Space-Survival
Relazione di progetto

Buizo Manuel
Matteini Mattia
Paganelli Alberto

25 giugno 2021

Indice

1 Analisi **3**
1.1 Requisiti 4
1.2 Analisi e modello del dominio 5

2 Design **7**
2.1 Architettura 7
2.1.1 MVC 7
2.1.2 Game Loop 9
2.2 Design dettagliato 10
2.2.1 Buizo Manuel 10
2.2.2 Matteini Mattia 18
2.2.3 Paganelli Alberto 25

3 Sviluppo **32**
3.1 Testing automatizzato 32
3.1.1 Buizo Manuel 32
3.1.2 Matteini Mattia 32
3.1.3 Paganelli Alberto 33
3.2 Metodologia di lavoro 33
3.2.1 Buizo Manuel 34
3.2.2 Matteini Mattia 34
3.2.3 Paganelli Alberto 35
3.3 Note di sviluppo 35
3.3.1 Buizo Manuel 35
3.3.2 Matteini Mattia 35
3.3.3 Paganelli Alberto 36

4 Commenti finali **37**
4.1 Autovalutazione 37
4.1.1 Buizo Manuel 37
4.1.2 Matteini Mattia 37
4.1.3 Paganelli Alberto 38
4.2 Difficoltà incontrate e commenti per i docenti 38

4.2.1 Matteini Mattia 38
4.2.2 Paganelli Alberto 38

A Guida utente **39**
A.1 Avvio del gioco 39
A.2 Menù principale 40
A.3 Settings 41
A.4 Sound 42
A.5 Help 43
A.6 Schermata di gioco 43

Capitolo 1

Analisi

Il gioco SpaceSurvival nasce come rivisitazione del famoso arcade Asteroids, l'obiettivo è quello di creare un gioco semplice ma al tempo stesso stimolante, nel quale non sia banale riuscire a sopravvivere a lungo.
L'utente, che avrà il controllo della navicella, dovrà superare più round possibili prima di perdere tutte le vite che ha a disposizione.
La navicella dovrà avere la possibilità di sparare ai nemici per sconfiggerli e superare così i vari round, inoltre potrà raccogliere dei bonus che si differenzieranno in vita e munizioni speciali.
Durante il gioco nasceranno quindi vari tipi di nemici, alcuni in grado di sparare, altri in grado di inseguire la navicella cercando di scontrarsi con essa. Tra questi ci sarà anche un Boss che apparirà ogni 5 round.
Il tutto sarà reso più difficile dagli immancabili asteroidi tipici dell'arcade, che avranno un ruolo neutrale, potranno quindi danneggiare la navicella ma anche tornare utili in alcune situazioni.
Il gioco avrà una difficoltà incrementale data dal numero dei nemici ed asteroidi che aumenteranno col passare dei round.
La navicella sarà comandata tramite la tastiera dall'utente, e avrà a disposizione un'arma (come ogni altra entità in grado di sparare). Quest'ultima avrà a disposizione proiettili normali, con un caricatore illimitato, mentre nel caso di proiettili speciali le munizioni saranno limitate.
L'utente per migliorare l'esperienza di gioco e sopravvivere più a lungo, dovrà cercare di raccogliere più bonus possibili, che appariranno casualmente durante la sopravvivenza ogni N secondi.
I bonus a differenza delle entità precedenti, saranno oggetti statici, non avranno quindi un movimento.
Non ci sarà un limite massimo di vite e l'utente perderà quando le esaurirà tutte, dopodiché verrà mostrata una schermata di Game Over.
Per quanto riguarda il menù, dovrà avere tre sezioni obbligatorie: Start, Settings e Help. Eventualmente anche un bottone Exit.
Cliccando su start apparirà la schermata di gioco, dalla quale inizierà la sopravvivenza e il

gioco vero e proprio. Cliccando su Settings, l'utente potrà scegliere l'aspetto della navicella mentre cliccando su Sounds potrà regolare i suoni di gioco.
Infine cliccando su Help, l'utente potrà visualizzare i possibili input da tastiera per comandare la navicella.

1.1 Requisiti

Requisiti funzionali

- Le entità potranno muoversi con una certa logica
- Le entità potranno collidere e queste ultime dovranno essere gestite
- Ci dovrà essere un menù iniziale al momento dell'avvio del gioco
- L'utente potrà regolare i suoni sia nel menù sia mentre gioca
- Dal menù si avrà accesso ad altre finestre, tra cui Help e Settings
- L'utente dovrà comandare la navicella tramite tastiera
- Sistema per gli spari
- HUD nella schermata il gioco
- Rendere intuitive le interfacce grafiche
- Bottone e finestra pausa durante il gioco

Requisiti non funzionali

- Gestione ottimale dei thread per rendere il gioco fluido
- Struttura scalabile e facilmente estendibile

1.2 Analisi e modello del dominio

Tutte le entità menzionate avranno dei dettagli in comune, infatti sono tutti oggetti di gioco che appariranno solo durante la sopravvivenza, li potremo identificare quindi come `GameObject`.

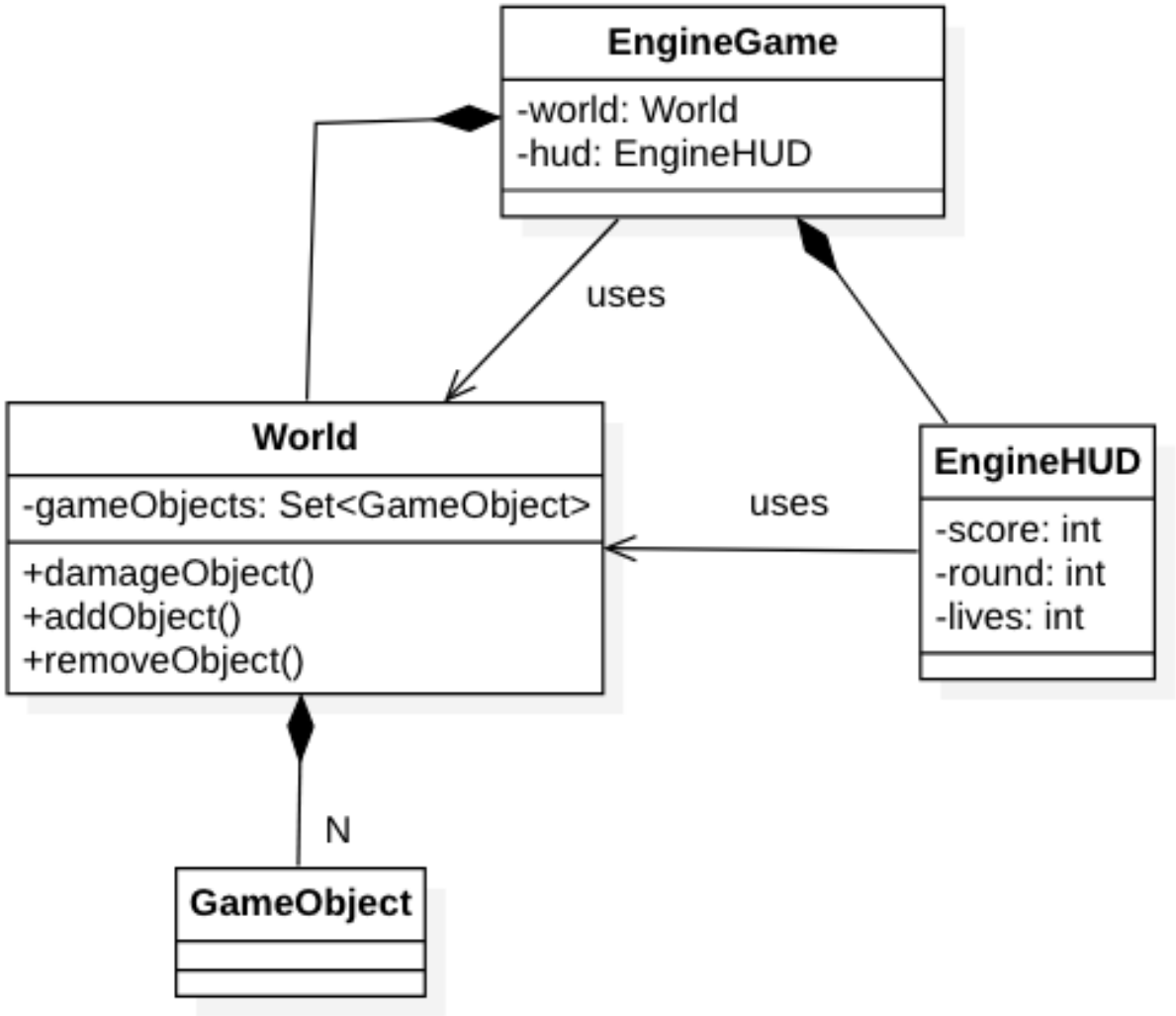
Avremo quindi nello specifico i seguenti oggetti di gioco:

- **SpaceShip**, entità comandata dall'utente
- **Asteroid**, entità neutrale con un velocità fissa
- **ChaseEnemy**, nemico in grado di inseguire la navicella
- **FireEnemy**, nemico in grado di sparare alla navicella, muovendosi casualmente nell'ambiente di gioco
- **Boss**, nemico con la stessa meccanica del precedente, ma con più vita e una logica di sparo migliorata
- **Bullet**, proiettile con velocità fissa che potrà essere sparato da chiunque abbia un'arma

e i bonus disponibili sono:

- **Ammo**: bonus che cambierà il tipo di proiettili sparati dalla `Weapon`
- **Heart**: bonus che rigenererà un determinato quantitativo di vita alla navicella

Tutte queste entità saranno raccolte nel `World`, che rappresenterà anche l'ambiente di gioco, controllerà le collisioni tra i `GameObject` e avrà dei "confini" che lo delimitano. Per quanto riguarda il core dell'applicazione, sarà gestito da un `EngineGame`, il quale avrà il compito di contenere lo stato della partita corrente. Per tenere traccia invece dei dati visualizzati dall'utente sullo schermo durante la partita, ci serviremo dell'`EngineHUD`.



Capitolo 2

Design

2.1 Architettura

2.1.1 MVC

L'architettura di SpaceSurvival è stata impostata seguendo il pattern di architetturale MVC, Model, View e Controller.

Model

Il model è stato implementato in modo da fornire al controller metodi corretti per l'accesso ai dati oltre a gestire come questi vengono influenzati tramite il cambiamento di stati, entità o iterazioni da parte dell'utente finale. Il model inoltre rende corretto e in modo coerente ed è l'entità sulla quale si basa la corretta comunicazione tra il controller e la view.

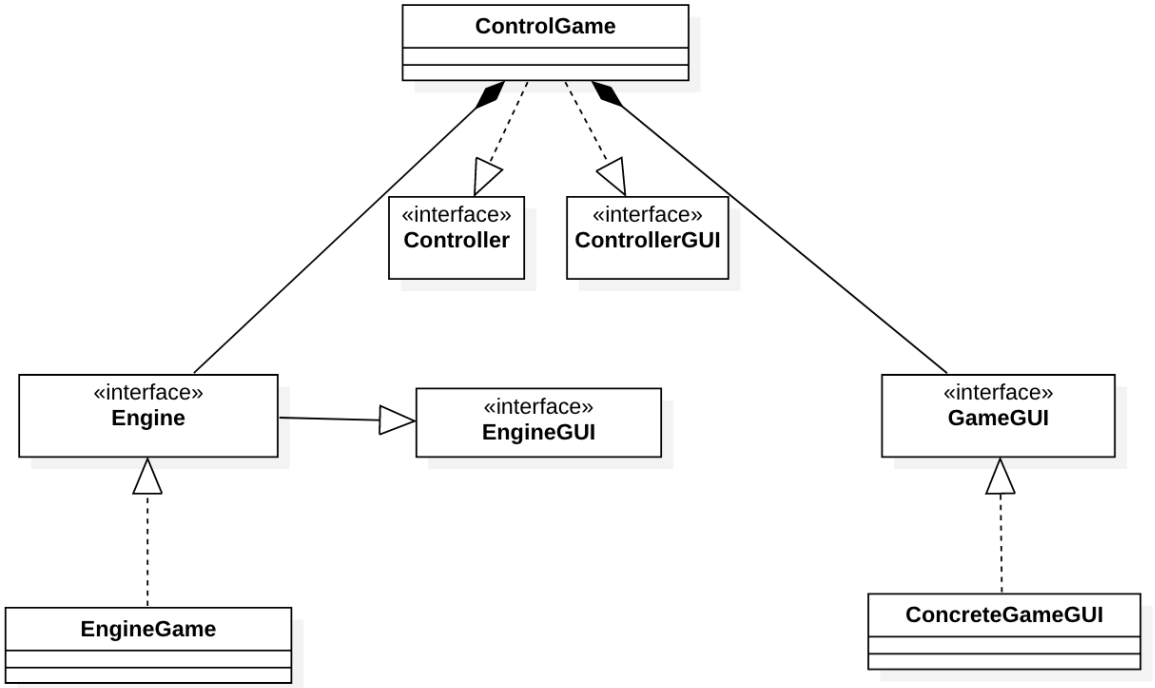
View

La view è composta dalle GUI del gioco e queste ultime rimangono indipendenti dal model e dal controller del gioco. Rimane facilmente implementabile e interscambiabile. Il compito della View è rendere visibili all'utente finale i dati del model in maniera facilmente interpretabile. Inoltre la View ha il fondamentale compito di rendere disponibili al controller quelle che sono le iterazioni dell'utente nelle varie GUI per poi passarle al model del nostro gioco.

