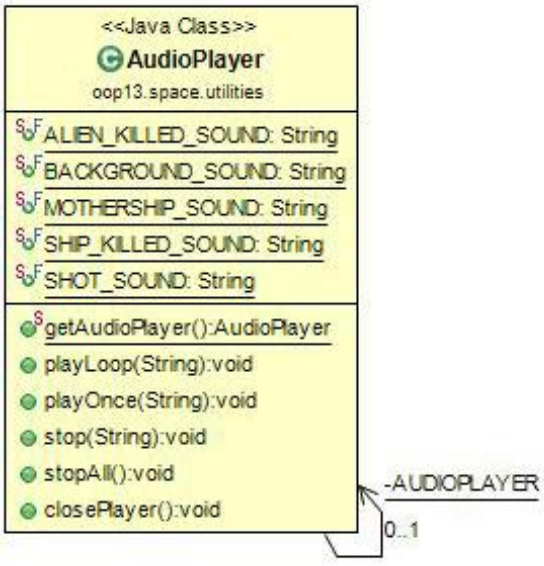
Il tasto "*Back"*, che permette di ritornare al menù principale tramite la sostituzione del pannello corrente, e il tasto *"Reset"*, che permette di azzerare la classifica corrente.

4) Utilities

Ho inoltre realizzato due classi di utilità generale, utilizzate in diversi punti dell'applicazione.

ListIOManager è una semplice classe che permette di leggere e/o scrivere liste di oggetti su file, ed è stata definita in modo generico per permettere l'utilizzo con liste di oggetti di tipi differenti. Il suo costruttore vuole come parametro il *File* su cui dovrà fare I/O. Oltre ai metodi *writeList ()* e *readList()* presenta il metodo *reset()* che permette di azzerare il contenuto del file.

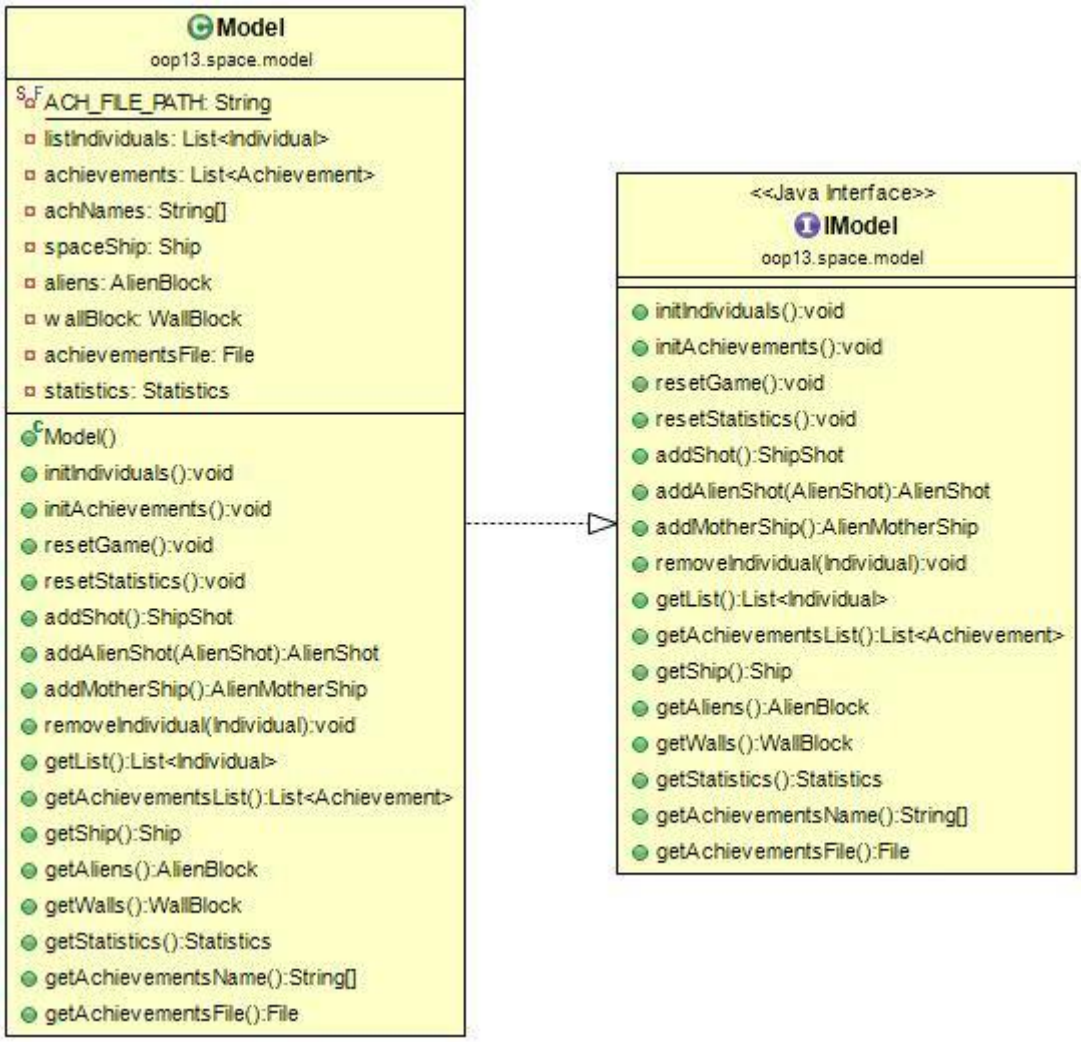
AudioPlayer è una classe che gestisce la parte audio dell'applicazione. È definita con pattern *Singleton* in modo da permettere una gestione unificata dell'audio, raggiungibile in modo statico in tutte le parti dell'applicazione in caso di necessità.
Essa è composta da un insieme di *Clip* (Classi di libreria scelte per riprodurre gli effetti sonori) mappate con delle stringhe, per facilitare l'utilizzo in altre parti dell'applicazione. Questa *Map<String,Clip>* è generata e riempita nella fase di creazione della classe stessa.
I metodi *playOnce()* e *playLoop()* riproducono rispettivamente una



