



1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

UNIVERSITÀ DEGLI STUDI DI URBINO CARLO BO

Dipartimento di Scienze Pure e Applicate
Corso di Laurea in Informatica Applicata

Relazione del Progetto di
Programmazione e Modellazione ad Oggetti

StarQuest

Muzi Alessio

Anno Accademico 2022-2023 - sessione Estiva

Indice

1	Analisi	1
1.1	Obbiettivo	1
1.2	Requisiti	2
1.3	Analisi e modello del dominio	3
2	Design	4
2.1	Architettura e design globale	4
2.2	Design dettagliato	6
3	Sviluppo	13
3.1	Metodologia di lavoro	13
3.2	Testing	14
3.3	Note di sviluppo	15
4	Appendice	17
4.1	Guida utente	17

Capitolo 1

Analisi

1.1 Obiettivo

L'obiettivo del progetto è lo sviluppo di un applicativo per giocare ad un videogioco chiamato "**StarQuest**". Il titolo del gioco deriva dall'ambientazione del gioco, lo spazio, e dal videogioco StarCraft, da cui riprende aspetto e asset grafici. Altri giochi di ispirazione per StarQuest sono Space Invaders e Gradius. La categoria che descrive meglio questo tipo di giochi, come anche StarQuest, è quella degli **sparatutto a scorrimento laterale**, in inglese **Shoot'em Up**. Lo scopo del gioco è quello di sopravvivere il più a lungo possibile a ondate di vari tipi di nemici sempre più forti e numerosi, cercando di accumulare il più alto punteggio possibile e nel frattempo battere il maggior numero di livelli.

Questa tipologia di gioco raggiunge la sua popolarità massima all'inizio degli anni Ottanta e metà dei Novanta, principalmente nelle sale giochi *arcade*, grazie alla sua semplicità di sviluppo che si rispecchia nelle semplici meccaniche di gioco di cui dispone, alle quali fanno però da contraltare una difficoltà sfidante e capace di generare una simil-dipendenza, anche grazie alle ricche ambientazioni (come per esempio, lo spazio) piene di mostri dal design accattivante. Con il passare del tempo e la nascita di nuovi generi più stratificati, complessi e immersivi e la conseguente quasi-estinzione delle sale con cabinati, questo tipo di giochi è stato relegato al *retrogaming* e a pochissime *software house* che li ripropongono con pesanti modifiche per renderli al passo con i tempi.

In particolare StarQuest si ispira a Lost Vikings, un minigioco presente nel videogioco *StarCraft II: Wings of Liberty* (2010) di *Blizzard Entertainment*, che a sua volta cita nel nome un vecchio *platform* uscito nel 1992 sempre di Blizzard, uno dei primi successi commerciali dello studio americano. Data la natura citazionista della fonte originale, anche StarQuest cita e utilizza alcuni asset grafici del primo *StarCraft* (1998) ma effettua una modifica per quanto riguarda la visuale, passando da verticale ad orizzontale.

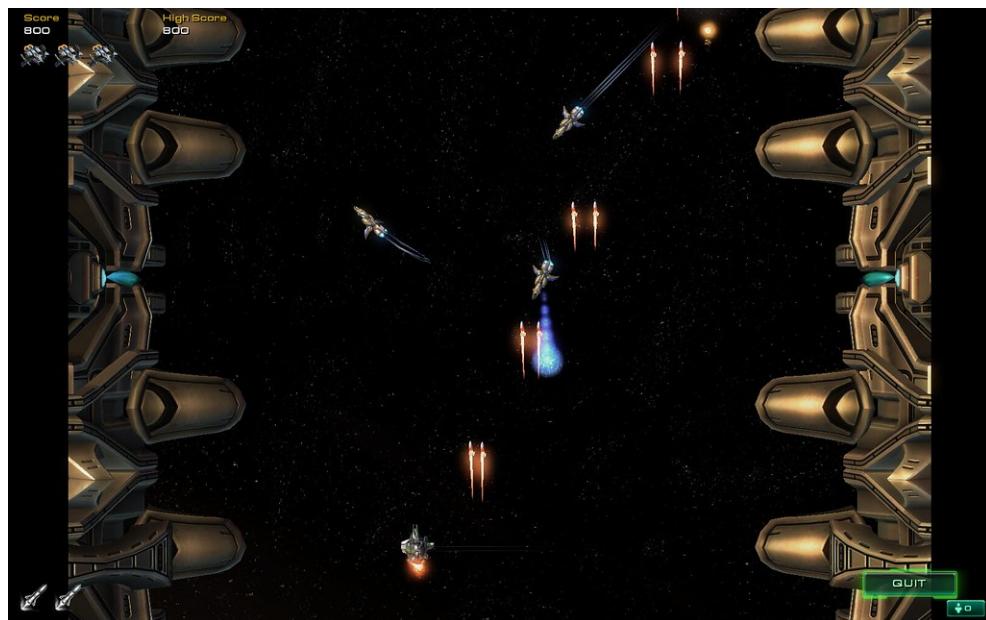


Figura 1.1: Lost Vikings - Starcraft II: Wings of Liberty

1.2 Requisiti

L'applicazione che gestisce il videogioco dovrà essere in grado di:

- Gestire i vari menu' tra cui menu' principale, opzioni e punteggi.
- Controllare in modo fluido l'astronave che rappresenta il giocatore.
- Gestire la generazione dei nemici secondo le varie caratteristiche.
- Gestire la generazione dei livelli e degli asteroidi che fungono da ostacolo.
- Gestire le collisioni tra entità e proiettili e l'apparizione dei power up.
- Presentare a schermo tutte le varie entità usando gli asset grafici adatti.
- Aumentare correttamente, registrare e salvare il punteggio corrente del giocatore e permettere di visualizzare i punteggi più alti.
- Permettere di cambiare le impostazioni, tra cui la difficoltà.