Controller

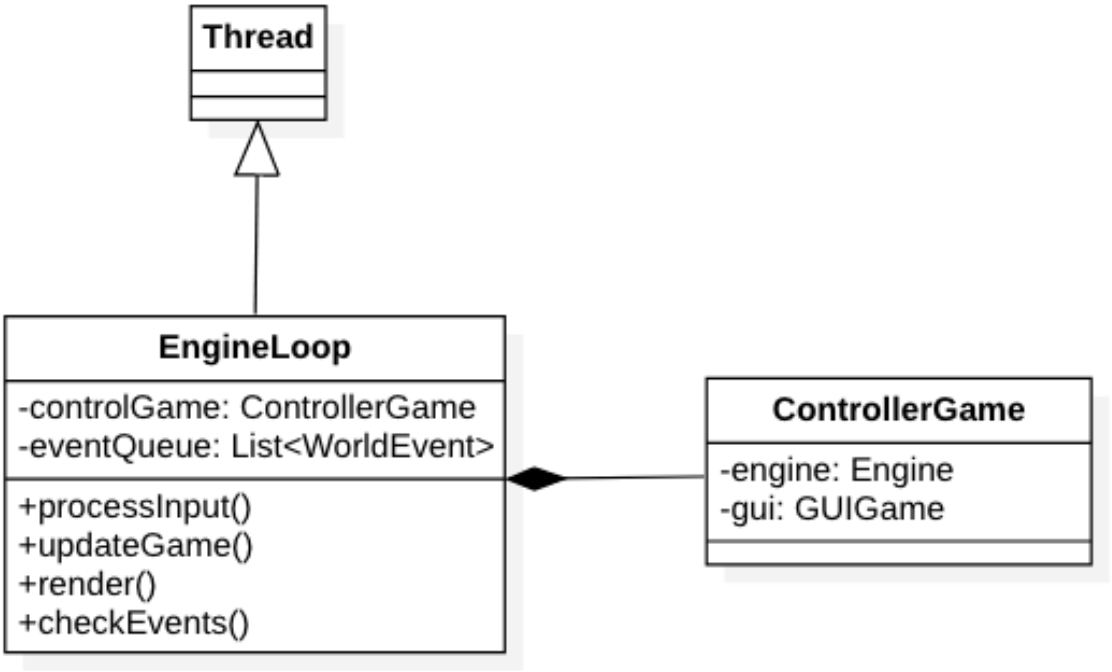
Il controller è quel componente dell'architettura che ha il compito di controllare e gestire l'ordine e la consequenzialità degli eventi.

Il controller inoltre deve gestire i comandi dell'utente letti tramite la View o input da tastiera per poterli trasferire al model e aspettare eventuali risposte e reindirizzandole nuovamente alla view.


ControlGame
«interface»
Controller
«interface»
ControllerGUI
«interface»
Engine
«interface»
EngineGUI
«interface»
GameGUI
EngineGame
ConcreteGameGUI

2.1.2 Game Loop

Inoltre è stato aggiunto un `Game Loop` tramite la classe `EngineLoop`, che serve per mantenere sempre aggiornati gli elementi di gioco. Il `Game Loop` esegue sempre le stesse operazioni, ovvero processa gli input, modifica i dati, e in ultimo luogo aggiorna la grafica.
Questo ciclo tradizionale del `Game Loop`, è molto importante per mantenere la consequenzialità e garantire consistenza.
Nell'`EngineLoop` gestiamo anche la serie di eventi che verranno mandati dai vari `GameObject`.
`EngineLoop` estende un `Thread` che verrà fatto partire al momento del lancio dell'applicazione.

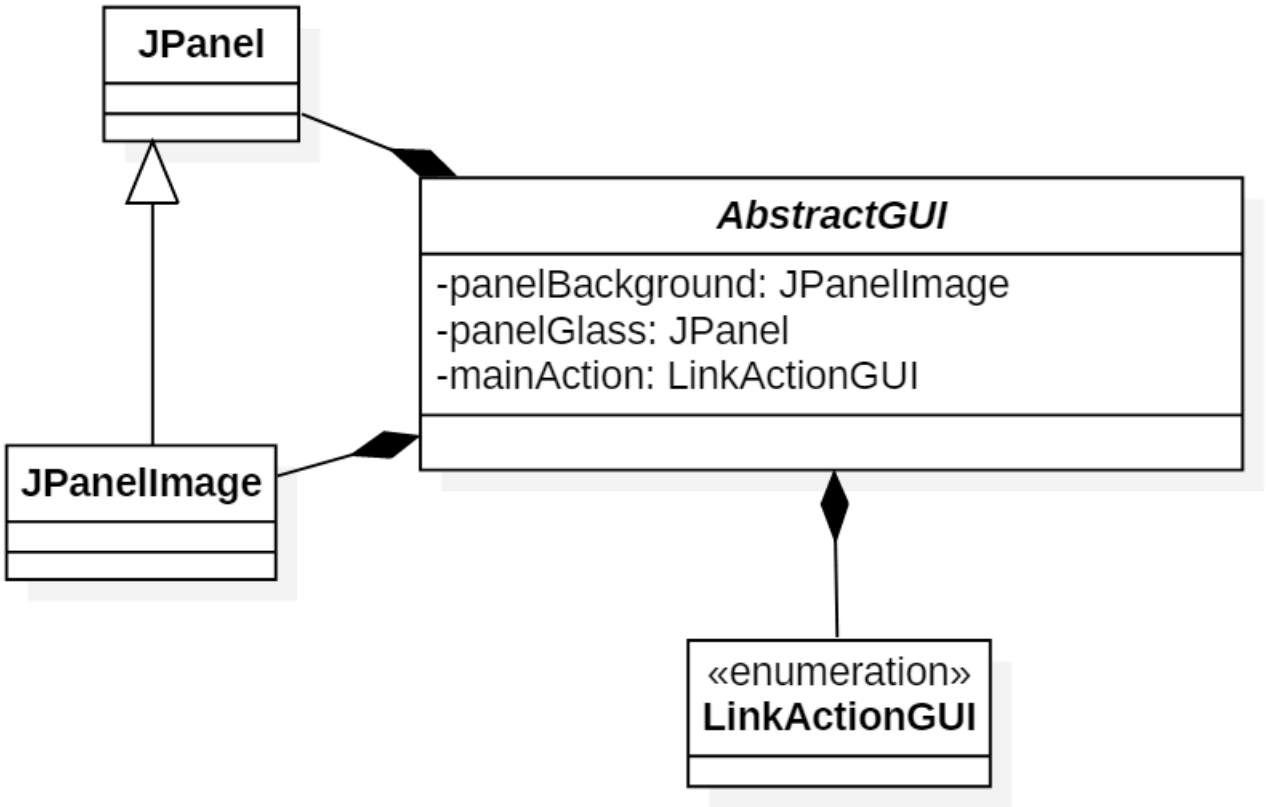


2.2 Design dettagliato

2.2.1 Buizo Manuel

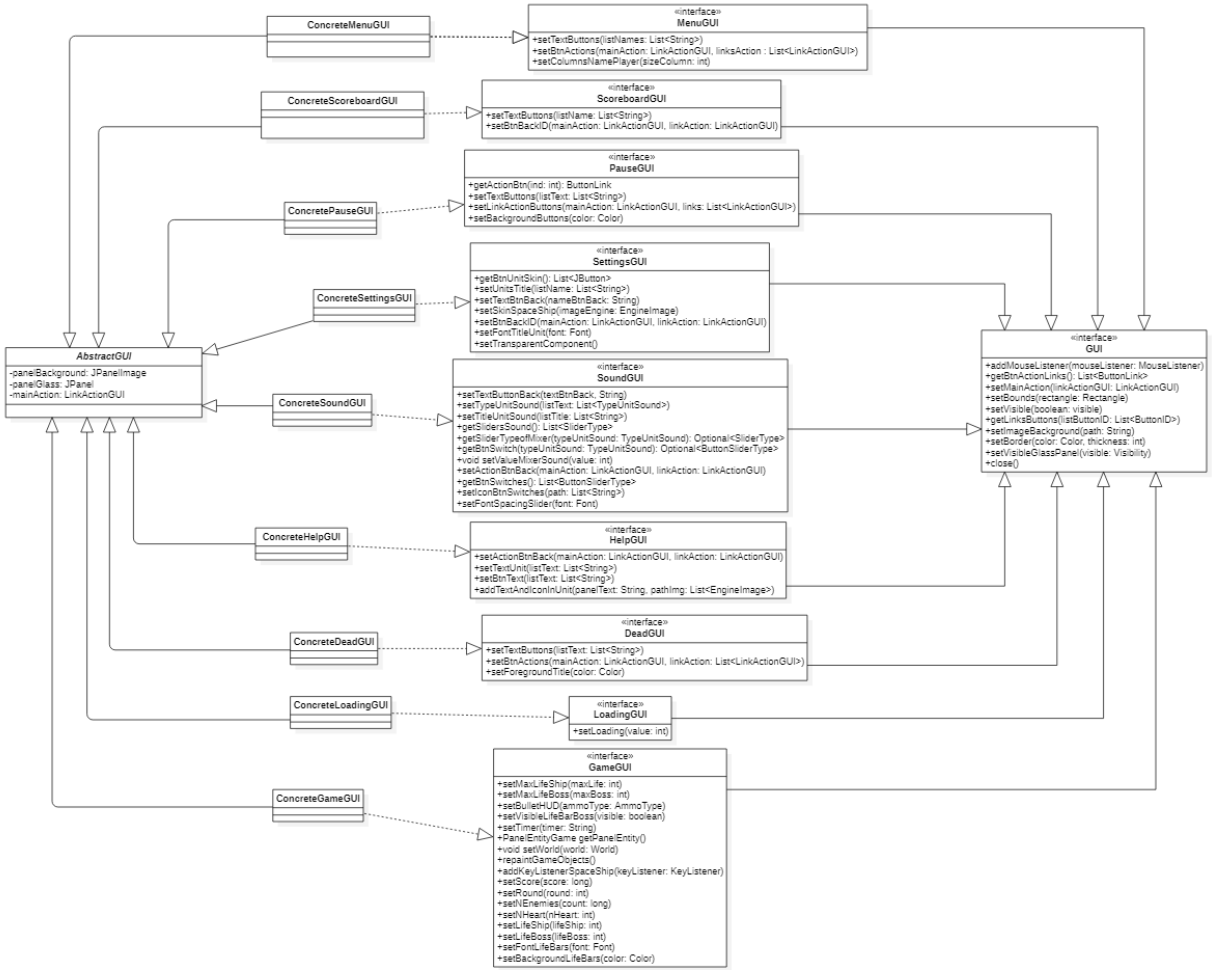
Creazione scheletro delle GUI

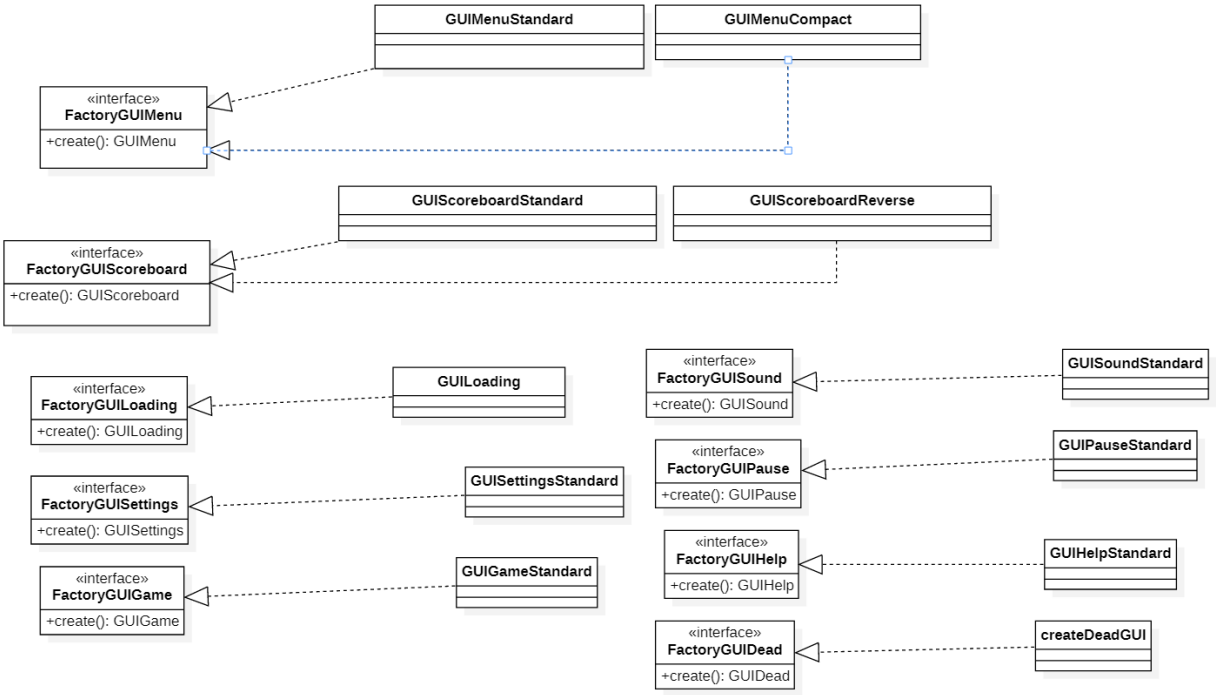
Per quanto riguarda la view, ho creato le finestre di gioco (GUI) usando la libreria Swing. In primis ho creato una classe astratta `AbstractGUI` estendendola dal `JFrame`, la quale ha i campi e le funzioni essenziali per la creazione di ogni finestra dell'applicativo.
Uso il `JPanelImage` come pannello di background per la finestra di gioco, nel quale aggiungo dei componenti e un'immagine di background. Inoltre aggiungo un `JPanel` al quale viene fissato il layout come `BorderLayout`, per renderlo in sovraimpressione.
Infine vi è un campo che funge da identificativo della finestra, dato da un Enumeration `LinkActionGUI` che serve per riconoscere la finestra e il proprio collegamento.



Uso del pattern per la creazione della grafica delle GUI di gioco

Mi sono avvalso del pattern `FactoryMethod`, infatti ho creato diverse factory per ogni singola finestra dell'applicativo, rendendo estendibile la creazione di nuove grafiche con l'utilizzo di *Swing*, senza perdere le proprie funzionalità.
Per racchiudere le funzionalità in comune di tutte le finestre, ho creato un interfaccia superiore denominata `GUI`. Poi ho definito un'interfaccia per ogni finestra lasciando alle sottoclassi la definizione della classe che deve essere istanziata. Eliminando così la necessità di riferirsi a classi dipendenti dall'applicazione all'interno del codice.