volta e in continuo la *Clip* corrispondente alla stringa passata come parametro ai metodi, mentre il metodo *stop()* ne termina la riproduzione.
Il metodo *StopAll()* ferma tutte le *Clip* in riproduzione, mentre il metodo *ClosePlayer()* termina e chiude tutte le *Clip*.

5.3 - Progettazione di dettaglio: Parte condivisa

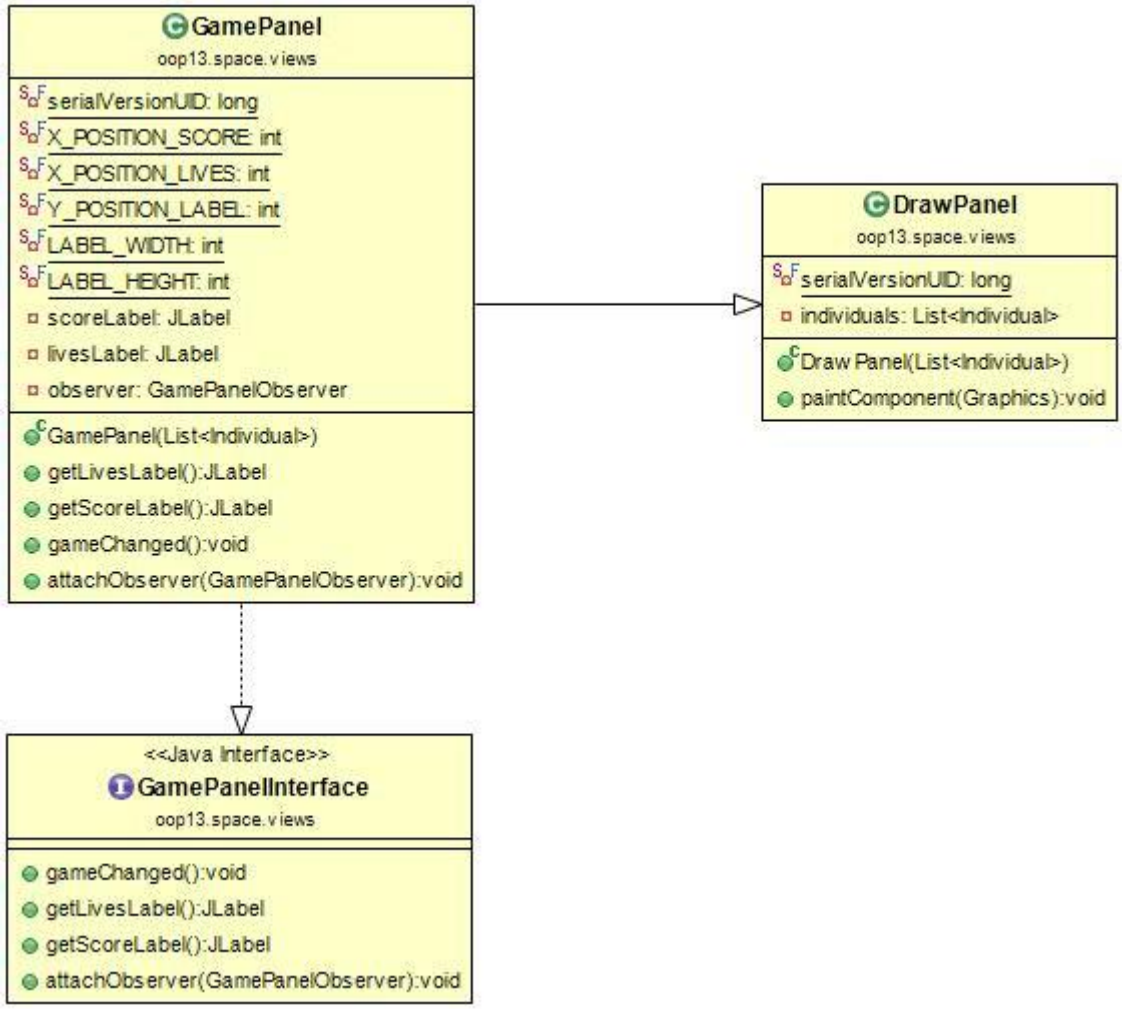
Per quanto riguarda il model, abbiamo lavorato in sinergia nella realizzazione della classe *Model*.