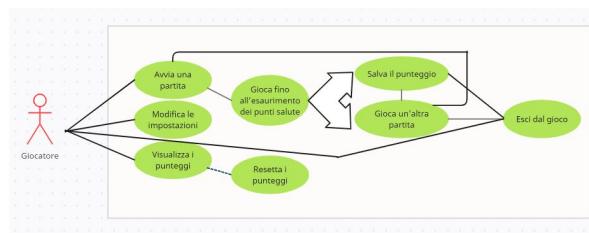


Figura 1.2: Diagramma dei casi d'uso

1.3 Analisi e modello del dominio

Le principali entità presenti nel software sono *Spaceship*, che rappresenta il giocatore, *Debris*, che rappresenta gli asteroidi e altri detriti e *Enemy*, che rappresenta i nemici. Queste entità hanno alcune caratteristiche comuni, gestite da *Entity* e *LivingEntity*. La capacità di sparare è modellata da *Weapon* e *Projectile* e inoltre è presente l'entità *PowerUp* che modella i potenziamenti che il giocatore può raccogliere durante la partita. Per concludere, esiste un'entità che racchiude la logica di gioco: *GameLogic*. In questa interfaccia vengono implementate tutte le regole necessarie al funzionamento del gioco, a partire dall'aggiornamento delle entità presentate a schermo alla gestione delle collisioni, fino ad arrivare alla gestione dei danni, degli attacchi e della salute.

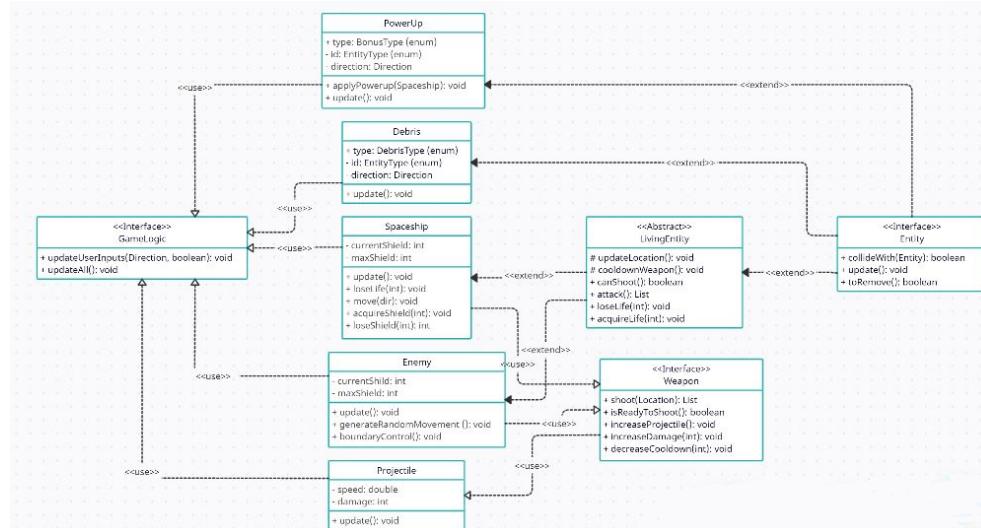


Figura 1.3: Schema UML dell'analisi del problema, con le entità principali ed i rapporti intercorrenti

La difficoltà primaria è quella di gestire le collisioni tra le varie entità in gioco, cioè tra l'astronave del giocatore, i suoi proiettili, i nemici, i loro proiettili, gli asteroidi e i power ups. Esistono altre interazioni rilevanti tra cui: l'aumento del punteggio quando un nemico viene ucciso, il corretto calcolo dello scudo e della vita persa dal giocatore.

Capitolo 2

Design

2.1 Architettura e design globale

Per lo sviluppo dell'architettura di StarQuest è stato applicato il pattern architettonico MVC (Model - View - Controller), in quanto garantisce una chiara suddivisione del codice dal punto di vista logico e permette di separare nettamente la componente grafica dal dominio dell'applicazione; ciò garantisce la possibilità di cambiare completamente l'aspetto grafico senza modifiche ad altre porzioni di codice al di fuori della view e la possibilità di riutilizzare la componente del model per altri applicativi che affrontano lo stesso dominio.

Le entità di gioco individuate nell'analisi del dominio sono definite all'interno del Model. Di seguito vengono descritte le loro caratteristiche principali e il loro comportamento. Il Model ha il compito di raccogliere la logica e le regole del gioco e di definire il comportamento delle sue entità in base a ciò che succede e, seguendo il pattern MVC, non deve interagire direttamente con le altre due componenti (View e Controller). Il Model fornisce attraverso i suoi metodi la possibilità al Controller di accedere ai suoi dati e allo stato attuale del gioco, anche per effettuare modifiche. Le due principali entità sono *Entity* e *LivingEntity*, che modellano una qualsiasi entità di gioco, essa sia passiva o attiva.

Il compito della View è quello di visualizzare su schermo tutte le entità e di fornire all'utente un modo per interagire con l'applicazione. Le classi principali che permettono di rappresentare le entità del gioco a livello grafico sono *DrawEntities* che disegna le entità nella corretta posizione sul background di gioco, *MainMenu* che gestisce il menu' principale e tutti gli altri pannelli come *Scores*, *Options*, *GameOver* e *Title*, *MainWindow* che gestisce la finestra principale di gioco e *GameScreen*, che gestisce la schermata dove il giocatore effettivamente gioca. Inoltre ci sono delle classi ausiliarie come *ClosureHandler* che si occupa della chiusura corretta del software e *InputHandler* che gestisce

gli input dell'utente. La classe *MainView* fornisce i metodi che permettono al Controller di disegnarla e modificarla in base a ciò che dice la logica del gioco. La progettazione delle interfacce grafiche è stata eseguita seguendo le linee guida del framework per Java utilizzato: JavaFX.

Infine il Controller rappresenta il filo conduttore del programma, il componente che garantisce la corretta interazione tra Model e View. Esso ha il compito di utilizzare le informazioni ottenute dal Model per visualizzare correttamente la View e di utilizzare gli eventi che avvengono nella View per modificare il Model. In quest'architettura il cuore del Controller è individuato dall'interfaccia *MainController*, che fornisce i metodi per l'avvio del programma, per metterlo in pausa, per riprendere il gioco, per abortire in caso di errori, per la gestione vera e propria degli input del giocatore e per tutto ciò che riguarda il salvataggio dei dati. Nella classe *GameLoop* risiedono tutte le regole del gioco, compreso il calcolo del punteggio che viene poi passato a *ScoreController*.