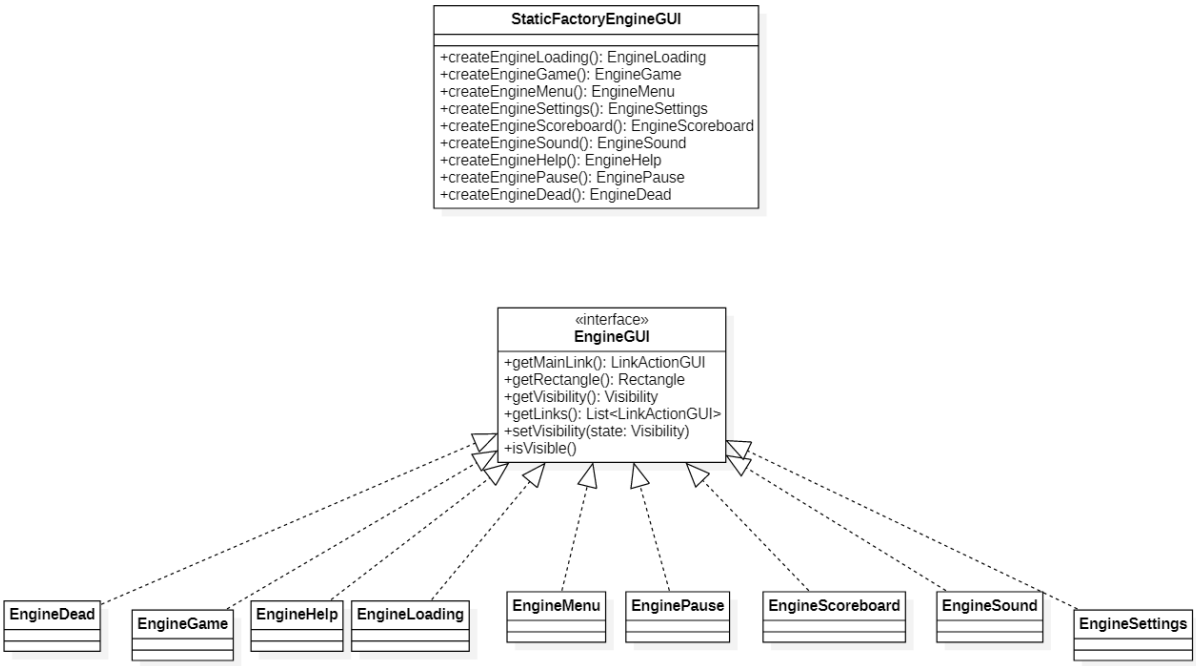


Creazione del modello delle GUI

Per quanto riguarda la parte del model delle GUI, ognuna di esse implementa un'interfaccia `EngineGUI`. Questa raccoglie le funzionalità indispensabili che servono alla view, ovvero:

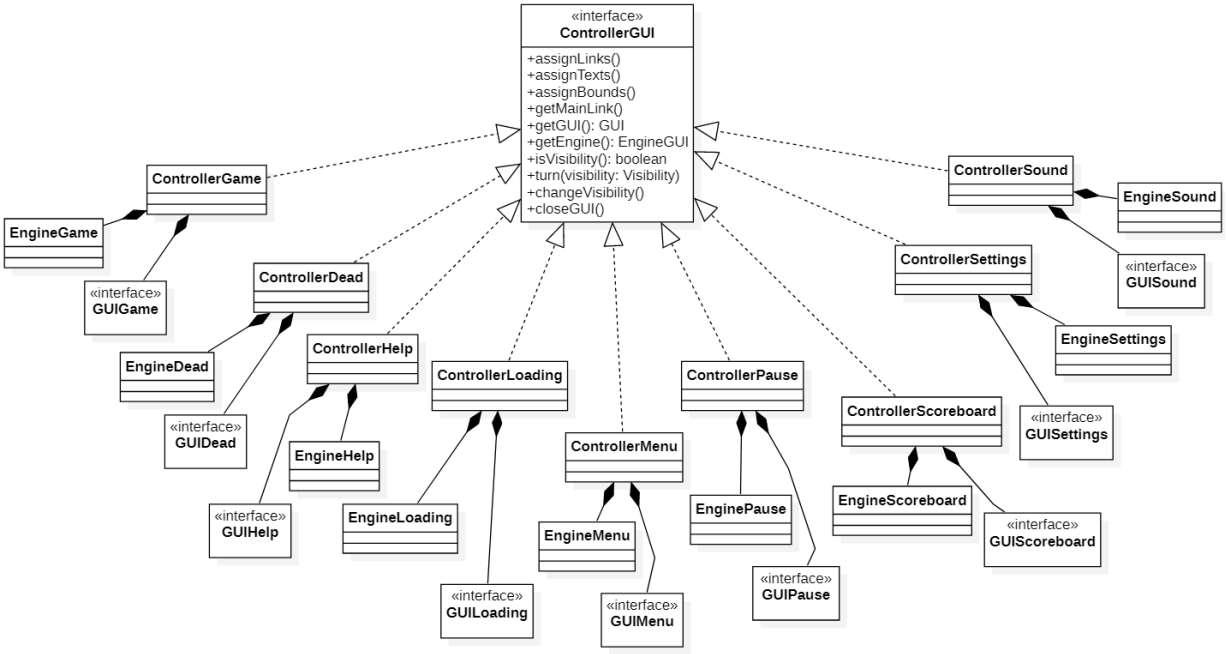
- L'identificatore
- La dimensione
- Le stringhe di testo delle componenti
- La visibilità
- I collegamenti verso le altre GUI

Il compito di istanziare tutti i modelli delle GUI l'ho attribuito alla classe `StaticFactoryEngineGUI`, che genera istanze specializzate di un Engine generica.
L'ho strutturata usando il pattern `StaticFactory`.


StaticFactoryEngineGUI
+createEngineLoading(): EngineLoading
+createEngineGame(): EngineGame
+createEngineMenu(): EngineMenu
+createEngineSettings(): EngineSettings
+createEngineScoreboard(): EngineScoreboard
+createEngineSound(): EngineSound
+createEngineHelp(): EngineHelp
+createEnginePause(): EnginePause
+createEngineDead(): EngineDead
«interface»
EngineGUI
+getMainLink(): LinkActionGUI
+getRectangle(): Rectangle
+getVisibility(): Visibility
+getLinks(): List<LinkActionGUI>
+setVisibility(state: Visibility)
+isVisible()
EngineDead
EngineGame
EngineHelp
EngineLoading
EngineMenu
EnginePause
EngineScoreboard
EngineSound
EngineSettings

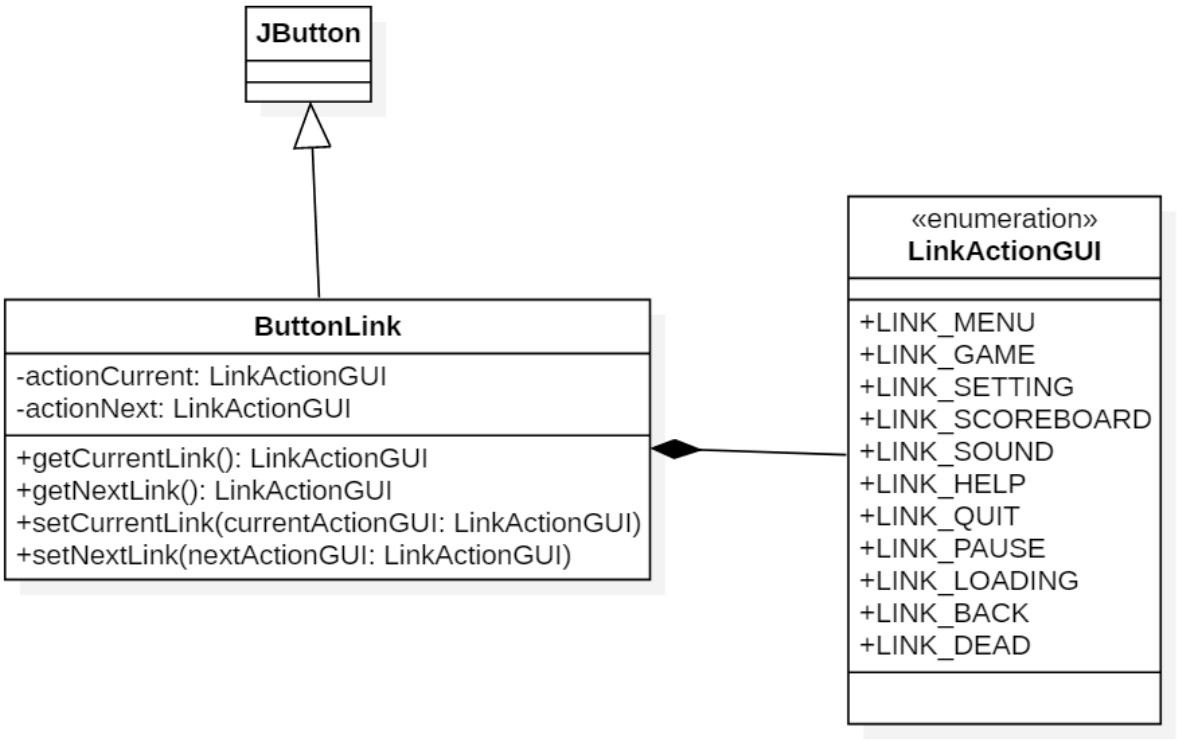
Realizzazione del controllo delle GUI

Per finire il quadro della struttura MVC (model-view-control) delle GUI, ho creato una classe controllo per ogni GUI, facendola aderire al contratto `ControllerGUI`. Ogni `ControllerGUI` ha il compito di recepire quando cambiano i dati del model, di comunicarlo alla controparte view, e di ottenere la propria view e model. Il `ControllerGUI` si salverà il `LinkActionGUI` della GUI associata.



Logica di collegamento delle GUI

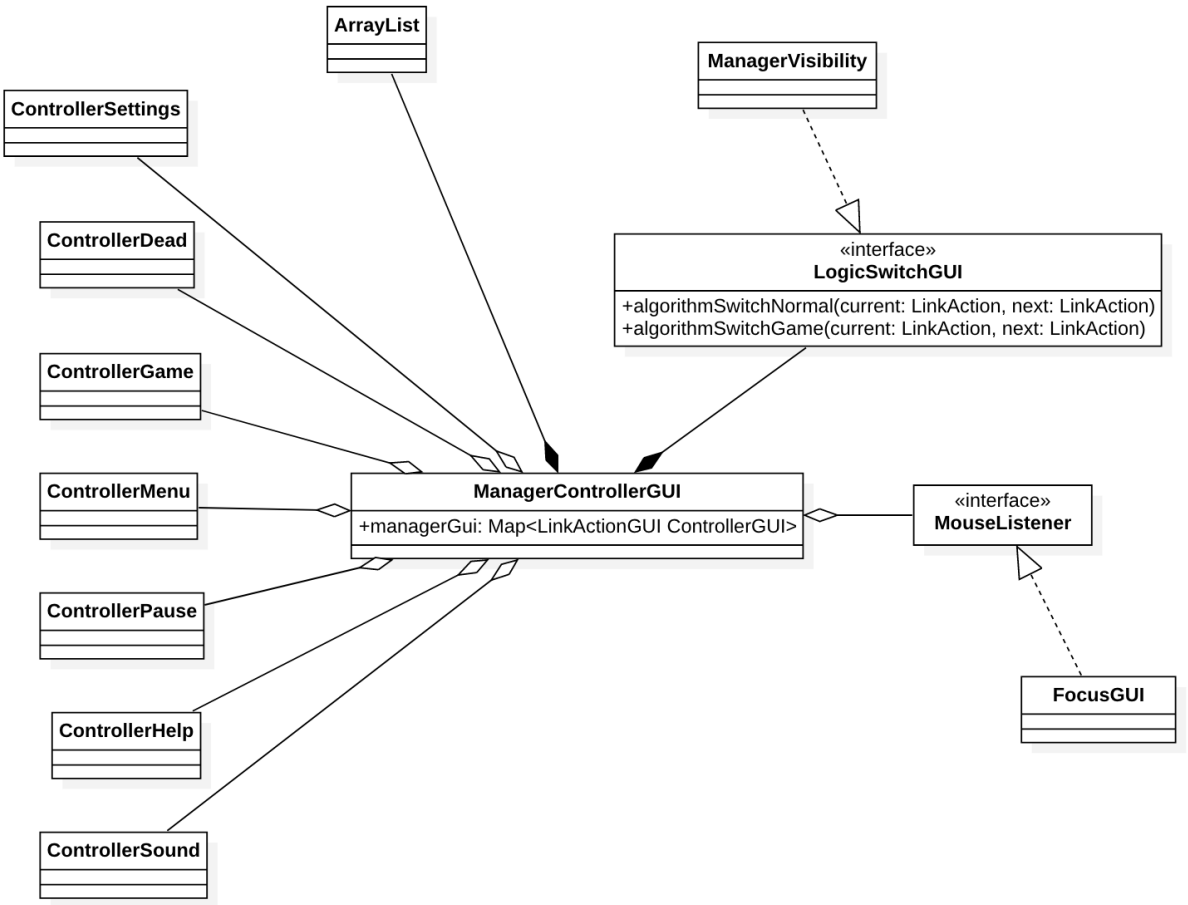
Le azioni che si possono eseguire da una GUI dipendono dall'Enumeration `LinkActionGUI`, la quale raccoglie tutte le possibili azioni eseguibili da una GUI(Esempio: `LinkActionGUI.LINK_MENU` rappresenta l'azione che fa apparire la schermata del menu principale). Ognuna di esse quindi possiede una `LinkActionGUI` come identificativo, ed una lista di queste, che rappresenta le azioni che si possono eseguire quando si è in quella GUI.