Model implementa l'interfaccia *IModel* e contiene tutti i dati utilizzati dall'applicazione e per ognuno di essi il rispettivo *getter*. In particolare, sono contenuti tutti gli individui del gioco (salvati in un'apposita lista, inizialmente vuota), gli obiettivi e le statistiche. I metodi *initIndividuals()* e *initAchievements()* inizializzano rispettivamente gli individui di gioco e gli obiettivi: nel primo caso vengono aggiunti alla lista degli individui la navicella, le barriere protettive (=*WallBlock*) e gli alieni; nel secondo caso vengono scritti su file gli obiettivi da conquistare, i cui nomi sono reperiti dall'array di stringhe *achNames*. I metodi *resetGame()* e *resetStatistics()* resettano rispettivamente qualsiasi cambiamento avvenuto nel gioco e tutte le statistiche. I metodi *addShot()*, *addAlienShot()* e *addMotherShip()* aggiungono nuovo individui alla lista e rispettivamente il primo aggiunge un colpo di navicella, il secondo aggiunge un laser lanciato dall'alieno e il terzo aggiunge una madre aliena. Infine il metodo *removeIndividual()* prende in input un qualsiasi individuo e lo rimuove dalla lista.

Per quanto riguarda le views, abbiamo lavorato in sinergia alla realizzazione del pannello di gioco.

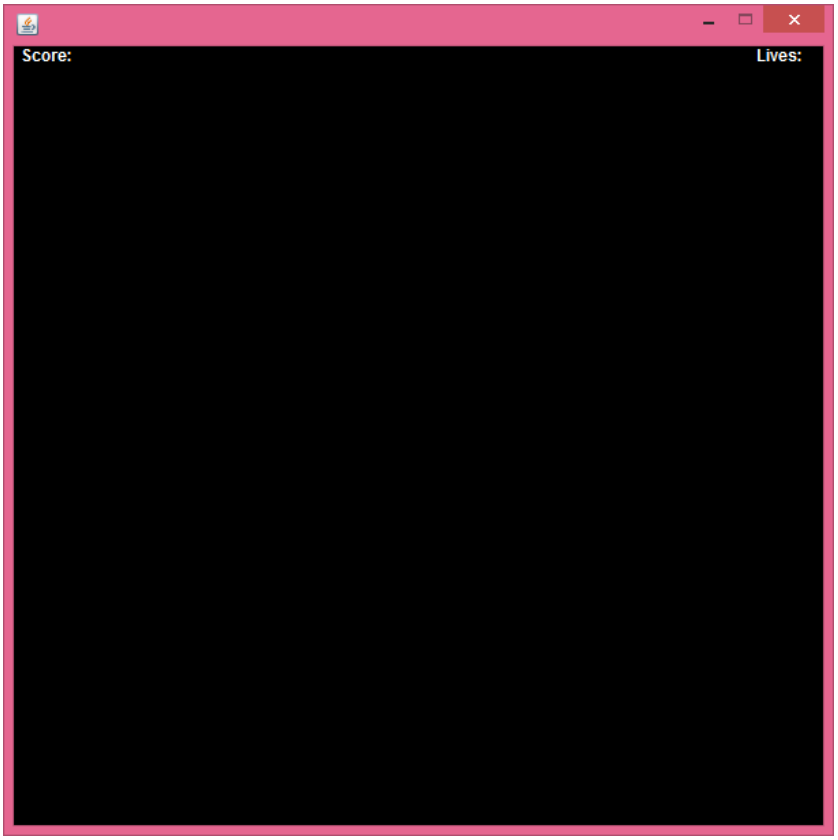
Diagramma UML delle views:



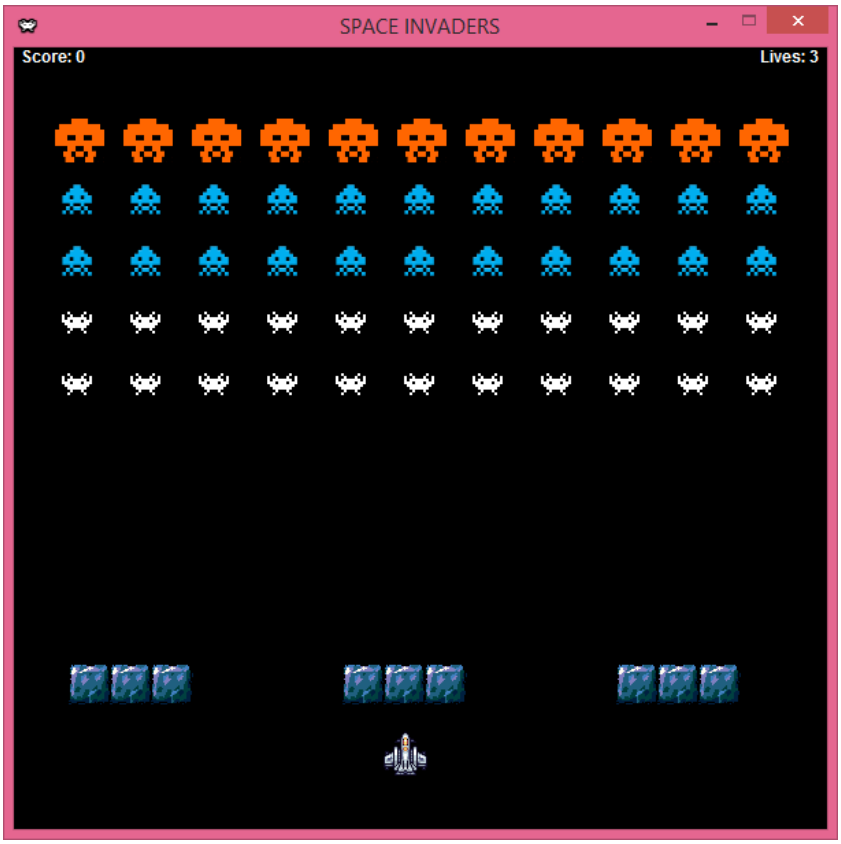
DrawPanel è un semplice pannello che, preso in ingresso una lista di individui, li disegna tramite il metodo *paintComponent()*.

GamePanel implementa l'interfaccia **GamePanelInterface** e estende la classe **DrawPanel**. Contiene due *label* (con i rispettivi *getter*) che visualizzano il punteggio e le vite della navicella e sono posizionate rispettivamente in alto a sinistra e in alto a destra nel pannello. Il metodo *gameChanged()* ha il compito di aggiornare il pannello a seguito di cambiamenti del gioco: lo fa richiamando il metodo *repaint()*.