Poichè la distinzione tra View, Controller e Model è stata effettuata con cura e applicando i principi della corretta programmazione ad oggetti, è possibile sostituire le classi che compongono la View con facilità, richiedendo pochi o nulli cambiamenti nel Controller. Di seguito un semplice mock-up che descrive l'architettura con un'immagine riassuntiva.

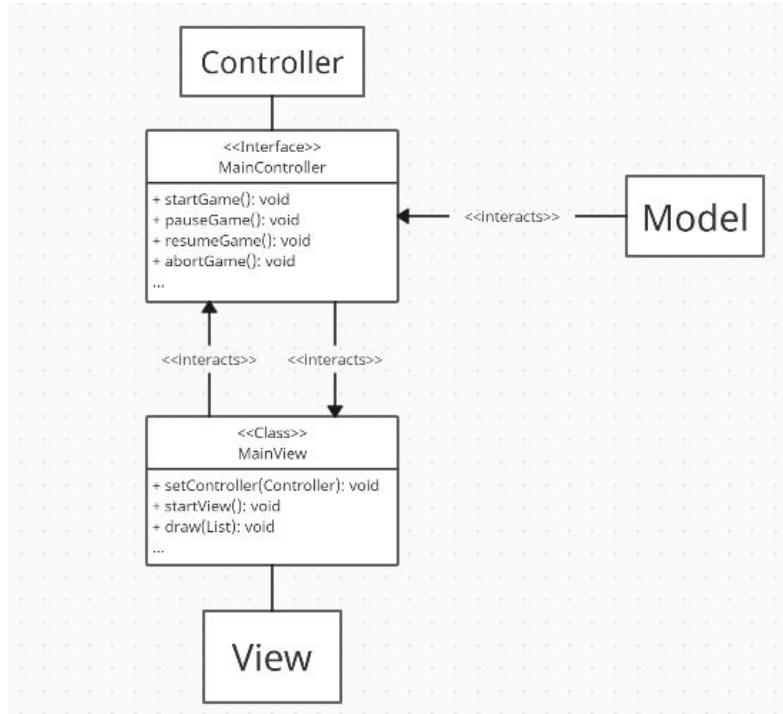


Figura 2.1: Implementazione dell'architettura MVC

2.2 Design dettagliato

Una volta stabiliti i requisiti funzionali minimi e l'architettura dell'intero progetto su più livelli, il passo successivo è quello dell'implementazione delle varie classi che svolgono le funzioni richieste. In questa sezione vengono descritte le implementazioni più significative nel dettaglio, divise per package.

Controller

Il package del controller contiene tutto ciò che agisce da tramite tra View e Model: in particolare contiene il controller principale, denominato *MainController*, e il controller che si occupa dei punteggi, lo *ScoreController*. Inoltre è presente un oggetto chiamato *GameLoop* il quale è un thread che si occupa dell'esecuzione vera e propria della partita. Il *GameLoop* agisce come una macchina a stati finita (indicati nel package `utils.enum`) che gestisce le caratteristiche della partita e il flusso di gioco con i metodi tipici dei thread. Tutto ciò che è contenuto nel metodo `run()` viene eseguito quando inizia la partita.

Per quanto riguarda i controller veri e propri, è rilevante l'uso di librerie per la lettura nella codifica UTF-8 per il salvataggio sicuro dei punteggi (insieme all'utilizzo di adeguate eccezione IO e `IllegalStateException`) su file nello *ScoreController* mentre nel *MainController* avviene la creazione del *GameLoop* e la generazione dell'input parser condiviso tra le classi che ne hanno bisogno, realizzato come un'interfaccia funzionale.

Un'alternativa alla soluzione implementata poteva essere quella di implementare il design pattern Command, il quale incapsula ogni comando possibile in un oggetto, ma ciò risultava peggiorativo delle prestazioni e superfluo poiché i comandi di cui necessita il giocatore sono pochi e gestibili con un albero di decisioni.

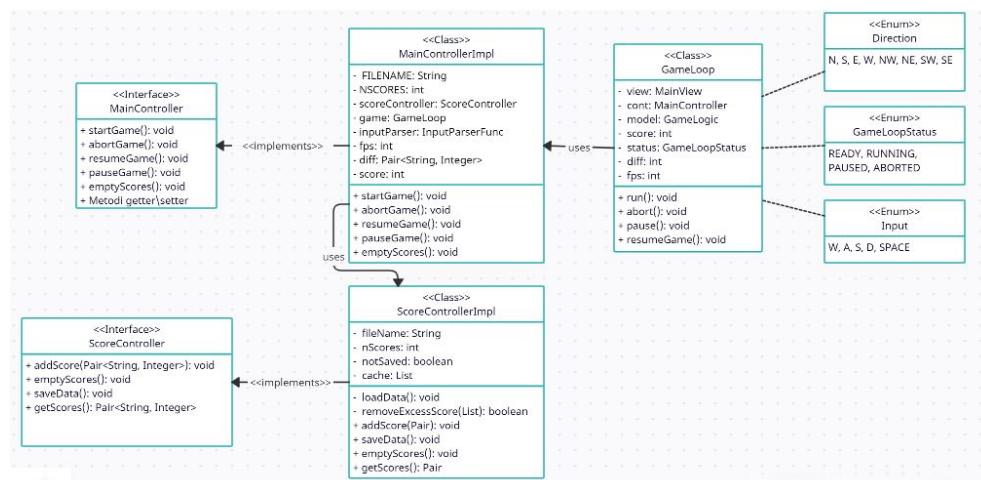


Figura 2.2: Schema UML del package controller

Main

In questo package è contenuta un'unica classe che contiene come unico metodo l'entrypoint dell'applicazione, separato da tutto il resto del progetto semplicemente per mantenere ordine e separazione dei package. Quello che avviene in questa classe è l'istanziazione e associazione di controller e view principale. Successivamente, il software che gestisce il gioco viene avviato.