System GUI

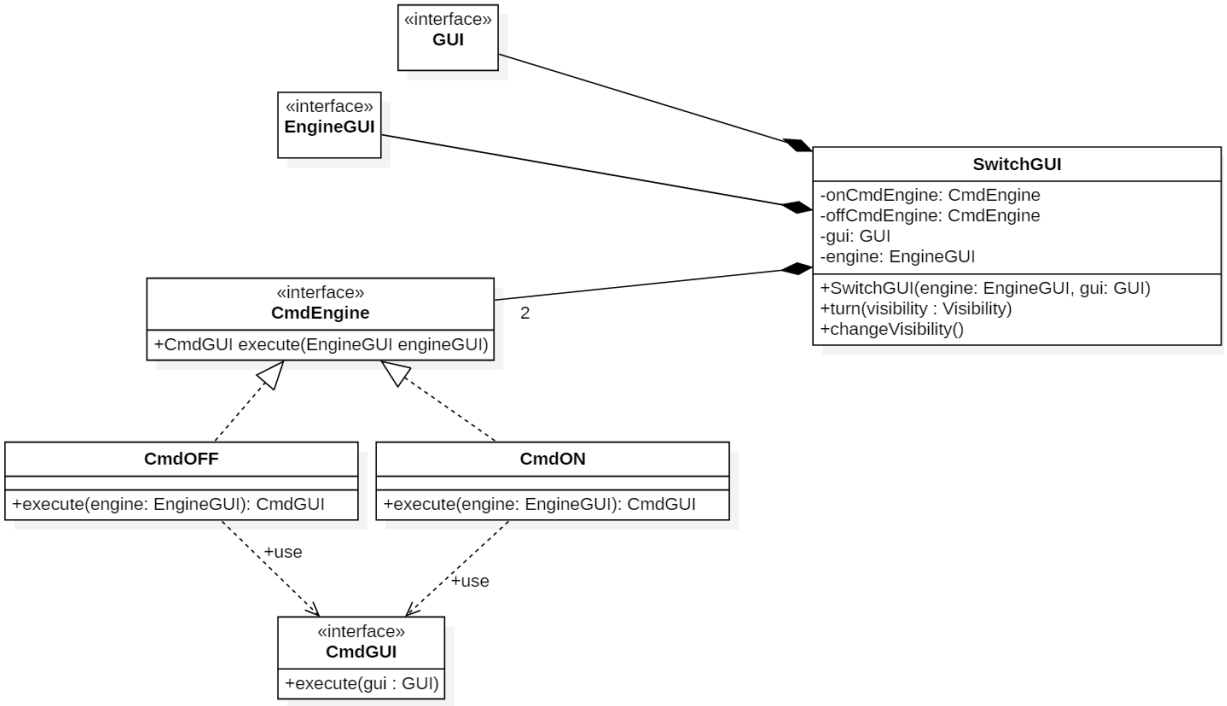
`ManagerControllerGUI` è la classe che si occupa di gestire i comandi lanciati dalle varie GUI. In base ad essi infatti, il manager mostra e/o nasconde le finestre di gioco. Il `ManagerControllerGUI` ha una `Map<LinkActionGUI, ControllerGUI>`, nella quale associa ogni `ControllerGUI` al proprio `LinkActionGUI`. La visibilità delle GUI è gestita tramite una logica `LogicSwitchGUI`, appoggiandosi ad una List di `LinkActionGUI`, per memorizzare la cronologia delle GUI visibili correnti. Alla lista, verrà aggiunta ogni volta l'azione che la GUI lancerà, tramite `LinkActionGUI` che sarà processata dalla `LogicSwitchGUI`, contenente le varie logiche.

Vi è anche un `MouseListener` per rimuovere tutte le GUI in sovrimpressione, fino a far apparire quella più in basso.



Interruttore delle GUI

Per cambiare lo stato della visibilità della `GUI` ho creato due interfacce `CmdEngine` e `CmdGUI`, implementando solo il `CmdEngine`, `CmdOFF` e `CmdON`. Il `CmdEngine` composto solo dalla funzione *execute* prende in ingresso un `Engine` e restituisce un `CmdGUI`, il quale prende in ingresso una `GUI`, obbligando quindi un ordine dei comandi, rispettando il pattern MVC. Per eseguire i comandi ho implementato la classe `SwitchGUI` che prende in ingresso un `Engine` e una `GUI`. La `SwitchGUI` tramite il metodo "`turn`" prende una `Visibility` come parametro e la utilizza per mostrare o nascondere le `GUI`, aggiornando prima lo stato del model e poi quello della view della `GUI`, avvalendosi delle lambda.

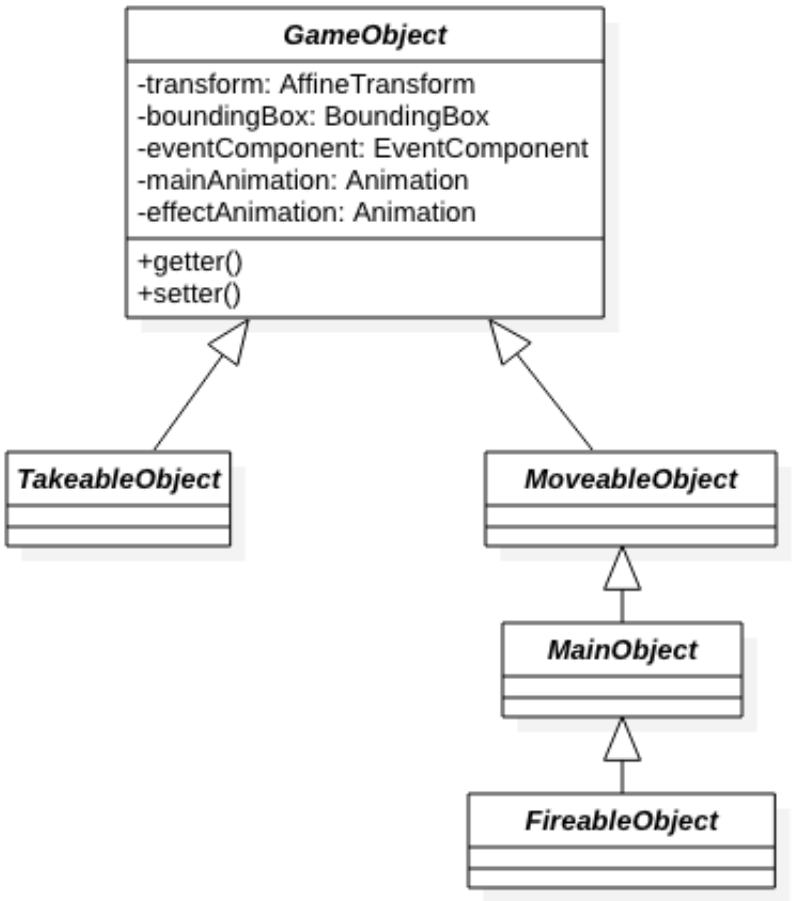


2.2.2 Matteini Mattia

GameObject

Le entità menzionate nell'analisi sono le protagoniste del gioco, tutte devono essere disegnate nella schermata di gioco, avere una posizione 2d sullo schermo, un componente per rilevare le collisioni e le animazioni(realizzate da Buizo). Ho quindi progettato una classe astratta che raccolga tutte queste caratteristiche, senza le quali nessuna classe sarebbe un oggetto di gioco.
Una nota importante riguarda la posizione, infatti inizialmente usavo sia l'`AffineTransform`, che un `P2d`, successivamente ci siamo resi conto che era superfluo perchè l'`AffineTransform` al suo interno aveva già le coordinate del punto.

A questo punto ho ideato la seguente gerarchia, creando le classi astratte in base alle funzionalità che gli oggetti devono avere e usando `GameObject` come "root", in seguito vedremo ogni classe nello specifico.

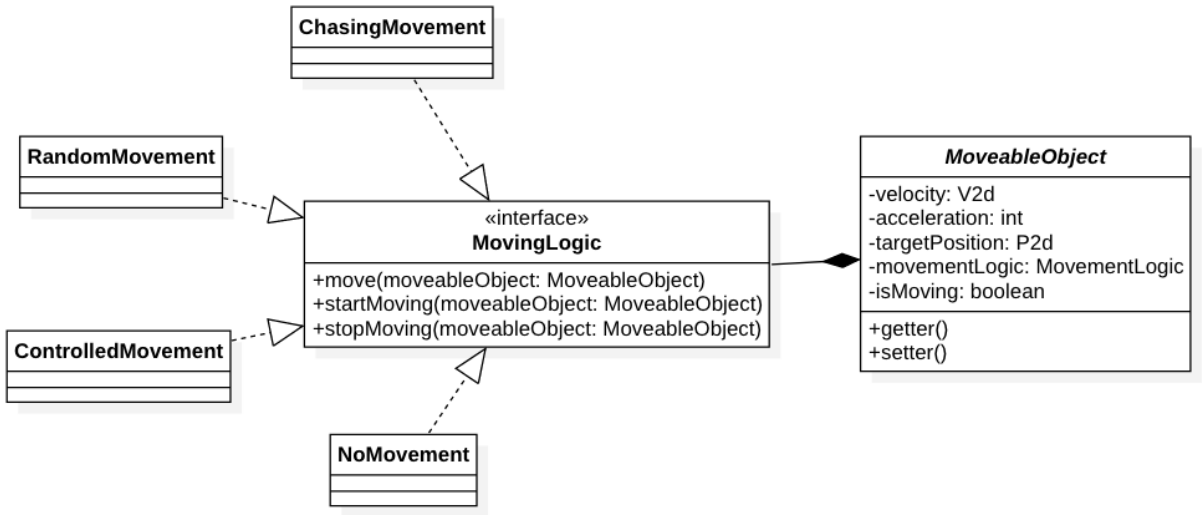


L'estensione più semplice di `GameObject` è `TakeableObject`, dalla quale estenderanno

i bonus, poichè non devono avere ulteriori funzionalità rispetto all'oggetto di gioco basico descritto in poco fa. Servirà semplicemente per garantire l'interoperabilità tra essi.

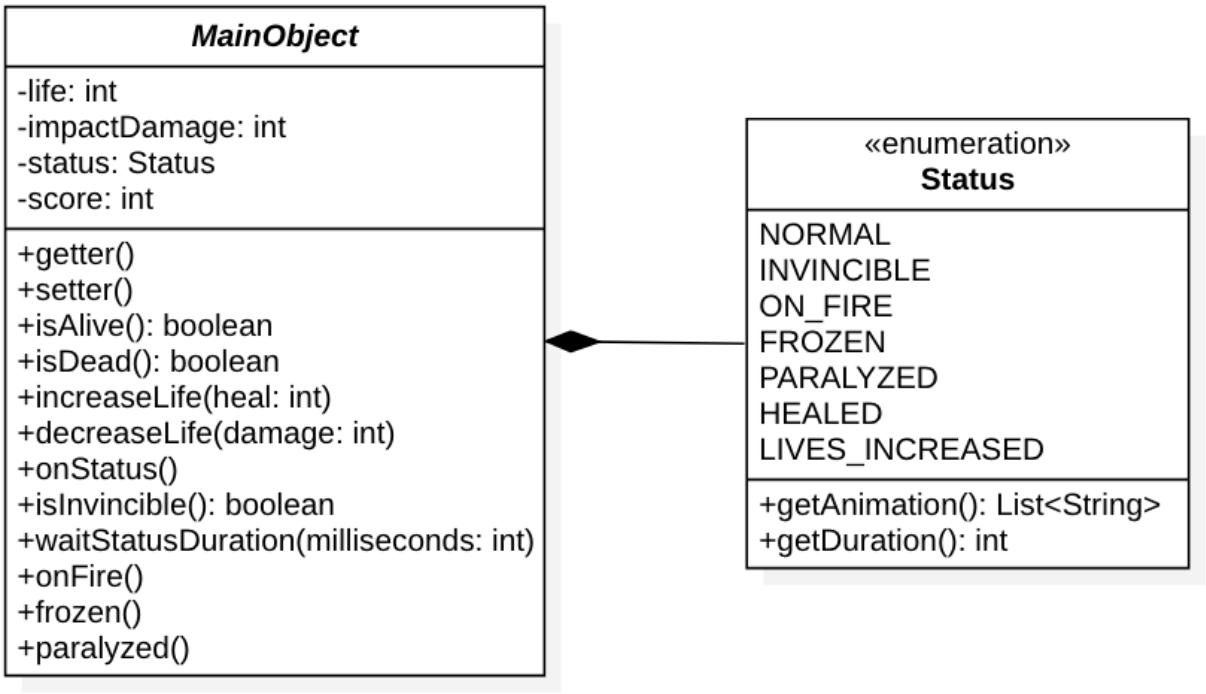
MoveableObject

Ora vediamo `MoveableObject`, questa classe implementa una logica di movimento tramite il pattern `Strategy`, per permettere all'oggetto di muoversi durante il gioco. Dispone inoltre di una velocità (vettore 2d), di uno scalare per l'accelerazione, e di punto rappresentante la posizione del bersaglio (il quale servirà per implementare una logica di inseguimento o di puntamento per la mira).