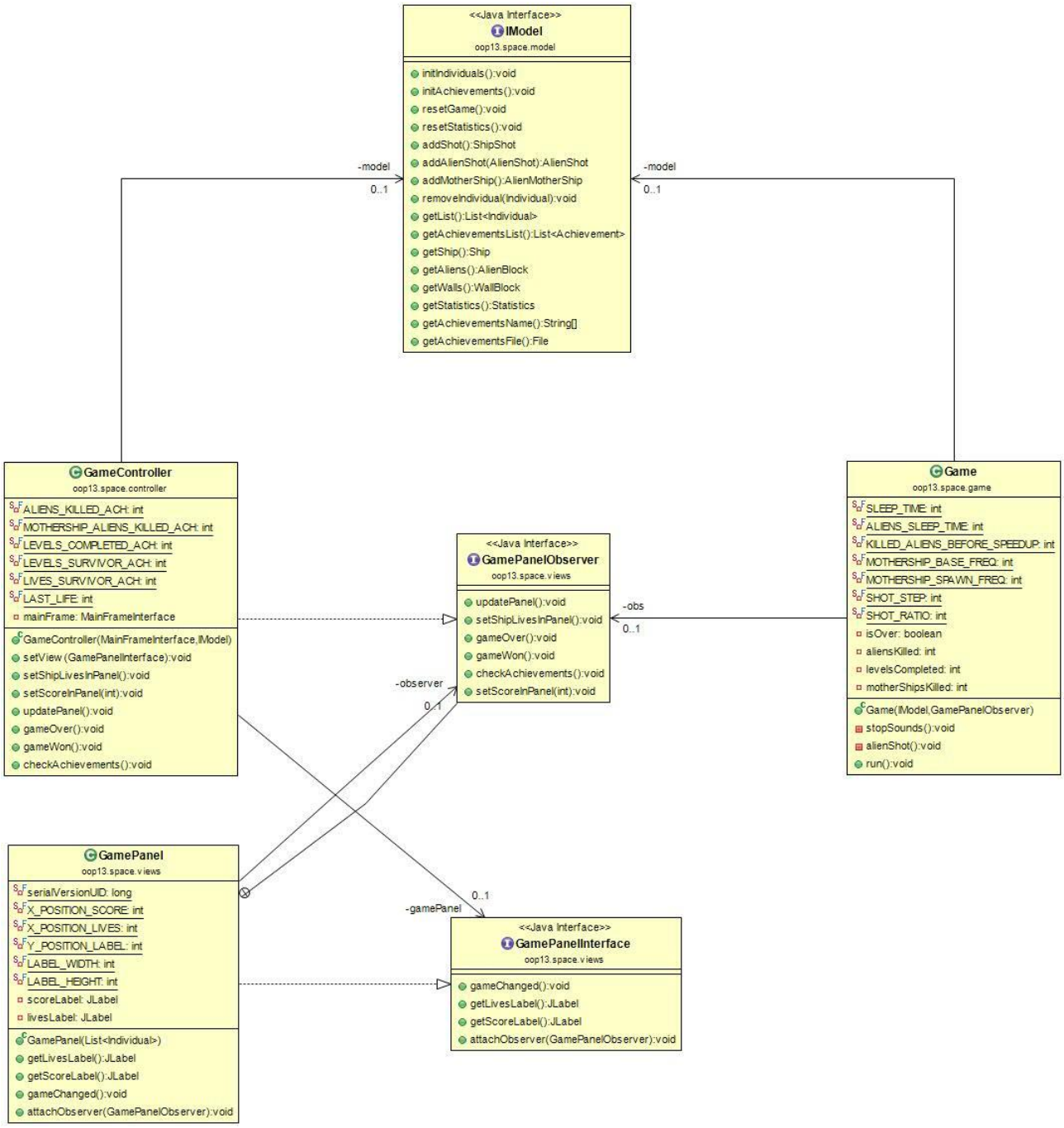
GamePanel senza individui



GamePanel con individui



Infine abbiamo gestito in sinergia la classe *Game* e l'interazione del controller *GameController* sul pannello di gioco.



Game è la classe che gestisce il game-loop. Estende *Thread* e termina solamente quando si vince o si perde un livello. Ha accesso al model e richiama l'observer quando accade uno dei seguenti eventi:

- ➔ Viene colpita la navicella da un laser alieno
- ➔ Viene ucciso un alieno
- ➔ Si ha game over
- ➔ Si vince un livello di gioco

Nel primo caso, *GameController* viene incaricato di modificare la *label* che visualizza a schermo le vite della navicella. Il vecchio valore infatti deve essere decrementato di uno e visualizzato a schermo.

Nel secondo caso, *GameController* viene incaricato di modificare la *label* che visualizza il punteggio. Il vecchio valore infatti deve essere aggiornato sommando il punteggio dell'alieno appena ucciso e mostrarlo a schermo.

Nel terzo caso, *GameController* viene incaricato di modificare il pannello corrente, sostituendolo con *GameOverPanel*, ovvero la schermata visualizzata quando si perde un livello.

Nel quarto caso, *GameController* viene incaricato di modificare il pannello corrente, sostituendolo con un *GameWonPanel*, ovvero la schermata visualizzata quando si vince un livello.

Inoltre, *GameController* aggiorna ciclicamente il pannello di gioco e, poco prima del termine del *thread*, viene richiamato nuovamente per controllare se eventuali obiettivi sono stati raggiunti. Il controllo della conquista degli obiettivi avviene nel seguente modo:

- ➔ Si caricano dal *Model* le varie statistiche
- ➔ Si carica dal *Model* il file contenente gli obiettivi e si carica la lista degli obiettivi
- ➔ Si confrontano le statistiche con i valori richiesti dagli obiettivi
- ➔ Se le statistiche soddisfano i requisiti dell'obiettivo – e quest'ultimo non è ancora stato completato – allora tale obiettivo viene contrassegnato come completato ponendo il booleano *isCompleted* a *true*.
- ➔ Si scrive la nuova lista di obiettivi nel file, sovrascrivendo la precedente

Game contiene due metodi privati: *stopSounds()* e *alienShot()*. Il primo termina tutti i suoni del gioco quando si vince o perde un livello mentre il secondo sceglie un alieno e lo fa sparare.