Model.entities

In questo package vengono raccolte tutte le entità che poi appariranno a schermo: esse sono *Debris*, *Enemy*, *Projectile*, *Weapon*, *PowerUp* e *Spaceship*. Le particolarità dell'implementazione di queste entità risiede nel fatto che alla base delle classi abbiamo un'interfaccia comune *Entity* e una classe astratta *LivingEntity* che mettono a fattor comune alcune caratteristiche come quella di avere una posizione, di essere aggiornabili, di avere o meno dei punti salute, di poter o meno sparare e di calcolare le collisioni con altre entità. Inoltre, la classe *PowerUp* contiene un enum chiamato *BonusType* che specifica i vari potenziamenti e i relativi asset grafici sotto forma di stringhe, così come la classe *Debris* contiene un enum chiamato *DebrisType* che indica il tipo di detrito. Tutte le entità hanno poi metodi specifici come quello che calcola salute e scudi del giocatore. Per quanto riguarda la capacità di sparare grazie all'entità *Weapon* funge da **Factory** per i proiettili sparati dal giocatore e dai nemici.

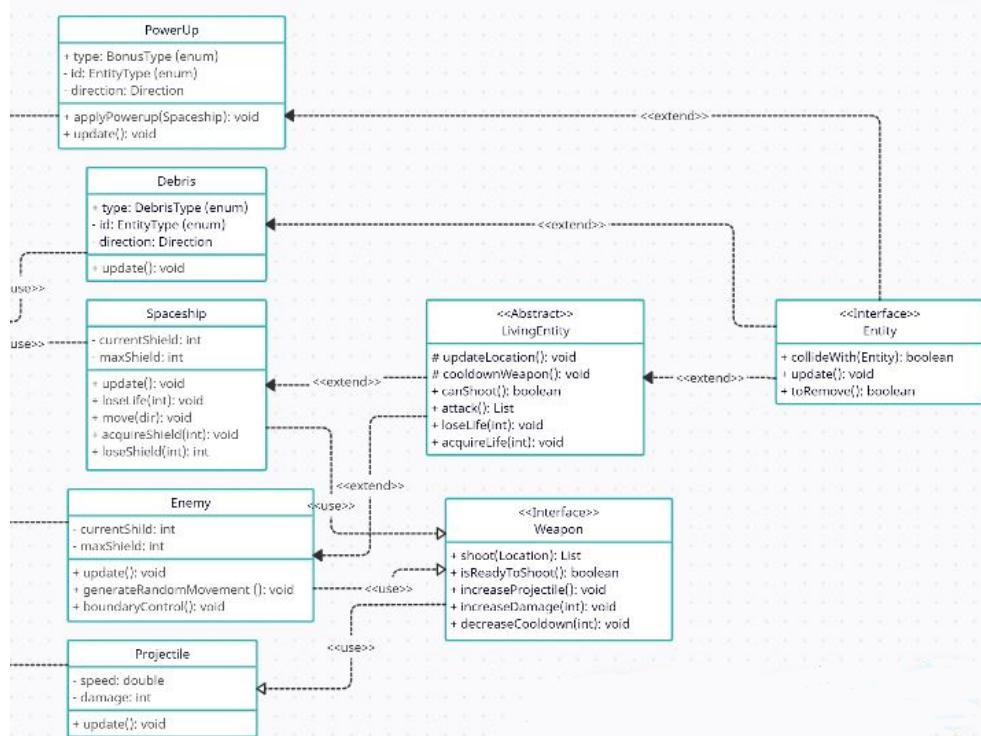


Figura 2.3: Schema UML delle entità

Model.generators

Il package generators è quello che contiene tutte le classi che si occupano di generare le entità a schermo implementate come pattern FactoryBuilder, il quale istanzia tutti i parametri correttamente inizializzati e produce le varie entità Debris, Enemy e Powerup. Inoltre, questi generatori hanno una radice unica in una classe astratta *GeneratorImpl*, il quale implementa l’interfaccia Generator. Ognuno dei generatori implementati dispone di alcuni metodi per settare i giusti parametri e poi un metodo denominato **gen()** che genera costantemente delle entità nei limiti dei parametri sopracitati. Nella definizione dell’interfaccia Generator, il metodo **gen()** utilizza una **wildcard** per permettere ai vari generatori di essere accettati allo stesso modo.