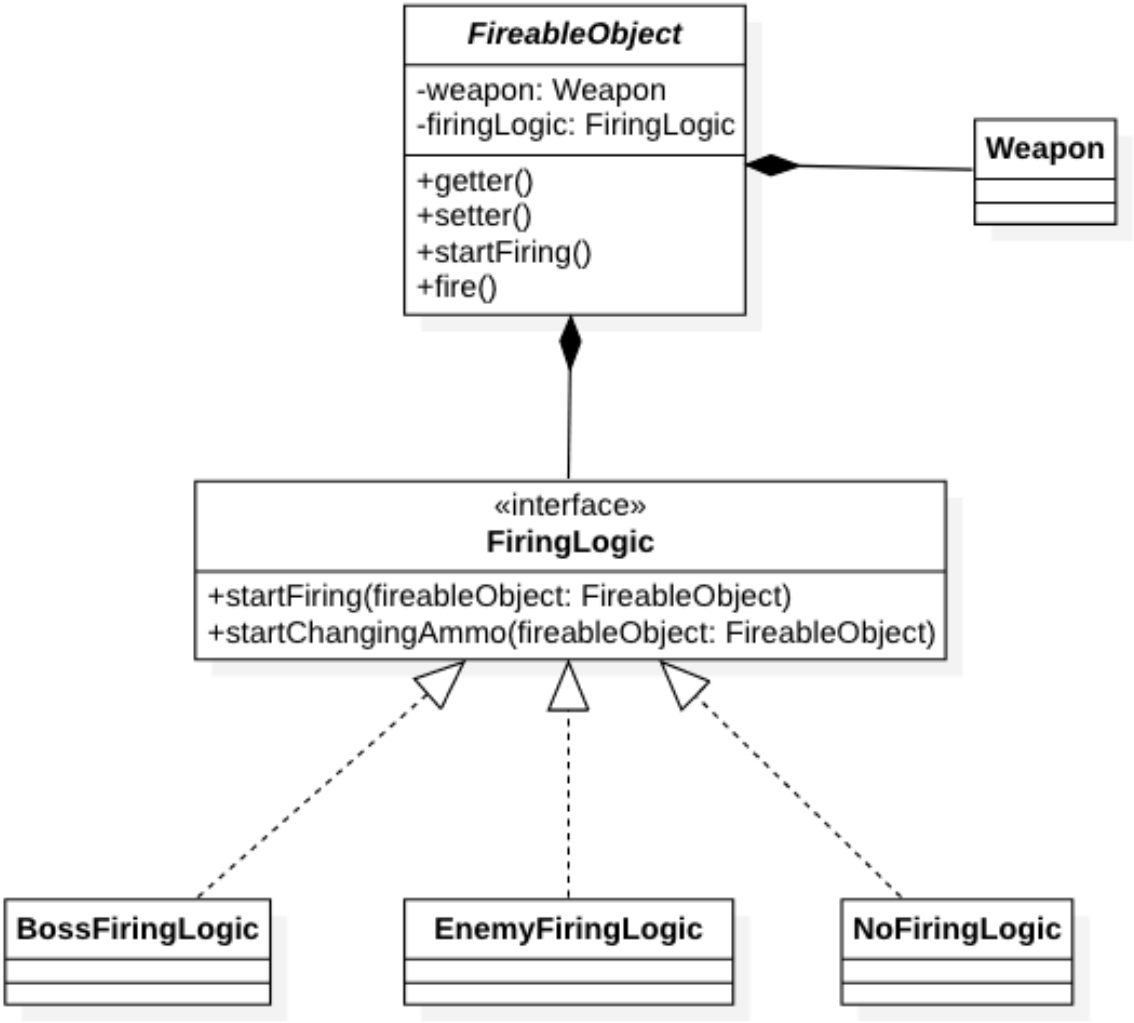
MainObject

Successivamente abbiamo il `MainObject`, che implementa tutto ciò che serve a un elemento di gioco "principale", ossia la vita, il danno da impatto, uno stato, e un punteggio.
Un qualsiasi `MainObject` può essere affetto da uno stato, quindi dovrà anche gestirlo (esempio: se è nello stato `Status.ON_FIRE` dovrà decrementare la sua vita per N secondi, poi tornare allo stato `Status.NORMAL`).
`Status` è un Enumeration che descrive tutti i tipi di stati che un `MainObject` può assumere, esso provvede anche la durata e una lista di stringhe per l'animazione dell'oggetto, la quale dipende dal tipo.



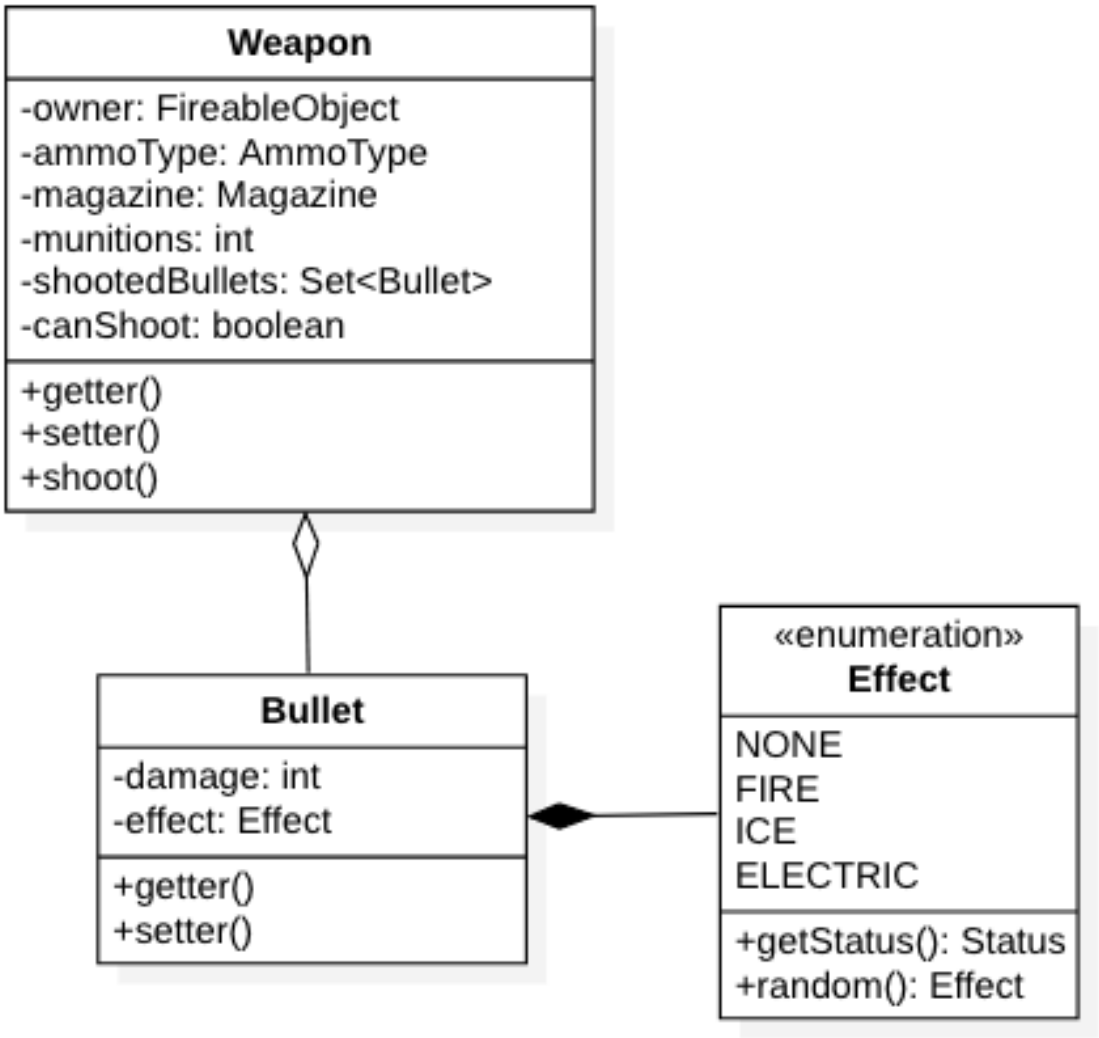
FireableObject

Infine abbiamo il `FireableObject`, questa classe fornisce una `Weapon`, e una logica per poter sparare implementata tramite il pattern `Strategy`.
La `Weapon`, tramite il metodo *shoot*, istanzia un nuovo `Bullet` con la stessa `AffineTransform` dell'oggetto che spara (owner) dato che il `Bullet` deve partire con la sua stessa posizione e rotazione.
Dopodichè, sempre il metodo *shoot*, aggiunge il `Bullet` appena creato al `World` e chiama la sua *startMoving* per farlo iniziare a muovere.



Bullet

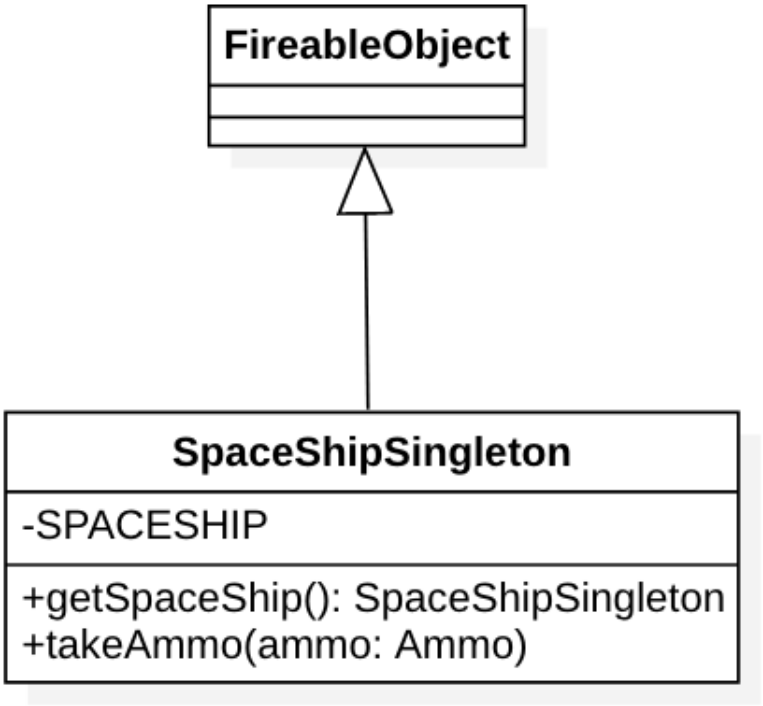
Il `Bullet` estende `MoveableObject` e possiede la logica di movimento `FixedMovement` che rende il suo moto rettilineo uniforme. Inoltre ciò che lo caratterizza è un danno e un tipo di effetto (`Effect`).
Nel momento in cui il `Bullet` andrà in collisione con un `MainObject` gli decrementerà la vita di una quantità pari al suo danno, e gli applicherà uno stato (`Status`) in base al tipo di `Effect`.



SpaceShipSingleton

Per quanto riguarda la navicella, ho deciso di usare il pattern Singleton, poiché il gioco è stato pensato per essere single player, quindi serve creare solo un istanza assicurandoci di fare sempre e solo riferimento ad essa.
Nello specifico, decido di istanziarla subito, seguendo quindi una versione eager di singleton, ho preso questa decisione perchè così facendo risolvo subito il problema della concorrenza (utilizzando vari thread nel gioco), rendendo l'istanza thread-safe.

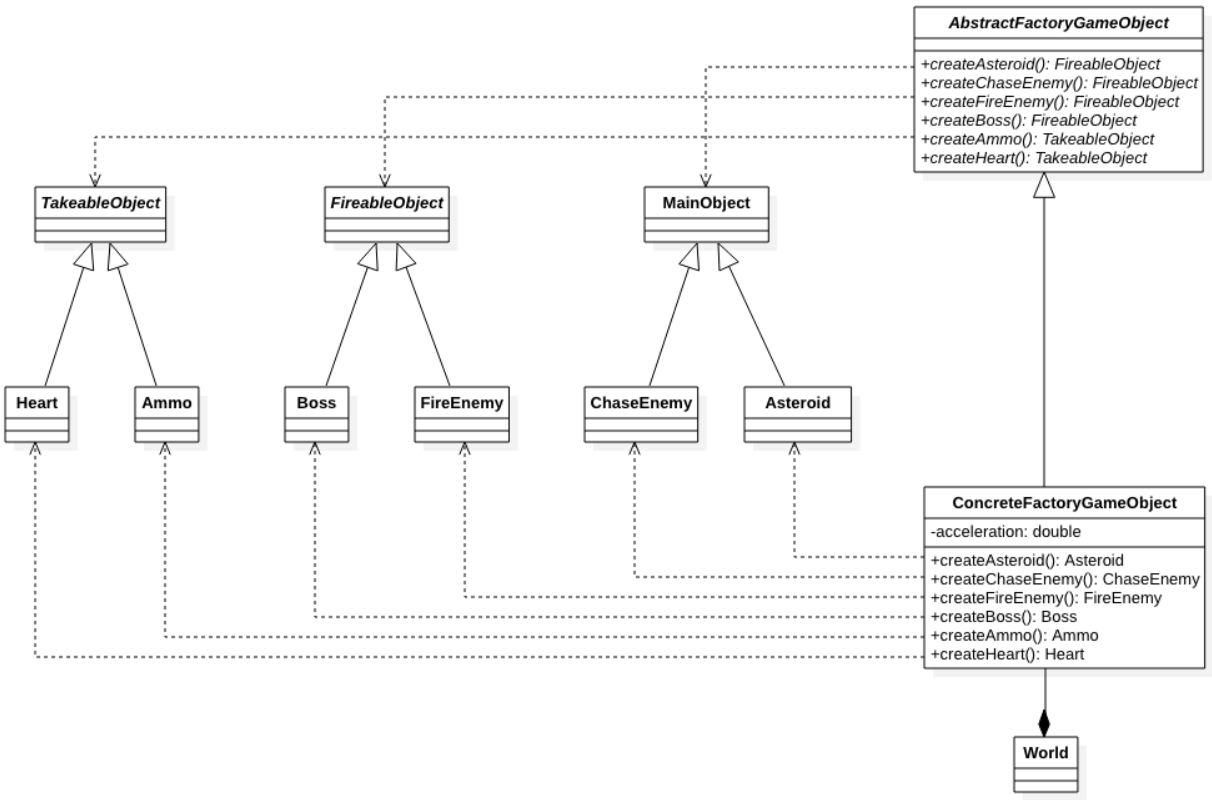
Ovviamente la navicella può sia sparare sia muoversi, quindi estende `FireableObject`. L'unica differenza rispetto agli altri oggetti di gioco che sparano è che, siccome gli spari sono gestiti dall'utente, la navicella implementa una `NoFiringLogic` come logica di sparo.



AbstractFactoryGameObject

Per la creazione degli oggetti di gioco, ho utilizzato il pattern `Abstract Factory`.

