2121 : The Last Man Standing

Matteo Belletti
Luca Cantagallo
Valerio Di Zio
Luca Morlino

25 aprile 2021

Indice

Capitolo 1

Analisi

Il software mira alla creazione di un videogioco survival a piattaforme. Per "survival" (ovvero un videogioco di sopravvivenza) si intende un sottogenere di videogioco d'azione ambientato in un ambiente ostile in cui i giocatori devono sopravvivere il più a lungo possibile. Per "gioco platform", invece, si intende un videogioco in cui la meccanica di gioco fa sì che il personaggio principale, incarnato dall'utente, è libero di muoversi nelle due dimensioni all'interno del mondo del gioco. Nello specifico, questo software ha un mondo ambientato in un immaginario futuro, in una schematica e simpatica realtà post apocalittica (da qui il titolo "2121 – The Last Man Standing"). Il player di "2121" è incarnato da un piccolo soldatino che, munito di tecnologiche armi da fuoco, dovrà sparare e scappare dagli zombie per rimanere in vita. Lo scopo del gioco è collezionare il maggior numero possibile di punti, ottenuti tramite i danni inflitti ai temibili zombie.

1.1 Requisiti

Requisiti funzionali

- Il player ha la possibilità di muoversi all'interno del mondo di gioco nonché la possibilità di sparare per uccidere gli zombie.
- Nel mondo appaiono potenziamenti sotto forma di oggetto che il player può prendere per incrementare e aggiungere delle abilità, o per cambiare arma.
- Gli zombie si muovono all'interno del mondo, alcuni dei quali seguono il player mentre altri si muovono in modo casuale. Al contatto con il

player lo zombie applica un danno.

Requisiti non funzionali

- L'applicazione dovrà essere utilizzabile su: Windows, Mac OS e Linux.

1.2 Analisi e modello del dominio

L'ambiente di gioco ha un terreno su cui si può camminare e saltare. Il player ha una posizione iniziale (al centro) che è uguale ad ogni avvio del gioco e risponderà agli input da tastiera eseguendo vari movimenti e azioni. I nemici appaiono dall'alto ogni quanto di tempo e vagano per la mappa in modo casuale oppure seguendo il player. Le entità vengono individuate nel mondo attraverso una dimensione e una posizione. Quando un nemico entra in contatto con il player quest'ultimo subisce dei danni. Al contrario il player può sparare, con la pressione di un apposito pulsante, infliggendo agli zombie dei danni qualora li colpisca. Se il player finisce tutti i punti vita il gioco termina. Durante la sessione di gioco appariranno dei potenziamenti che il giocatore può equipaggiare, entrandovici in contatto, per avere armi o abilità differenti. L'utente attraverso il menù iniziale può impostare il proprio nome utente, scegliere la mappa su cui giocare e visualizzare i migliori punteggi per ogni mappa. Queste preferenze vengono salvate per eventuali future partite. In caso di un ottimo punteggio il sistema aggiorna la classifica dei punteggi relativa alla mappa appena giocata.