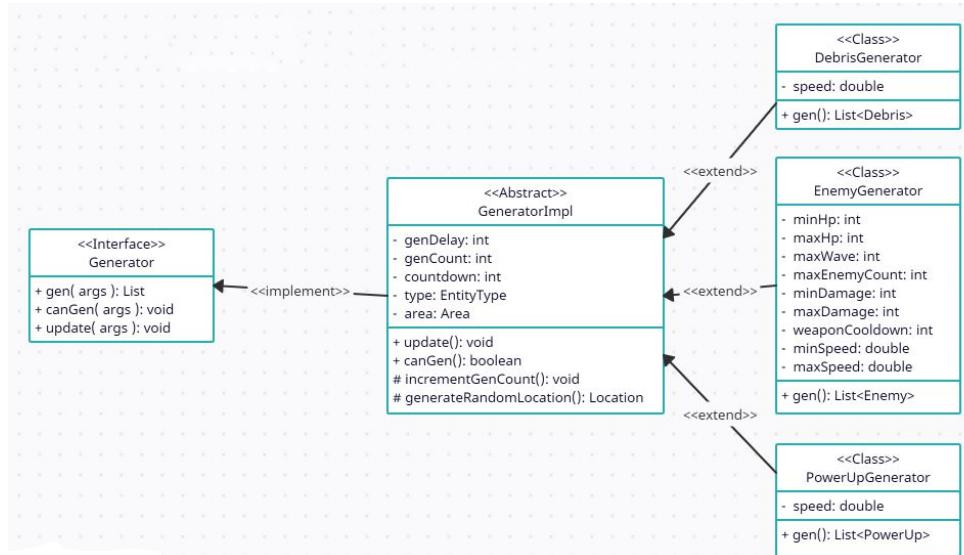


Figura 2.4: Schema UML dei generatori

Model.levels

In quest’ultimo package della sezione Model sono raccolte le classi che riguardano i livelli e le regole degli stessi. I livelli, contenuti nel file sorgente *Level*, sono generati dalla classe *LevelFactory*, il quale applica il pattern Factory. Questo pattern si adatta agevolmente alla situazione presentata poichè i livelli seguono tutti lo stesso schema, le stesse caratteristiche e dopo l’uso vengono eliminati. L’uso di questo pattern permette una continua linea di produzione automatizzata di livelli tramite un metodo chiamato **createLevel()**, il quale dispone di alcuni parametri modificabili come difficoltà, numero di nemici, velocità, danni, salute e tutte le altre caratteristiche e potenziamenti dei nemici presenti nei livelli. Tra le varie caratteristiche dell’entità Level corrente ci sono anche i generatori delle entità presentate nel package precedente.

Per quanto riguarda invece le regole di gioco, sono tutte contenute nella classe *GameLogicImpl*, che implementa l’interfaccia *GameLogic*. Alcuni campi rilevanti sono le liste che contengono tutte le entità attive ed eliminate divise nei vari tipi, lo status della partita in corso, l’ultimo power up preso dal giocatore, il livello corrente, prodotto dalla Factory. il punteggio e l’entità *Spaceship* che rappresenta il giocatore. Inoltre viene usato anche un’enumerazione chiamata *MatchStatus* che raccoglie lo stato della partita attuale: **RUNNING, WON o LOST**. Le regole principali gestite da queste classi riguardano le collisioni, l’aggiornamento di tutte le entità a schermo per quanto riguarda l’aspetto logico e statistico e l’esecuzione dei comandi stabiliti dalla lista degli input del giocatore corrente. Nell’implementazione del calcolo delle collisioni è rilevante segnalare l’utilizzo degli stream e delle lambda espressioni per rendere più agile tutte le operazioni, le quali sono computazionalmente costose e fondamentali per il corretto svolgimento del gioco e il rispetto dei requisiti minimi. Oltre a ciò, è presente anche un metodo privato che controlla se e quando ogni singolo nemico può sparare, seguendo dei timer regolabili e modificabili.

Il motivo per cui vengono tenuti in memoria anche l’ultimo power up preso e la lista delle entità eliminate è per due motivi: il primo è stabilire quando l’astronave del giocatore è arrivato al massimo del livello in quella specifica caratteristica ed il secondo è per tenere conto di quante entità nemiche sono rimaste per completare il livello corrente.

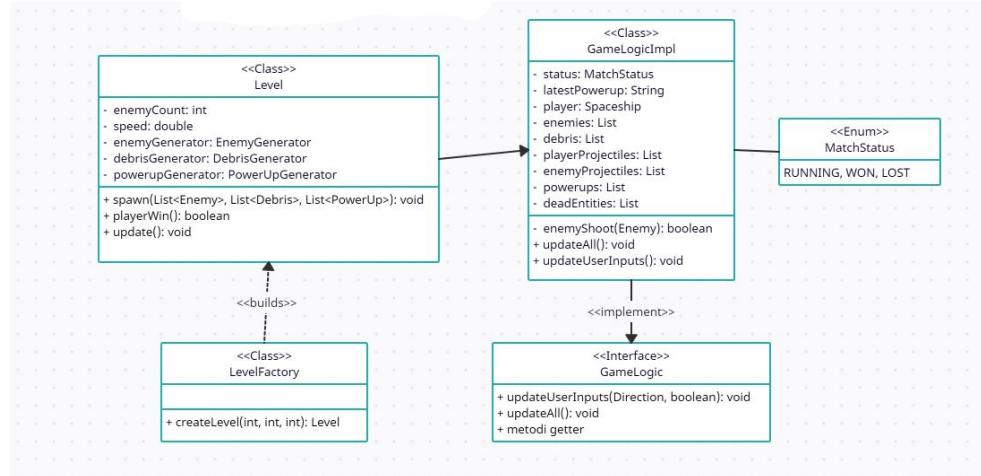


Figura 2.5: Schema UML dei livelli

Utils

Sono stati raggruppati qui i due package di utilità sviluppati per il semplice motivo che svolgono lo stesso compito, ovvero quello di essere ausiliari a varie implementazioni in package diversi. Sono quindi suddivisi in *enums* e *generics* solamente per questione di organizzazione interna e ordine, ma in principio seguono lo stesso scopo. Delle classi sviluppate in questi package è rilevante

segnalare per quanto riguarda gli enum le due strutture ripetute: in *Difficulty* e *Resolution* abbiamo delle coppie stringa-numero e relativi metodi getter e setter, mentre in *Direction* e *EntityType* abbiamo delle strutture con metodi più complessi, come per esempio la corretta associazione tra parametri e immagini dei nemici oppure la gestione degli spostamenti, che le avvicinano a classi vere e proprie. Inoltre sono presenti dei semplici elenchi di keyword che indicano lo stato per quanto riguarda i file sorgente *GameLoopStatus*, *MatchStatus* e *Input*.

Per quanto riguarda il package generico abbiamo invece una raccolta di classi e interfacce funzionali utilizzate in tutto il progetto: in particolare sono implementate una coppia generica *Pair* e due classi che indicano la posizione chiamate *Area* e *Location*. Inoltre sono presenti altre due classi di utilità chiamate *InputParserFunc* e *ImageLoader*, che rispettivamente gestiscono gli input dell'utente e caricano le corrette immagini da associare alle entità su schermo.

View.game

Arriviamo ora all'ultima macrosezione del design dettagliato, quella che riguarda la View e tutta la parte grafica. In questa sezione troviamo molte classi che svolgono singoli compiti ed una separazione che segue nei suoi dettagli macroscopici la suddivisione logica, grafica e funzionale delle interfacce. Ciò vuol dire che una classe gestira il menù principale, una il menù dei punteggi, una la schermata di gioco durante l'esecuzione e così via.