6 - Testing

Il testing dell'applicazione è avvenuto seguendo le seguenti fasi:

- ➔ Test del model
- ➔ Test dei pannelli
- ➔ Test delle collisioni
- ➔ Alpha test
- ➔ Beta test

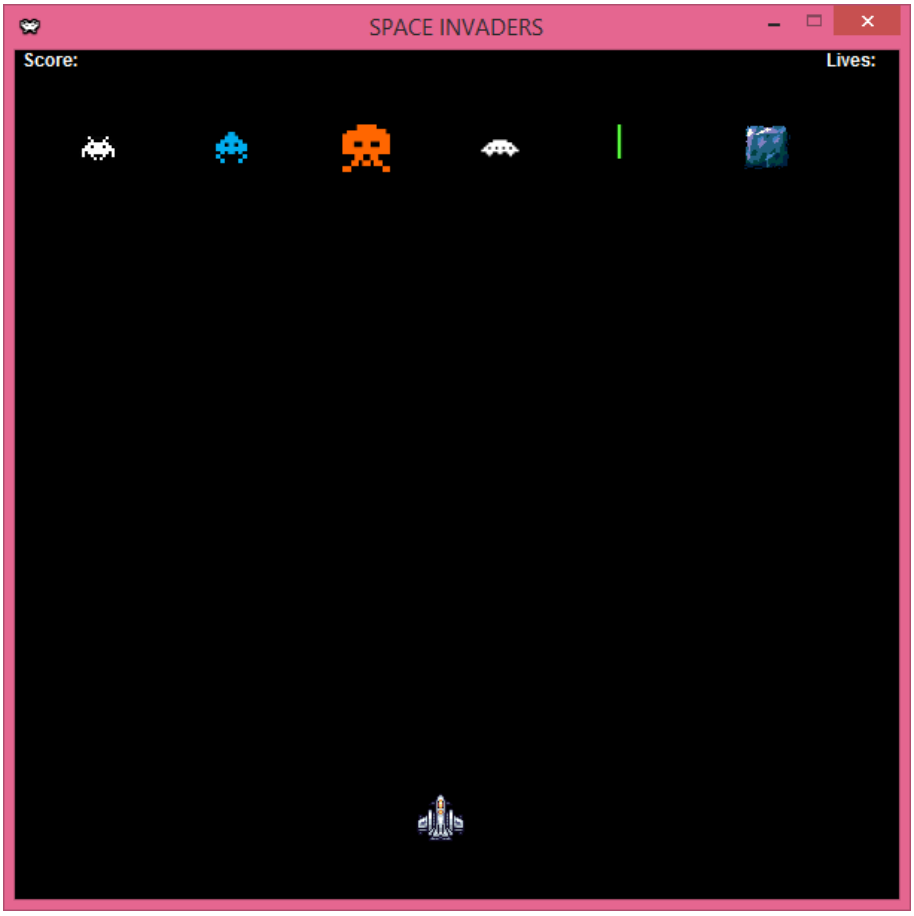
Il testing di **oop13.space.Model** viene effettuato dalla classe di testing **oop13.space.testing.TestModel** e in particolare:

- **testInit:** controlla la corretta inizializzazione degli individui e degli obiettivi di gioco
- **testAddIndividuals:** controlla il corretto inserimento di altri individui nella lista degli individui. In particolare, l'aggiunta del laser alieno, del colpo della navicella e l'aggiunta di una madre aliena.
- **testRemoveIndividuals:** controlla la corretta rimozione di qualsiasi individuo dalla lista degli individui
- **testChangesInModel:** controlla possibili cambiamenti che possono avvenire nel *Model* a seguito di una sessione di gioco. In particolare controlla il corretto movimento della navicella e il test di alcuni

suoi metodi; controlla che le collisioni producano esattamente le conseguenze volute e viene controllato che il punteggio ottenuto dall'uccisione degli alieni sia corretto.

Successivamente per testare a livello pratico le collisioni sono state utilizzate due classi: **oop13.space.testing.TestCollisionsInFrame** e **oop13.space.testing.TestCollisionsToShip**.