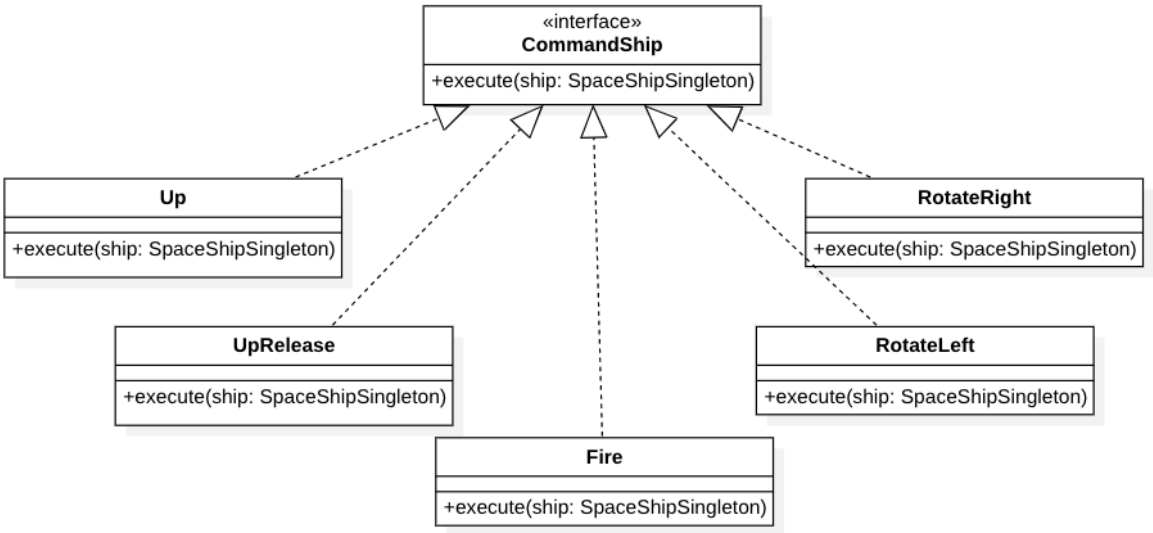
Così facendo posso creare agilmente tutti i tipi di oggetti di gioco di cui ho bisogno, rendendo così facile l'eventuale estensione dei loro metodi di creazione. Nel caso concreto, uso la factory per generare i bonus (`Ammo` e `Heart`) in posizioni casuali e i restanti oggetti (`Boss`, `ChaseEnemy`, `FireEnemy` e `Asteroid`) sui bordi dello schermo di gioco.
Inoltre, i `MoveableObject` che questa factory produce, non hanno accelerazione.



2.2.3 Paganelli Alberto

Commands for the space ship

Un requisito fondamentale in Space Survival è il movimento della navicella che viene comandata tramite tastiera dall'utente. Ho utilizzato il `Command` pattern perchè mi rende semplice l'implementazione dei comandi base oltre a consentirmi un'ottima estendibilità.

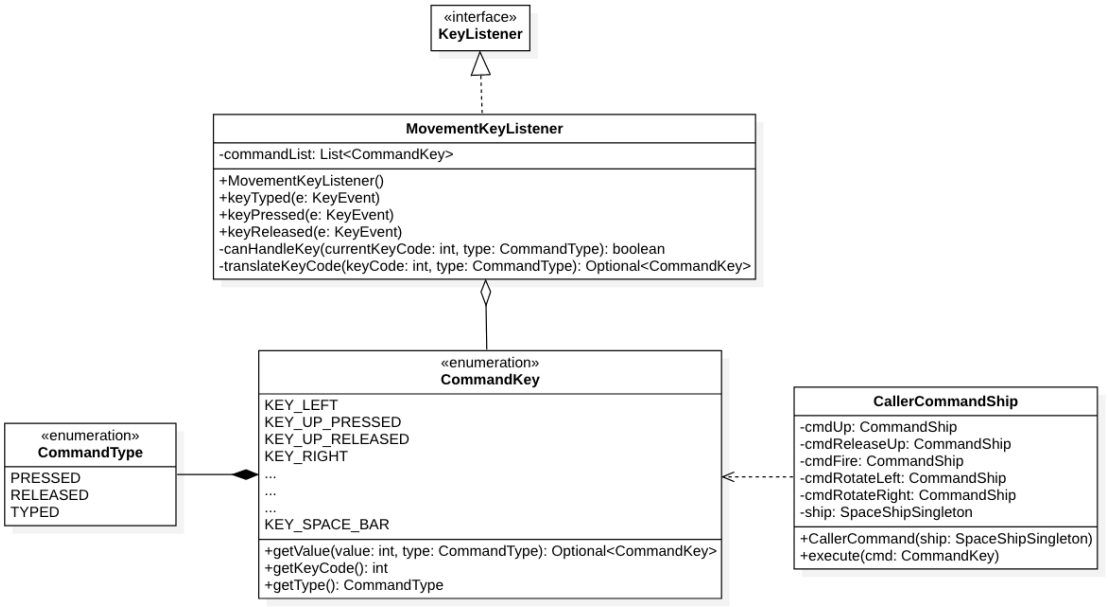


Essendo tutti gli input provenienti dalla tastiera, ho implementato un `MovementKeyListener` personalizzato partendo dall'interfaccia `KeyListener` che accetta solo i tipi di comandi specificati nell'Enumeration `CommandKey`.
L'Enumeration è composta dal nome del comando e dal relativo key code.

Oltre al `MovementKeyListener`, che inserisce i comandi in una opportuna lista, vi è il `CallerCommand`, ovvero un telecomando che, traducendo il `CommandKey`, richiama il giusto insieme di azioni specifiche per quel comando.

Tramite tastiera possono essere eseguiti 5 principali comandi:

- L'accelerazione della navicella
- La decelerazione della navicella
- La rotazione a sinistra
- La rotazione a destra
- Lo sparo



Quest'ultimo è stato composto da 5 campi `CommandShip` ed ognuno eseguirà la propria implementazione.

Questi 5 comandi principali possono essere richiamati tramite le frecce direzionali destra o sinistra per le rotazioni oppure con quella direzionata verso l'alto per l'accelerazione. Inoltre vi è la possibilità di usare le lettere A (rotate left), W (accelerate), D (rotate right), K (fire), Space Bar (fire). In quest'ottica, per aggiungere nuovi comandi basta implementare `CommandShip`, e, se quest'ultimo lo vogliamo richiamare tramite input da tastiera basterà aggiungere il keycode relativo al tasto in `CommandKey` e gestire il caso del nuovo comando in `CallerCommand` (se volessi eseguire una qualunque combinazione di comandi).

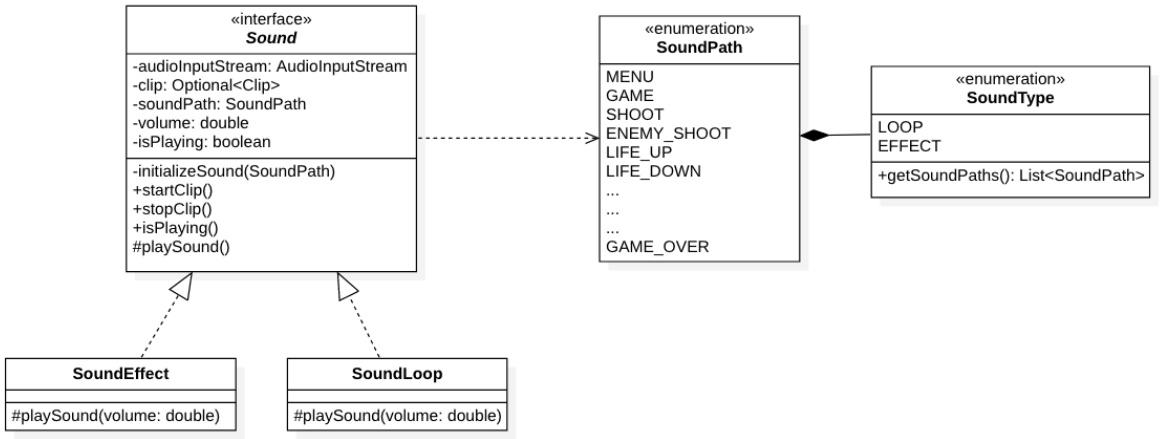
Suoni

In questo software il suono è presente sin dall'avvio. Il pattern usato è sempre il `Command` per poter accendere (`CommandAudioType.AUDIO_ON`), spegnere (`CommandAudioType.AUDIO_OFF`) o resettare (`CommandAudioType.RESET_TIMING`) il timing relativo al suono, a prescindere dal tipo di quest'ultimo.

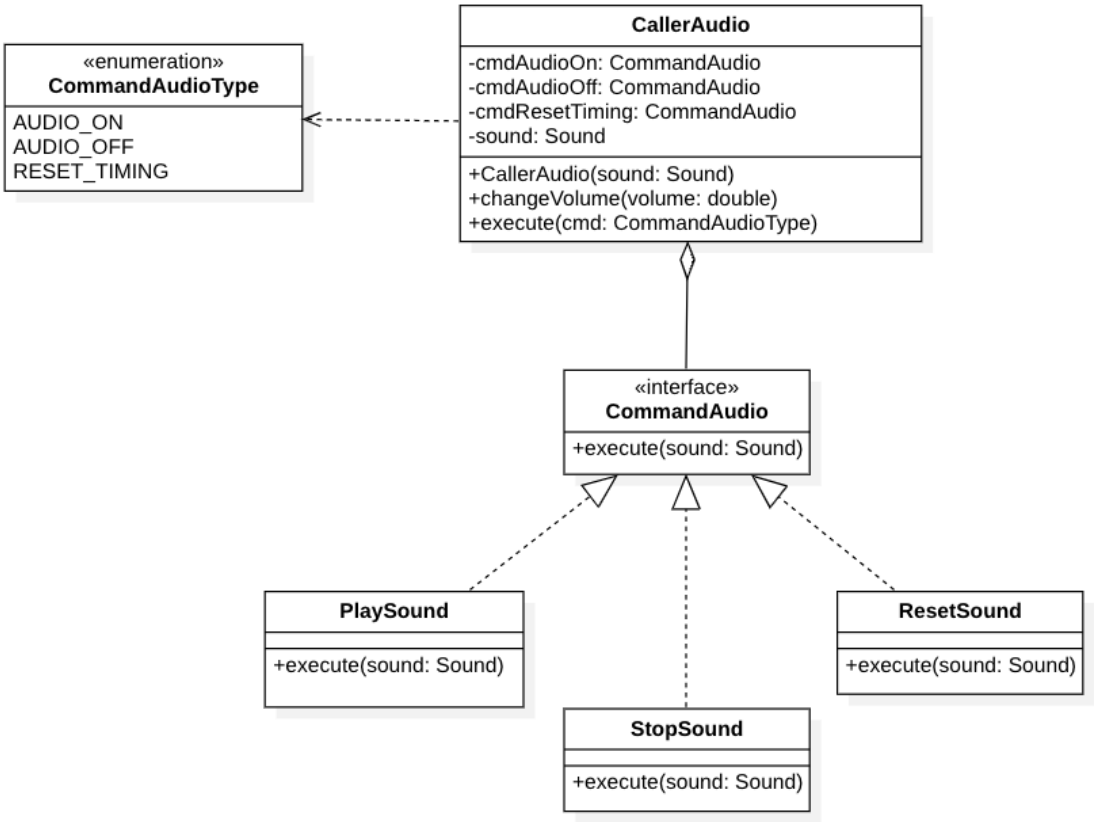
Ho creato una classe astratta `Sound` composta dai metodi di base per la gestione del suono, come lo start (*startClip*), lo stop (*stopClip*) o il cambio del volume stesso (*setVolume*).

Dopodiché ho implementato due classi, `SoundEffect` e `SoundLoop`. Come si può evincere dal nome le due classi eseguiranno il suono una sola volta o in loop. In questo modo posso decidere di avviare un suono come meglio credo implementando solo il metodo *playSound* e soprattutto senza riscrivere tutti gli altri metodi.

Ogni suono è caratterizzato da una Enumeration di tipo `SoundPath`, composta dal percorso e dal tipo di quest'ultimo, in modo che il `CallerAudio` riesca a capire come gestire il suono passato.
Per aggiungere nuovi suoni basterà inserire il nome del suono nell'Enumeration, aggiungere il file nella cartella /res/sounds e richiamare la *playSound* passandogli il `SoundPath` desiderato.

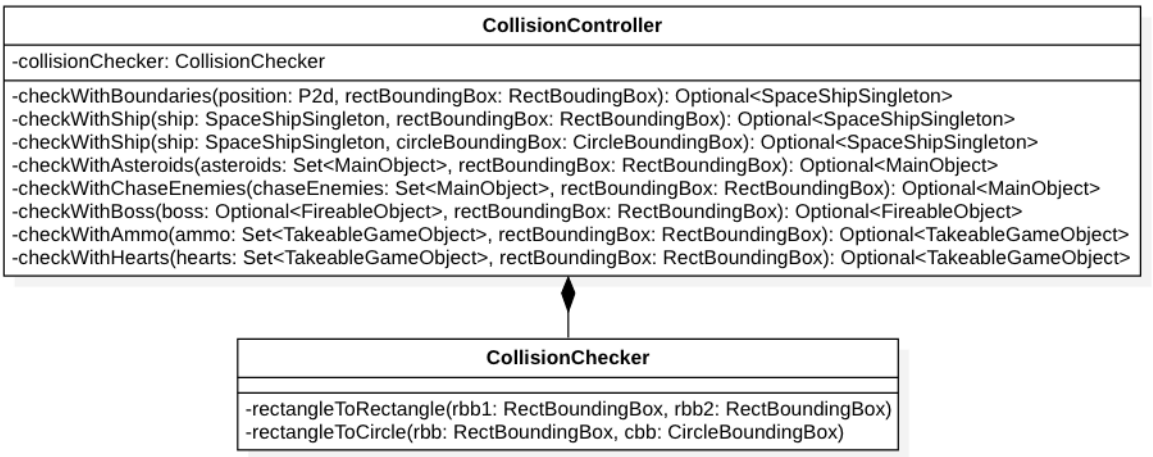


`SoundLoop` è la classe di appartenenza del suono durante il gioco o quello del menu di pausa mentre `SoundEffect` è la classe di appartenenza di tutti gli effetti di gioco come sparo, raccolta di un perk, ecc.
L'obiettivo posto era rendere indipendenti i vari suoni evitando che, alla partenza di un nuovo effetto, quello prima si stoppasse. Ho evitato questo problema componendo un `CallerAudio` per ogni effetto, ed eseguendo un comando `CommandAudioType.AUDIO_ON` ogni volta che avveniva un certo evento come collisione con perk o con nemici.