In particolare, nel package view.game troviamo tutte le classi che gestiscono ciò che succede durante la partita vera e propria. La classe principale di questo package è sicuramente *GameScreen*, la quale gestisce tutto ciò che succede a schermo quando il giocatore da inizio ad una partita. Lo schermo viene diviso in varie sezioni che hanno funzioni diverse: nell'angolo in alto a sinistra abbiamo un box orizzontale che contiene salute, scudi e punteggio del giocatore e utilizza vari metodi e descrizioni grafiche per presentare al meglio le informazioni al giocatore. Poi abbiamo la gestione del bottone della pausa, il quale è descritto graficamente nel file *button.css*, in modo da evitare ripetizioni inutili di codice CSS in tutte le altre implementazioni, dato che i bottoni sono alcuni degli elementi grafici più comuni in una qualsiasi applicazione ed in particolare il bottone che ho scelto di implementare in questo progetto utilizza due diverse modalità grafiche (per identificare il cambiamento e restituire un feedback visivo quando viene selezionato con il cursore dal giocatore), un font esterno che aggiunge immersività e dei gradienti di colore molto complessi. Questa classe si occupa anche di mostrare a schermo i messaggi dei power up conquistati dal giocatore e di fine livello una volta sconfitti tutti i nemici: questi testi vengono mostrati a schermo per un tempo prestabilito controllato da un timer interno e generato per ogni scritta definito dal metodo **showText()**. L'ultima funzione

fondamentale di questa classe è quella di dimensionare correttamente tutti gli elementi a schermo in base alla risoluzione scelta nelle opzioni, in modo da poter regolare gli asset grafici su qualsiasi schermo l'utente voglia utilizzare per giocare a questo gioco, tramite il metodo **resize()**

Un'altra funzione molto importante è quella di gestire gli input dal punto di vista visivo, funzione che avviene con l'ausilio della classe *InputHandler*, implementata con il pattern Singleton, in modo che sia presente un'unica istanza della classe in un dato momento. Questa funzione gestisce anche il comando rapido di pausa, attivato dal tasto P. La gestione degli input sfrutta la classe di JavaFX KeyCode.

Un'altra delle classi fondamentali di questo package è la classe *ClosureHandler*, anch'essa implementata con il pattern Singleton e serve a gestire la corretta chiusura del software in modo sicuro, con messaggi personalizzati e richiesta di una doppia conferma da parte dell'utente utilizzatore, per evitare chiusure accidentali.

Le ultime due classi di questo package sono *DrawEntities* e *GameOver*: mentre la seconda è un semplice panello che appare a schermo una volta che il giocatore perde tutti i punti salute e può scegliere se salvare o meno il proprio punteggio e ricominciare un'altra partita o uscire dal software, la prima risulta fondamentale poichè si occupa di disegnare a schermo le entità sul background. Per fare ciò e per dare l'illusione di uno sfondo stellato in movimento, ciò che succede è che vengono caricate due istanze della stessa immagine contenuta nei file di gioco che vengono fatte muovere e alternare tra di loro per non interrompere bruscamente l'illusione e l'immersività dell'ambientazione.

La maggior parte delle classi in questo e negli altri package della macrosezione view dispongono di un metodo chiamato *get()* che oltre a restituire il riferimento allo specifico elemento grafico, gestisce la risoluzione e le dimensioni dello stesso.

View.menu

In questo package sono contenuti due diversi gruppi di classi: il primo riguarda i vari pannelli statici che rappresentano i menù, mentre il secondo raccoglie tutte le implementazioni grafiche sotto un unico endpoint in modo da rendere più facile la comunicazione con il MainController, soprattutto per quanto riguarda gli elementi attivi, in movimento, mutevoli e con necessità di essere aggiornati come punteggi, statistiche, scritte a schermo e entità attive.

View.message

Quest'ultimo piccolo package contiene due classi chiamate *GenericMessage* e *ConfirmMessage* che implementano due semplici box mostrati a schermo in

determinate situazioni all’utente per comunicare qualcosa. Solitamente non c’è bisogno di implementare ulteriori elementi grafici specifici perché le librerie di JavaFX sono fornite di soluzioni efficienti a molti problemi. Nel caso di questo progetto però è risultato utile creare dei box personalizzati con icone che cambiano a seconda del tipo di messaggio e che richiedano in alcuni casi anche una seconda conferma da parte dell’utente, per questo ho scelto di implementare queste due classi grafiche.