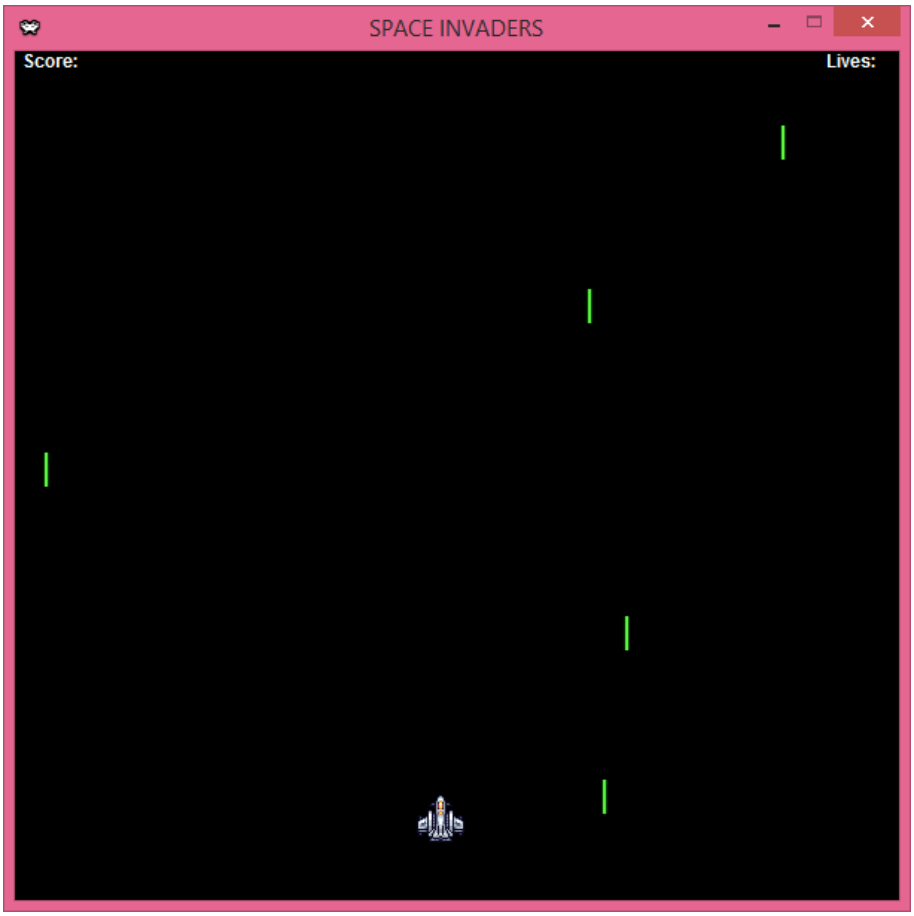
La prima classe permette di testare le collisioni tra il colpo della navicella e tutti gli altri individui. Posiziona su una riga tutti gli individui del gioco e in basso posiziona la navicella, controllata esattamente come avviene in una sessione di gioco. La classe contiene al suo interno un'altra classe che è il *Thread* che gestisce il test delle collisioni: questo termina quando vengono uccisi tutti gli individui. Quando ne viene ucciso uno, su console vengono stampate informazioni utili quali il nome dell'individuo, la sua posizione X e la sua posizione Y, sia dell'individuo colpito sia del colpo della navicella. Questo permette di calibrare al meglio le collisioni tra lo sparo della navicella e tutti gli altri individui.




```
Name: ShipShot
Position X: 230
Position Y: 60

Name: Alien
Position X: 220
Position Y: 50
```

La seconda classe permette di testare le collisioni tra il laser sparato dagli alieni e la navicella. È un thread che termina solo quando la navicella ha esaurito le vite. La navicella è posizionata come al solito nella parte inferiore del pannello mentre nella parte superiore in cascata scendono laser alieni rallentati. Questo permette di posizionare la navicella nella giusta posizione e farla collidere con il laser. Quando viene colpita, vengono stampati su console importanti informazioni quali il nome dell'individuo, la sua posizione X e la sua posizione Y. Questo permette di calibrare al meglio la collisione del laser sulla navicella.




```
Name: ShipTest
Position X: 230
Position Y: 500

Name: AlienShot
Position X: 250
Position Y: 500
```

Per testare i vari pannelli è stata utilizzata la classe **oop13.space.testing.TestPanel**. E' una semplice classe che prende in ingresso un pannello e lo visualizza in un frame.

```java
public class TestPanel {

    private static final int WIDTH = 600;
    private static final int HEIGHT = 600;
    private static final String NO_VALID_PANEL = "Error: insert a valid panel!";

    private JFrame mainFrame;

    public TestPanel(JPanel panel) {
        this.mainFrame = new JFrame();
        this.mainFrame.setSize(WIDTH, HEIGHT);
        this.mainFrame.setResizable(false);
        this.mainFrame.setBackground(Color.BLACK);
        this.mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.mainFrame.add(panel);
        this.mainFrame.setVisible(true);
    }

    public static void main(String[] args) {
        try {
            new TestPanel(null);
        } catch (NullPointerException e) {
            System.err.println(NO_VALID_PANEL);
        }
    }

}
```

Per inserire un pannello, basta sostituire *null* nel *main* con un pannello valido. Questo permette di lavorare su un pannello e visualizzare le proprie modifiche semplicemente lanciando il *main* di *TestPanel*.

Infine il progetto ha subito una fase di **Alpha Test** e di **Beta Test**.