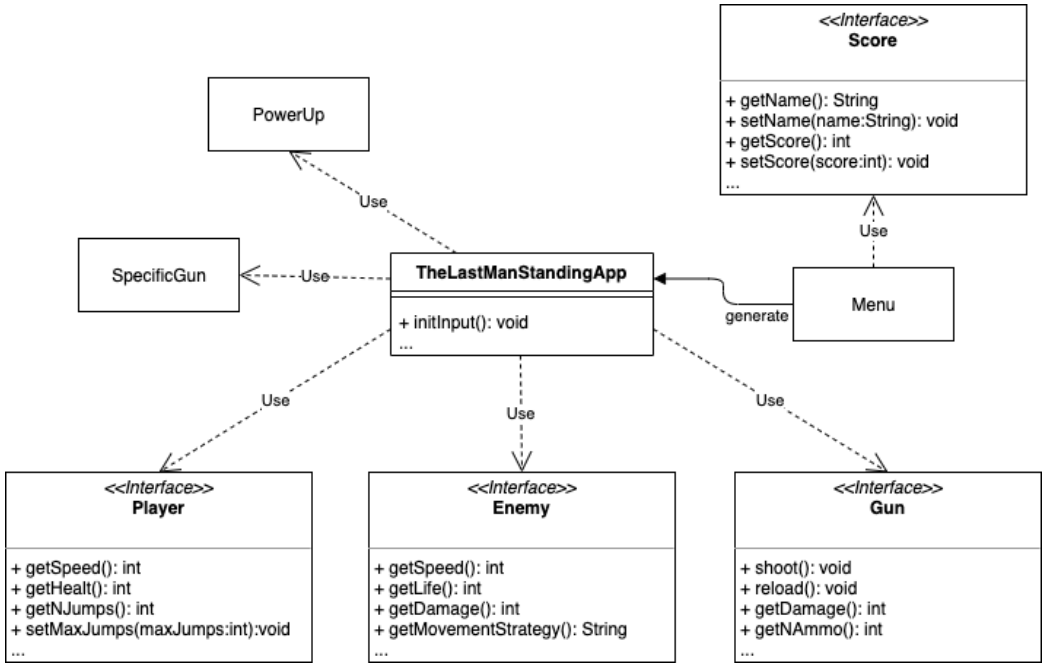


Figura 1.1: Schema UML dell'analisi del problema

Capitolo 2

Design

2.1 Architettura

Per lo sviluppo di "2121: The Last Man Standing" abbiamo optato per una architettura personalizzata (Figura 2.1) che si basa sui concetti di Entity-Component-System e Model-View-Controller (Figura 2.2). Il motivo dell'utilizzo di un pattern personalizzato è stata la struttura monolitica della libreria scelta. Siamo partiti con la creazione del "model" indipendente dal framework. Il "model" definisce lo scheletro delle entità, ad esempio posizione e velocità di movimento per quanto riguarda le entità mobili o altre caratteristiche per quanto riguarda le entità statiche come ad esempio le Hit-Box[1]. Un'entità viene costruita attaccandole dei "component": questi definiscono alcune delle sue funzionalità, ad esempio la capacità di muoversi o la capacità di fare danno. Il "system" tiene traccia di tutte le entità e le aggiorna costantemente: può aggiungerne, rimuoverle o modificarne le proprietà. Per quanto riguarda invece il menù e l'aggiornamento dei record del punteggio abbiamo utilizzato il pattern Model-View-Controller. L'utente interagisce con la "view" del menu che al momento di una richiesta di input/ output come l'aggiornamento dell'username o la visualizzazione della classifica notifica il "controller". Il "model" riceve questi dati e dopo un'elaborazione vengono aggiornati dal "controller" che li restituisce alla "view".

[1]Contorno invisibile di dimensione variabile utilizzato per intercettare le collisioni.