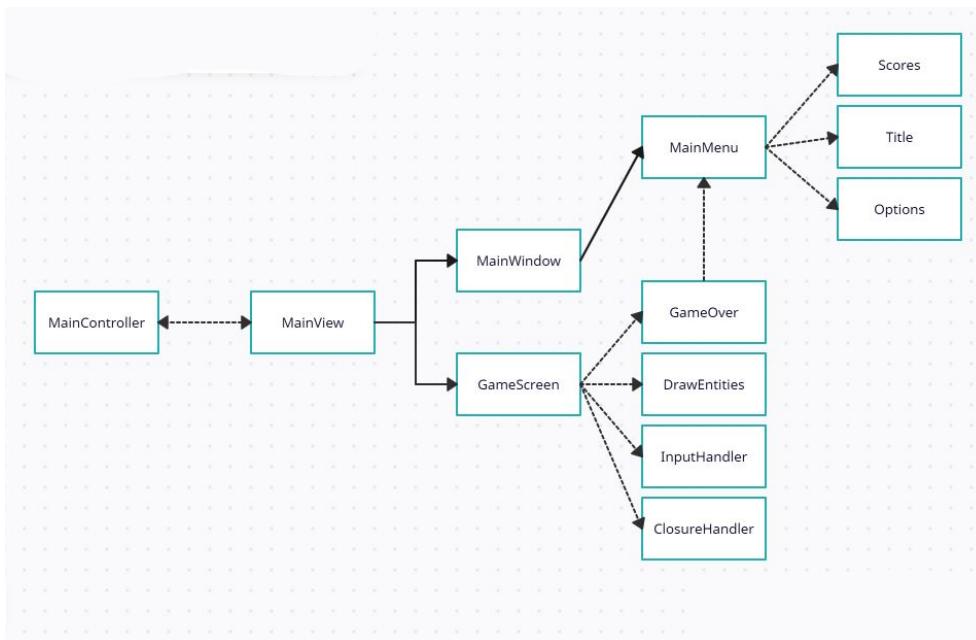


Figura 2.6: Schema UML della View

Capitolo 3

Sviluppo

3.1 Metodologia di lavoro

Trattandosi di un progetto sviluppato singolarmente tutta la progettazione e l'implementazione è stata svolta unicamente da me per cui non vi è stato alcun lavoro di integrazione. Tuttavia ho deciso lo stesso di utilizzare il DVCS come se si fosse trattato di un progetto cooperativo, eseguendo commit frequenti ogni volta che introducevo modifiche e aggiunte sostanziali al codice e effettuando delle push sul repository mantenuto su GitHub, poichè il DVCS scelto per la gestione del progetto è stato Git. Il primo commit in cui viene creato il progetto Eclipse e settate tutte le relative impostazioni è stato effettuato sul branch denominato *main*. Solo quando il codice ha raggiunto la sua versione definitiva è stato aggiunto al branch *main* per eseguirne la release.

Durante il corso dello sviluppo ci sono state delle verifiche della prosecuzione del lavoro a cadenza regolare, principalmente per verificare che i requisti decisi all'inizio della progettazione erano stati rispettati durante l'implementazione e se fosse necessario effettuare modifiche, aggiungere feature essenziali o strutturare delle ulteriori interfacce o classi per organizzare in modo migliore il progetto. Una parte importante della progettazione è stata quella di definire con cura i package che avrebbero contenuto i vari file sorgente sviluppati per soddisfare le richieste, in modo da organizzare in modo chiaro il progetto e fornire un primo livello di documentazione leggibile per altre eventuali persone, ma anche per ordine personale.

L'ultima parte del lavoro nella fase essenziale che è svolto è la scelta degli asset grafici da utilizzare per rappresentare i vari nemici, detriti, l'astronave del giocatore e tutte le altre entità presenti nel gioco, compresi i menu' di scelta e le icone che indicano i vari tipi di messaggi a schermo. Dopo aver scelto questi file, sono state effettuate alcune leggere modifiche alla View per integrare al meglio le immagini nel progetto. Inoltre in questa fase finale della costruzione

dello scheletro essenziale è stato scelto il font da utilizzare ed il *color coding* sia dei vari elementi a schermo, sia della paletta scelta per i vari oggetti grafici.

Una volta conclusa la prima iterazione essenziale del progetto, il problema è stato innanzi tutto confrontato con le specifiche inizialmente dichiarate per verificare che fossero tutte state rispettate e nel caso la risposta fosse stata negativa, quantificare in quale misura. Successivamente ad un risultato positivo, il problema è stato nuovamente analizzato da cima a fondo per definire eventuali funzioni aggiuntive, presenza di nemici variegati, bilanciamento delle statistiche e delle regole di gioco, regolazione delle caratteristiche del giocatore e delle varie difficoltà presentate, anche per eventuali sviluppi futuri al di fuori della consegna attuale. Un'esempio di questo bilanciamento effettuato è stata l'aggiunta di un power up che aumentasse la velocità del giocatore, per regolare l'esperienza e la difficoltà di gioco, cosicchè all'inizio il giocatore sia adeguatamente lento ma senza trovarsi in difficoltà per il numero esiguo di nemici, mentre con il passare del tempo deve necessariamente potenziare la propria nave per resistere maggiormente. Anche il power up del potenziamento delle armi deriva da una riflessione simile, per ovviare al fatto che i nemici si possono spostare sia davanti al giocatore ma anche dietro di esso. Queste ed altre modifiche sono necessarie per fornire un adeguato **senso di progressione**.

Per quanto riguarda la gestione dei vari task, delle feature da implementare e dei vari bug incontrati durante lo sviluppo ma anche dopo la conclusione dei lavori ho fatto utilizzo il sito di project management **ClickUp**, il quale viene principalmente utilizzato per coordinare il lavoro in un team, ma può essere utilizzato anche da una singola persona che deve gestire un progetto personale. In particolare, il fatto che faccia uso del cloud è risultato molto utile poichè non sempre mi sono ritrovato a sviluppare questo progetto nella mia postazione abituale ed ho potuto accedere ai vari task rimasti sospesi anche in posizioni remote.

3.2 Testing

Per il testing è stato usato il framework di unit testing **JUnit 5** seguendo le idee di Test Driven Development e automatizzazione. I test sono stati scritti in modo tale da cercare di "rompere" il software per scovare i problemi ed errori altrimenti difficilmente individuabili. Come principio, nel caso un test non risultasse valido, lo sviluppo si interrompeva alla ricerca attenta del problema incontrato, per poi passare alla risoluzione. Non venivano aggiunte feature e non si proseguiva con lo sviluppo fino a che i test non fossero positivi. Il motivo principale che giustifica tale metodologia di sviluppo è che ad ogni nuovo metodo o feature aggiunta ad una classe è stato possibile verificarne subito il suo funzionamento in modo indipendente. Infatti, nel caso in cui il

test fosse risultato corretto si poteva continuare normalmente con lo sviluppo del programma, ma, soprattutto, se il test fosse fallito era facilmente immaginabile che l'errore dovesse riguardare il segmento di codice appena aggiunto, risparmiando tempo di debug.

Attraverso i metodi forniti da JUnit 5 sono stati scritti i test per verificare se i risultati restituiti da un metodo, una stampa su schermo, oppure una nuova feature, fossero esattamente quelli attesi. Fondamentale è stata la definizione dei casi limite in ogni funzione che veniva testata.