Nella prima fase il progetto è stato testato solamente da noi realizzatori dell'applicazione. Nello specifico:

- ➔ Test della GUI e degli eventi ad essa connessi:
  - controllo della reattività della GUI durante il cambio di pannello.
  - controllo che alla pressione di un bottone si verifichi l'evento corretto.
- ➔ Test di una sessione di gioco:
  - controllo del corretto movimento degli individui di gioco.
  - controllo delle possibili collisioni.
  - controllo dell'aggiornamento del punteggio e delle vite della navicella nelle rispettive *label.*
  - controllo del corretto caricamento dei suoni e delle immagini.
- ➔ Test dei casi limite:
  - controllo che ogni individuo non si sposti oltre i limiti sinistro e destro imposti.
  - controllo che gli alieni, una volta raggiunta la navicella, facciano terminare il gioco.
  - controllo che, uccisa una colonna di alieni, se questa era all'estrema sinistra o all'estrema destra, la colonna di alieni adiacente alla colonna appena uccisa posso raggiungere il limite sinistro/destro specificato in *Individual*.
  - controllo che, se rimane un solo alieno vivo, la sua velocità venga incrementata.
  - controllo che nella classifica vengano mostrati solo i primi 10 punteggi migliori.

➔ Test della corretta scrittura su file:
- controllo che ogni punteggio venga scritto correttamente nella classifica e che quest'ultima sia in ordine decrescente.
- controllo che ogni obiettivo completato venga scritto correttamente e che sia visualizzabile nel pannello specifico.

➔ Test della corretta chiusura dei file:
- controllo che ogni file venga chiuso correttamente.

Nella seconda fase (quella di Beta Test) il JAR eseguibile è stato distribuito ad una cerchia ristretta di persone per il testing finale.

7 - Note finali

Commento alle fasi di sviluppo

Il processo di sviluppo è stato abbastanza lineare, in quanto si aveva un'idea chiara di come dovesse funzionare l'applicazione e una possibile struttura risolutiva.

Ci sono stati dei cambiamenti durante la programmazione dell'applicativo, che hanno permesso di migliorare le prestazioni generali del gioco, di rendere il codice più corretto ed elegante e di correggere alcuni bug riscontrati.

I cambiamenti più significativi sono stati:

→ Aggiunta della classe **ListIOManager**: questa classe generica ha permesso una gestione unificata delle operazioni di I/O, in modo tale che sia *HighScore* sia *GameController* utilizzassero questa classe per modificare le proprie liste: nel primo caso, per aggiornare i punteggi di gioco; nel secondo caso, per controllare l'effettiva conquista di un obiettivo.

→ Aggiunta della classe **GameStrings**: questa classe è stata utilizzata per unificare l'uso delle stringhe, ovvero che classi che usavano le stesse stringhe non le dichiarassero ogni volta. Questo permette una maggiore uniformità nella lettura e scrittura del codice.

→ Aggiunta delle classi *Singleton* **AudioPlayer** e **ImageLoader**: queste hanno permesso un caricamento più efficiente dei file multimediali e una loro migliore gestione.

Questi cambiamenti hanno portato ad una rifattorizzazione (non complessa) di alcune classi, in particolare il *Model*.

Ogni componente del gruppo ha sviluppato individualmente la sua parte, in quanto ad ognuno era stato assegnato una parte di *Model,* una serie di views e i rispettivi controller. Solo poche classi sono state sviluppate in sinergia, in quanto richiedevano parti di in comune. Per esempio, la classe *GameController* utilizza sia la view *GameOver*, sviluppata da Manuel Bottazzi, sia la view *GameWon*, realizzata da Mattia Capucci.

Alcune delle scelte implementative sono state fatte appositamente per permettere un'eventuale successiva espansione ed aggiunta di nuove funzionalità all'applicazione al di fuori del contesto del progetto per il corso. Ad esempio, nella classe *GamePanel* è presente l'observer *GamePanelObserver* che non è utilizzato direttamente, ed è quindi stato contrassegnato come "*unused*". Abbiamo preferito lasciare questo observer per un possibile aggiornamento futuro del gioco, in quanto potrebbe essere utile nell'aggiunta di qualche feature.

Tutto il materiale multimediale è stato reperito su Internet e inoltre:

→ Le immagini hanno subito leggere modifiche tramite il programma di grafica *Gimp*
→ I suoni hanno subito lievi modifiche tramite il programma di audio-editing *Audacity*.

Evaluation del risultato finale

Siamo in generale soddisfatti del risultato ottenuto, anche se alcune delle funzionalità pensate in fase di presentazione del progetto (come ad esempio la modalità di gioco 1 vs 1) non sono state realizzate a causa del limite delle 100 ore, raggiunto da entrambi.

Siamo, però, soddisfatti delle funzionalità realizzate, e del funzionamento complessivo dell'applicazione che rispecchia l'idea iniziale che avevamo di essa.