«enumeration»
CommandAudioType
AUDIO_ON
AUDIO_OFF
RESET_TIMING
CallerAudio
-cmdAudioOn: CommandAudio
-cmdAudioOff: CommandAudio
-cmdResetTiming: CommandAudio
-sound: Sound
+CallerAudio(sound: Sound)
+changeVolume(volume: double)
+execute(cmd: CommandAudioType)
«interface»
CommandAudio
+execute(sound: Sound)
PlaySound
+execute(sound: Sound)
StopSound
+execute(sound: Sound)
ResetSound
+execute(sound: Sound)

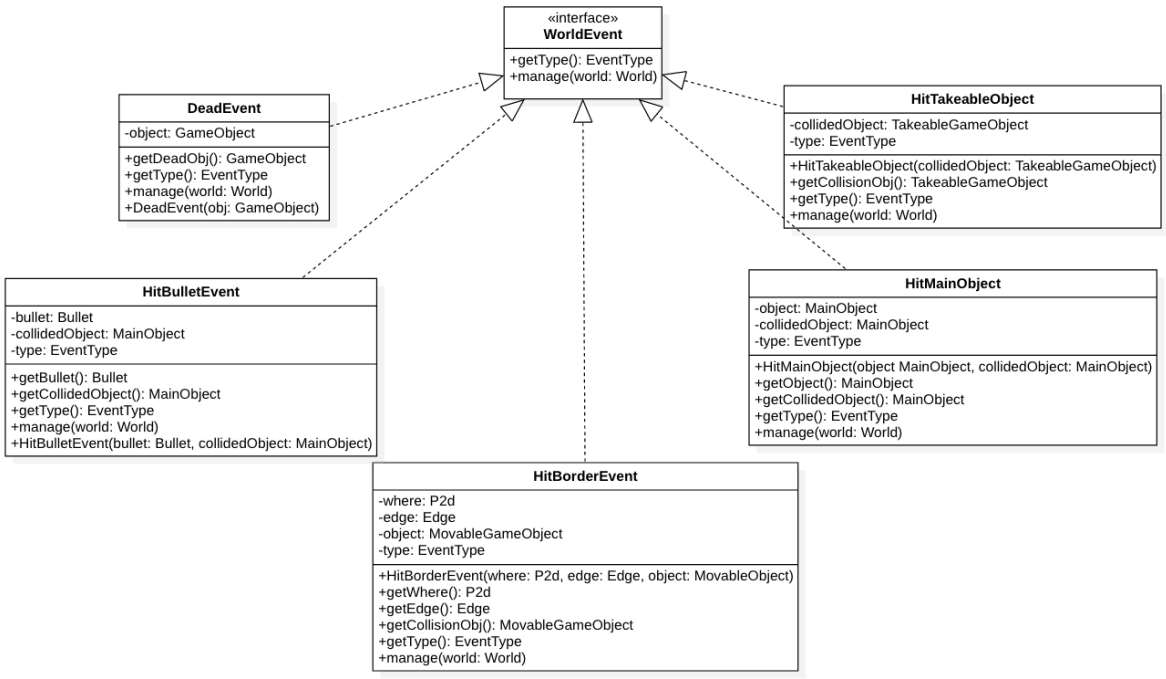
Collisioni

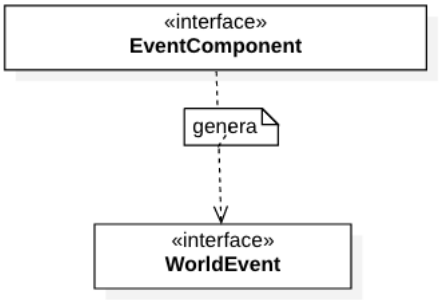
Per effettuare il controllo sulle collisioni tra gli oggetti di gioco sono partito creando una classe `CollisionChecker` nella quale ho implementato 2 metodi per controllare tutte le possibili collisioni tra gli oggetti. Ho basato il `CollisionChecker` su `RectBoundingBox` e `CircleBoudingBox`, ovvero i contorni relativi a ogni tipo di oggetto di gioco.
Questa sarà la classe sulla quale agire nel caso si vogliano controllare collisioni di elementi di gioco che hanno una forma per adesso non tollerata, ovvero diversi da un cerchio oppure da un rettangolo.
La classe `CollisionChecker`, tramite l'uso del metodo dell'`AffineTransform` *createTransformedShape*, controlla le collisioni anche delle figure sottoposte a rotazione, quindi trasformate.
Inoltre è stato creato un `CollisionController` per poter controllare solo le collisioni desiderate.
Bisognerà agire su quest'ultimo se si vogliono rilassare collisioni o aggiungerne altre con nuovi elementi di gioco.



Oltre alle collisioni è stata implementata un struttura tramite la quale è possibile gestire gli eventi di gioco.
Per poter notificare eventi diversi senza ripetizioni di codice è stata creata l'interfaccia `WorldEvent` con un metodo *manage* poi da implementare per ogni singolo evento.
In questo modo, richiamando la *manage* sull'evento che implementa l'interfaccia `WorldEvent`, verrà richiamata la gestione specifica nell'evento.

La struttura degli eventi è la seguente:





Anche qui è stata utilizzata una enumeration `EventType` per identificare il tipo di evento. Gli eventi inoltre non hanno necessariamente un legame con le collisioni, quindi se ne possono creare di nuovi per gestire altri scenari riuscendo ad implementare facilmente nuove gestioni degli stessi.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

3.1.1 Buizo Manuel

Test in JUnit realizzati:

- `TestMenu`,
- `TestSwitchGUI`

Ho utilizzato il `TestMenu` per controllare se la parte dell' `Engine` viene assegnato tramite il Controller alla View, ovvero controllando l'assegnamento dei test e la visibilità ma soprattutto verificando che i collegamenti che ha il model passino alla view.
Ho utilizzato questa GUI per effettuare il test della struttura MVC.

Ho utilizzato invece il `TestSwitchGUI`, per controllare il cambiamento dello stato di visibilità, verificando che la struttura delle interfacce a cascata funzioni, infatti creando un comando `CmdGUI` ed facendo *execute* della GUI, il suo stato non rispetta lo stato del Model.

3.1.2 Matteini Mattia

Test in JUnit realizzati:

- `TestGameObject`,
- `TestWeapon`,
- `TestFactory`

Mi sono avvalso del `TestGameObject` per testare le funzionalità principali del `GameObject`, partendo dalla posizione, arrivando alla gestione degli stati del `MainObject`.
Il `TestWeapon` contiene i test per controllare che la `Weapon` esegua bene gli spari, scali le

munizioni e cambi tipo di munizioni in maniera corretta.
Invece con il `TestFactory`, mi assicuravo che ogni oggetto fosse creato con i giusti valori iniziali.

3.1.3 Paganelli Alberto

Test in JUnit realizzati:

- `TestCallerAudio`,
- `TestCollisionChecker`,
- `TestTranslationKeyCode`

L'obiettivo dei test automatizzati per il `CallerAudio` era testare che i `CallerAudio` eseguissero un comando specifico come il `CommandType.AUDIO_ON` e il `commandType.AUDIO_OFF` e che anche nel caso di comandi multipli su uno stesso suono questo non desse problemi.
Inoltre per quanto riguarda il `CollisionChecker` ho eseguito più test, sia tra rettangoli e rettangoli, che tra forme trasformate.
Infine ne ho creati due per il `TranslationKeyCode`, che pur essendo un semplice componente con il compito di traduttore il codice del tasto nel comando effettivo, mi aveva dato un problema in fase di sviluppo.

3.2 Metodologia di lavoro

La metodologia di lavoro che abbiamo seguito è stata inizialmente un po' caotica, abbiamo avuto qualche problema in fase di analisi, avevamo chiaro il gioco che volevamo sviluppare ma non sapevamo come distribuire in maniera efficace il carico di lavoro e soprattutto non sapevamo in quanto tempo saremmo riusciti ad implementare le funzionalità obbligatorie in maniera corretta.
Ci siamo approcciati al problema proseguendo a step, creando grafici per rendere idee generali più concrete, come ad esempio schemi ristretti ma significativi per Game Object e GUI, oltre ad evidenziare come le parti implementate singolarmente sarebbero state integrate.
Dopo un analisi dettagliata, durata qualche ora in più rispetto al previsto siamo riusciti a garantire un buon carico di lavoro adeguato a ogni studente del team e soprattutto ad andare avanti parallelamente.
Abbiamo proceduto a step, seguendo un todo list condiviso che veniva aggiornato periodicamente.
Circa 2-3 giorni inoltre ognuno di noi aggiornava il team sul lavoro svolto, ci sentivamo inceve con costanza quando invece vi era la necessità di integrare al meglio le funzionalità che richiedevano entità di gioco sviluppate separatamente.

Vi sono state alcune mansione che abbiamo svolto insieme, come alcune integrazioni tra classi sviluppate da persone diverse del team.

DVCS

Il DVCS utilizzato è stato git, abbiamo creato nostri 3 branch, non essendo inizialmente pratici con git con il tempo abbiamo avuto modo di comprendere al meglio le sue funzionalità base oltre alla comodità assoluta di un sistema distribuito.

3.2.1 Buizo Manuel

Ho iniziato il mio lavoro creando la prima finestra di gioco, "il menu". Inizialmente creando una un piccolo modello, creando dei testi e assegnandoli ai componenti, per esempio i bottoni. Una volta assegnati i testi senza passare da terze classi, ho implementato un Controller che si prendesse carico dell'assegnamento. Una volta che il ciclo dell'assegnazione avveniva con successo rispecchiando MVC, ho aggiunto nuove componenti alla view e il suo relativo modello come per esempio l'assegnamento dell'identificativo della GUI.
Ogni volta che la classe diventava troppo corposa cercavo di dividerla per domini, in modo da dare una una composizione di oggetti dello stesso dominio.
Una volta che l' architettura MVC era interamente costruita, cercavo di rendere il più estendibile l'architettura dei componenti assegnando interfacce e creando delle factory per creazione di oggetti.
Riguardo il lavoro in team ho fornito componenti finiti, ad esempio con Matteini ho creato le componenti animazioni per i suoi `GameObject`. D'altra parte con Paganelli, per la creazione dei suoni gli ho fornito il componente Controller della finestra di suono, dove l'utente interagisce con la GUI grazie ai Listener offerti da Swing.

3.2.2 Matteini Mattia

Realizzazione delle entità principali di gioco, quindi del `GameObject` e di tutte le sue estensioni, con relativa factory per la loro creazione.
Ideazione della logica di sparo per i `FireableObject`, e della logica di movimento per i `MoveableObject` (con l'aiuto di Paganelli per far fronte ai problemi con l'`AffineTransform`).
Realizzazione dei vari tipi di bonus e gestione delle interazioni tra essi e le armi(`Weapon`).
Gestione degli effetti(`Effect`) e di come vengono applicati ai `GameObject`, cambiando il loro stato (`Status`).
Creazione dei proiettili (`Bullet`), implementazione parziale del `World` e dell'`EngineLoop` (ai quali abbiamo lavorato tutti e tre).
L'implementazione degli eventi e delle collisioni è stata frutto di un lavoro congiunto con Paganelli, poichè bisognava collegare gli oggetti di gioco al sistema delle collisioni da lui implementato.

3.2.3 Paganelli Alberto

Il mio ruolo era la gestione dei comandi della navicella, quindi leggere gli input e tradurli in azioni specifiche della navicella con possibilità di aggiungere eventuali comandi in maniera semplice per sviluppi futuri.
Inoltre dovevo comporre il sistema di suono del gioco compreso sia da quello in background che dagli eventuali effetti durante il gioco, anche qui in maniera facilmente estendibile.
Ed infine come ultimo elemento ho composto la classe `CollisionChecker` per il check delle collisioni in forme trasformate oltre ad un sistema di eventi per notificarne la presenza.
Per quanto riguarda le integrazioni con altri componenti del gruppo ho svolto insieme a Matteini quelle che sono le collisioni degli oggetti di gioco e gli oggetti di gioco stessi che ci hanno portato a comporre eventi per poter notificare le collisioni indipendentemente da tutti gli altri oggetti di gioco.
Inoltre un'altra integrazione anch'essa importante è stata tra i movimenti e la `SpaceShipSingleton`. Abbiamo notato che avere un campo `P2d` (quindi un punto con coordinate x e y) e un campo `AffineTrasform` (anch'esso con coordinate x e y) fosse superfluo, così abbiamo modificato il metodo *getPosition* in modo da mantenere un linguaggio comprensibile ma evitando attributi inutili.

3.3 Note di sviluppo

3.3.1 Buizo Manuel

Features avanzate utilizzate:

- **Optional**, per gestire eventuali valori non corretti di ritorno
- **Lambda Expressions e Stream**, per la ricerca, la sostituzione e eliminazione di elementi dalle collection.
- **Thread**, per caricamento lista usata per graficare oggetti di gioco.

Preso da *StackOverflow* algoritmo per calcolare il colore della vita data la vita stessa.

3.3.2 Matteini Mattia

Features avanzate utilizzate:

- **Optional**, utilizzati spesso per ricavare gli oggetti di gioco dal `World` poiché durante la partita è facile che vengano eliminati al momento della richiesta, e quindi ottenere *null*.
- **Stream**, usato per effettuare ricerche profonde sui set dei `GameObject` nel `World`.
- **Thread**, fondamentali per la realizzazione di un gioco di questo genere, li ho usati per implementare gli stati dei `MainObject` al fine di non bloccare tutto il gioco.

- **Lambda Expressions** sicuramente molto utili per ottimizzare la scrittura del codice, usate in particolare in `Stream` e `Thread`.

Preso spunto da *StackOverflow* per tornare un elemento casuale da un Enumeration.

3.3.3 Paganelli Alberto

Features avanzate utilizzate:

- **Optional**, utilizzati per controllare la validità dei comandi inseriti dall'utente.
- **Lambda Expressions**, per facilitare la lettura del codice stesso.
- **Stream** similmente alla lambda expression rende il codice molto più pulito e facile da interpretare.
- **AffineTransform**, non è una libreria esterna ma l'ho trovata molto potente.

Algoritmi interessanti implementati:

- **Sistema di puntamento**, utilizzo del *Teorema di Talete* per calcolare la giusta angolazione e ruotare un oggetto verso un determinato punto.

Ho rivisto le classi `RectBoundingBox` e `CircleBoundingBox` aggiungendo loro trasformazioni, oltre alle classi `P2d` e `V2d` utilizzate anche durante il seminario **Game As A Lab** tenuto dal professore **A. Ricci**.