Data la natura del progetto e del software realizzato, potrebbe risultare difficile o poco performante effettuare test di questo tipo, cioè interattiva, dipendente dagli input del giocatore e basata anche su un elemento di randomicità, utilizzando JUnit 5. In realtà, anche se alcune funzioni potevano necessariamente essere testate esclusivamente avviando il software e effettuando modifiche, principalmente tutto ciò che concerne la parte grafica, per quanto riguarda i metodi, le feature e le funzioni della parte logica, JUnit 5 risulta utile nel testare collisioni, punteggi, corretto calcolo dei punti salute, generazione dei nemici e utilizzo delle armi e dei proiettili.

I test prodotti sono raccolti nei file sorgente:

- **TestGame**, che raccoglie i test riguardanti salute, attacchi e collisioni.
- **TestLogic**, che riguarda le funzioni di pausa, ripresa e salvataggio dei dati di gioco.
- **TestScores**, che riguarda il calcolo dei punteggi e la visualizzazione dei risultati migliori.

3.3 Note di sviluppo

In quest'ultima sezione sono elencate le funzionalità avanzate di Java utilizzate per la realizzazione dell'implementazione del progetto:

- **Optional**: sfruttata per evitare l'uso della keyword null nei casi in cui il GameLoop ancora non è stato avviato, la cache dei punteggi salvati ancora non è stata caricata o per segnalare che ancora non era stato preso nessun power up.
- **Stream**: usati principalmente nella gestione delle collisioni, per permettere di gestire al meglio le collezioni di entità presenti a schermo.
- **Lambda Expressions**: usate negli stream per stabilire le caratteristiche degli elementi da filtrare ma anche al di fuori di essi, principalmente

per gestire gli input dell’utente, le azioni dei bottoni, la chiusura corretta del programma e il salvataggio dei punteggi.

- **Generici:** usati nella classe generica *Pair* e uso di **generici bounded** nell’interfaccia *Generator* per gestire la generazione di varie entità.
- **Interfacce funzionali:** per quanto riguarda il parsing e la gestione degli input, è stata utilizzata un’interfaccia funzionale, segnalata dall’annotazione alla JVM **@FunctionalInterface**. L’interfaccia in questione è contenuta nel file *InputParserFunc* ed è poi condivisa tra le classi che necessitano gli input dell’utente.
- **Classi astratte:** usate per modellare al meglio alcuni aspetti nelle classi *LivingEntity* e *GeneratorImpl*.
- **Runnable e Thread:** usati nella classe *GameLoop*, il quale è un thread condiviso da varie classi che regola inizio e fine di una partita di gioco.
- **JavaFX:** è stato il motore grafico dell’applicazione, il quale è molto flessibile ed espressivo. Inoltre, per raccogliere alcuni stili grafici riutilizzati, è stato usato il linguaggio di formattazione **CSS** (*Cascading Style Sheets*), raccolti nel file *button.css*.

Crediti: Gli asset grafici usati per la realizzazione dell’astronave del giocatore, delle tipologie di nemici, del menu’ principale e dei menu’ dei punteggi, delle impostazioni e di game over provengono dai videogiochi Starcraft (1998) e Starcraft II (2010) di proprietà di Blizzard Entertainment ed il loro uso ricade nel diritto del fair use. Tutti gli altri asset provengono invece da librerie free o sono stati realizzati per l’occasione.



Capitolo 4

Appendice

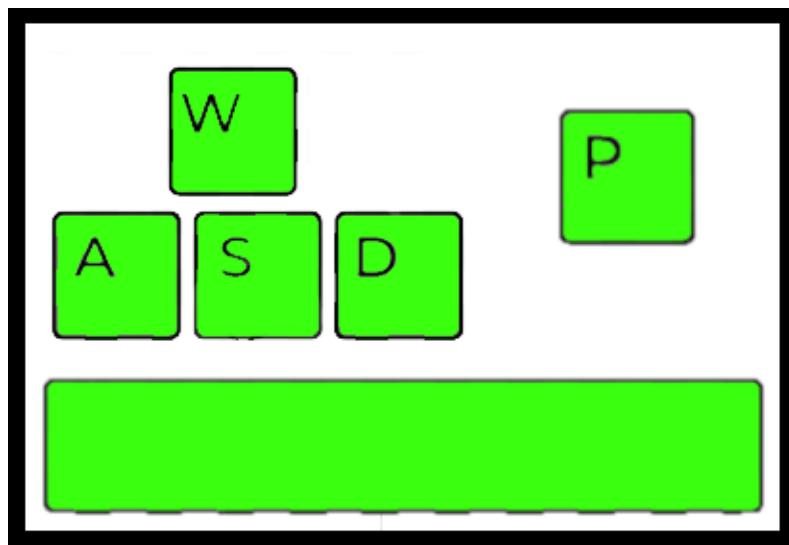
4.1 Guida utente

Come già indicato nelle precedenti sezioni, l'utilizzo del software e soprattutto dei comandi di gioco deve essere semplice e intuitivo.

All'avvio del programma, appare a schermo il menù principali con 4 opzioni: iniziare una partita [1], visualizzare i punteggi [2], cambiare le impostazioni [3] o uscire [4]. Nella schermata delle impostazioni è possibile cambiare risoluzione (se permesso dalle dimensioni dello schermo), difficoltà e Frame Per Secondo. Nella schermata delle impostazioni è possibile visualizzare i punteggi più alti o resettare il salvataggio degli stessi.



Una volta iniziata la partita, i comandi sono semplici: il movimento dell'astronave avviene con i tasti **W**, **A**, **S** e **D**, che rispettivamente spostano il giocatore nelle direzioni su, sinistra, giù e destra, mentre per sparare si usa la **barra spaziatrice/SPACE**. Inoltre, premendo il tasto **P** il gioco viene messo in pausa o riprende se lo era già.



L'obiettivo del gioco è resistere il più possibile e accumulare punteggio, gestendo due risorse: **scudo** e **punti salute**, dove i primi vengono consumati per primi e una volta scesi a 0, iniziano a venire decrementati i secondi.

Una volta che i punti salute arrivano a 0, appare un'altra schermata, quella del game over, che permette al giocatore di salvare il proprio punteggio, visualizzare quelli degli altri giocatori, ricominciare un'altra partita, tornare al menù principale o uscire dal software.