Capitolo 4

Commenti finali

4.1 Autovalutazione

4.1.1 Buizo Manuel

Ritengo che la maggior parte del progetto che ho svolto sia corretta ma ritengo che ci sia ancora un buon margine di miglioramento dell'applicativo. Durante la stesura del codice ho cercato sempre di appoggiarmi ad un interfaccia in modo da rendere il più delle volte il codice estendibile tramite una `FactoryMethod` e altre. Infatti Le migliori soluzioni che trovato, le ho ottenute applicando i patters o creando meccanismi tra interfacce. Avrei voluto gestire meglio il coordinamento delle GUI, avvaledomi di `Strategy`.
Solo arrivando alla fine del progetto, riesco ad avere più chiaro il dominio applicativo del gioco. Tornando indietro adopererei più interfacce per i controlli oltre a dividere meglio le gestioni di controllo dei componenti.

4.1.2 Matteini Mattia

Personalmente penso di avere svolto un lavoro discreto nel complesso, tuttavia potevo fare di meglio nelle fase di analisi e di progettazione, infatti andando avanti nel progetto, mi sono reso conto che potevo progettare meglio la gerarchia dei `GameObject`, utilizzando per esempio delle interfacce per garantire una migliore estensibilità. Il ruolo che ho avuto è stato molto stimolante, infatti ho ideato lo scheletro degli oggetti di gioco, che sono una parte fondamentale del modello. Nel complesso sono abbastanza soddisfatto del risultato, sia perché è venuto un gioco divertente (a mio parere), sia perché possiamo pensare a molteplici funzionalità possibili da implementare in poco tempo grazie all'impegno messo da tutti per renderlo facilmente estendibile. Abbiamo intenzione infatti di aggiornarlo per aggiungere nuove funzionalità ed ottimizzare il carico computazionale.

4.1.3 Paganelli Alberto

Durante questo progetto ho avuto momenti nei quali pensavo di non riuscire a finire in tempo, o comunque mi preoccupava quella che era l'integrazione dei vari elementi di gioco. Questo è dovuto al fatto che essendo un progetto piccolo ma con tante possibili varianti era interessante renderlo estendibile. Mi ha stimolato molto l'utilizzo di tecniche avanzate poichè sono molto soddisfacenti dal punto di vista dello sviluppo.
Questo è un lato che approfondirò sicuramente poichè credo che la programmazione a oggetti abbia un'enorme potenzialità. Credo che sia stato sviluppato al 70% bene ma sicuramente almeno il 30% del codice presente necessita quasi obbligatoriamente un refactor.
In conclusione posso dire che procedendo dal basso, siamo riusciti ad ottenere una buona struttura di gioco anche se non la migliore. Ritengo il tempo impiegato e reso disponibile dai professori giusto.

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Matteini Mattia

Una prima difficoltà incontrata sicuramente è stata l'utilizzo efficiente di git, secondo me infatti, nonostante le basi imparate a lezione sul DVCS, serve qualche nozione più avanzata, è inevitabile infatti scontrarsi con qualche problema che richiede più conoscenze per essere aggirato. In secondo luogo trovo che questo corso, dato il carico di lavoro e di impegno che richiede, debba essere da almeno 18 crediti, soprattutto se verificato con queste modalità di esame. Personalmente ho dovuto tralasciare altri esami per concentrarmi su questo.
A parte questi aspetti ritengo che questo sia uno dei corsi che mi ha formato di più, e sicuramente mi ha indirizzato verso uno un approccio più professionale della programmazione in generale.

4.2.2 Paganelli Alberto

Credo sia uno dei corsi più interessanti affrontati fino ad ora, chiaramente non è un corso facile, ma mi ha stimolato a studiare la programmazione ad oggetti e soprattuto a vederla come soluzione a molti problemi reali. Ho capito che in una programmazione eccellente il codice sembra scritto in linguaggio naturale e l'uso di pattern di programmazione rende veramente estendibile il software, aspetto secondo me fondamentale in un ottica lavorativa e per garantire un ciclo di vita del software senza riscrivere molte classi o parti di codice, quindi aiutando sia lo sviluppatore che il committente.

Appendice A

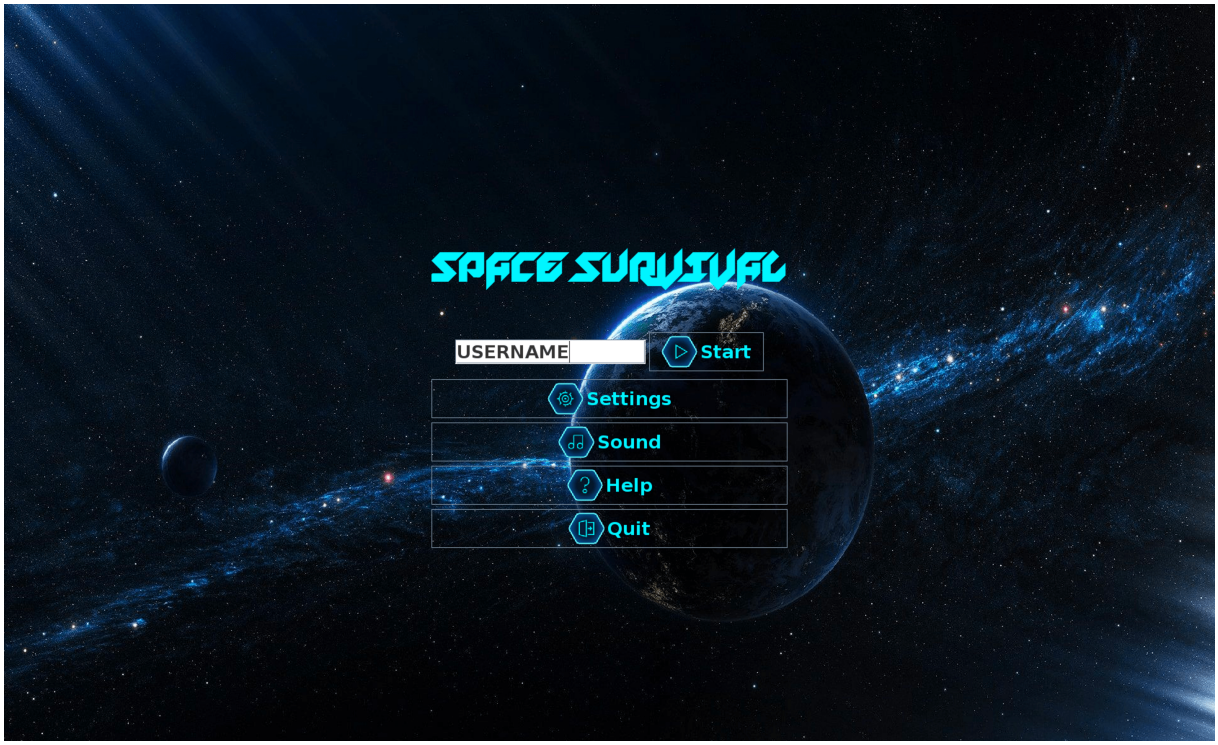
Guida utente

A.1 Avvio del gioco

Eseguendo il comando "`java -jar SpaceSurvival.jar`" verrà aperto il gioco in modalità full screen e effettuato un caricamento iniziale.

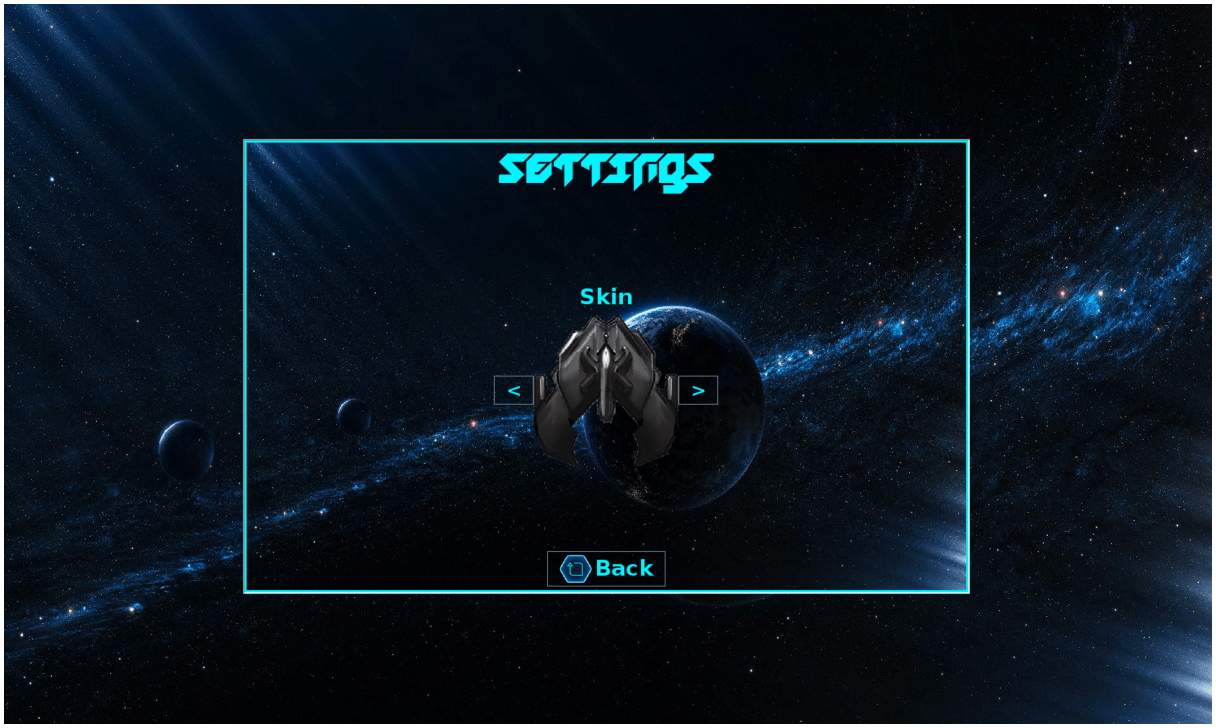
A.2 Menù principale

A fine caricamento l'utente si troverà davanti al menù principale, nel quale ha la possibilità di inserire il proprio username e iniziare una partita.



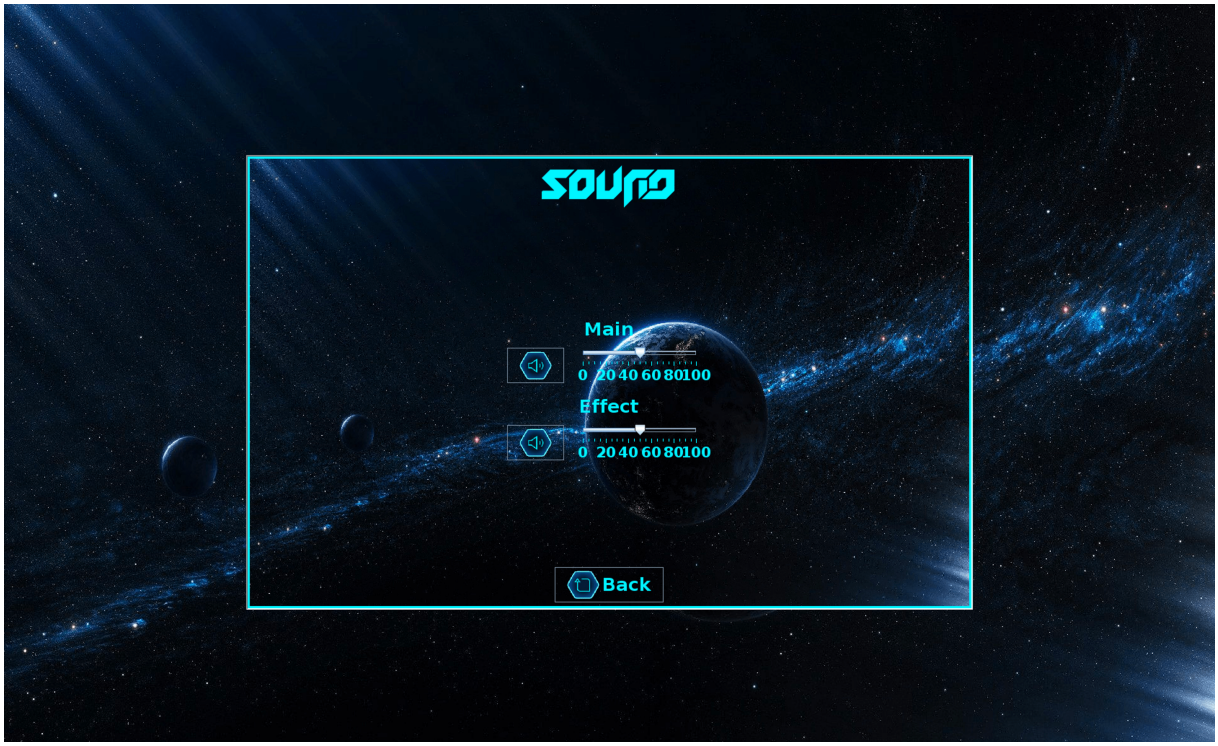
A.3 Settings

Nella sezione Settings l'utente ha la possibilità di scegliere l'aspetto della navicella tra 5 skin predefinite.



A.4 Sound

Nella sezione Sound l'utente può scegliere il volume preferito sia per effetti di gioco che del suono in background.



A.5 Help

Vi è anche una sezione Help nella quale vengono specificati i comandi disponibili nel gioco.

Infine un bottone Quit, per chiudere il gioco.

A.6 Schermata di gioco

L'interfaccia di gioco è molto semplice. Oltre alla navicella, che inizialmente si trova in centro allo schermo, vi è l'HUD che informa l'utente sullo stato attuale della partita. Nell'HUD troviamo:

- **Punteggio**, informa l'utente sul punteggio attuale
- **Timer**, il tempo trascorso dall'inizio della partita
- **Round**, il round corrente
- **Nemici rimasti**, il numero di nemici rimasti per passare il round

- **Tipo di munizioni**, le munizioni attuali sparate dalla navicella
- **Vita rimasta**, vita rimasta alla navicella