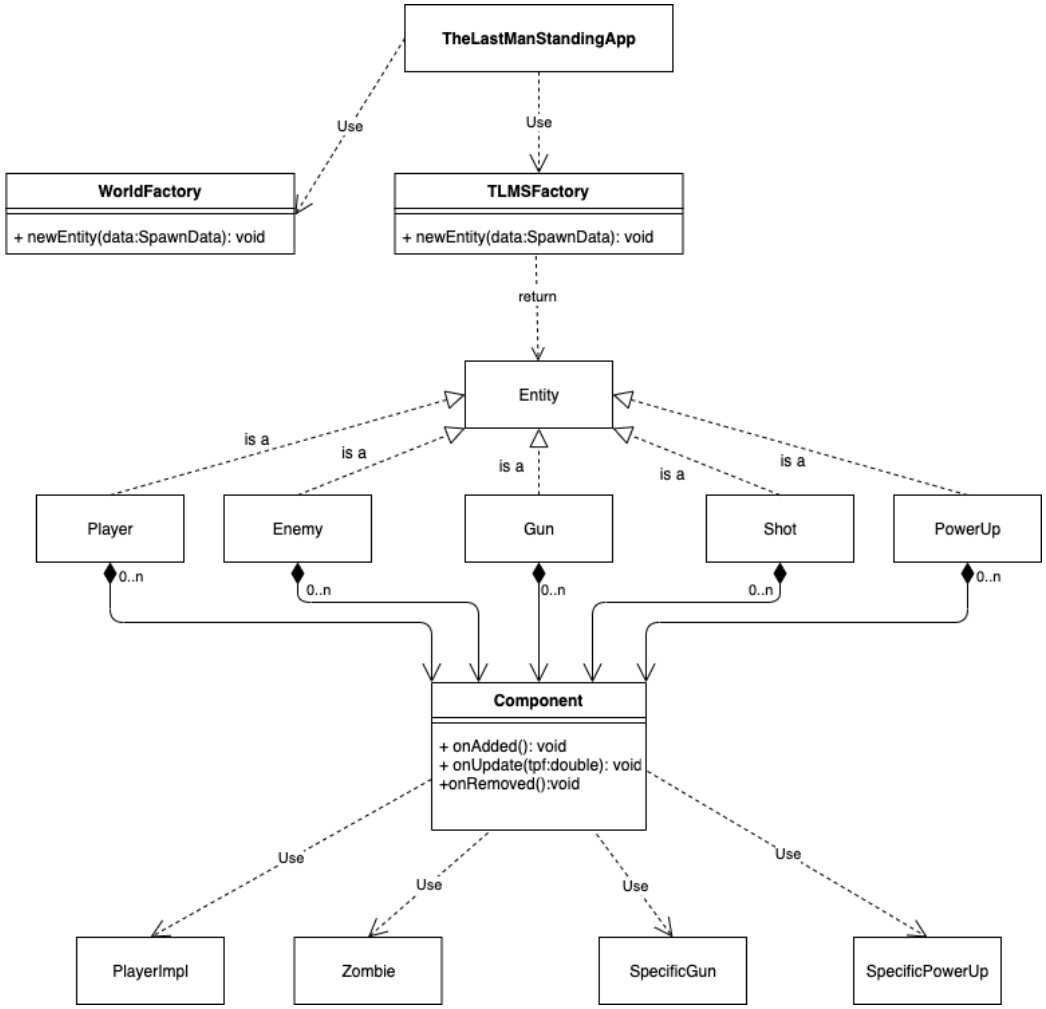


Figura 2.1: Architettura di gioco da noi personalizzata

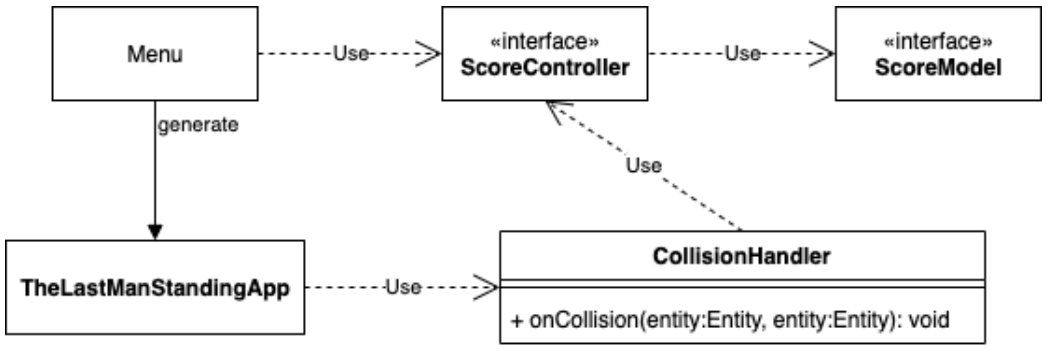


Figura 2.2: Architettura MVC del menù e della gestione dei record

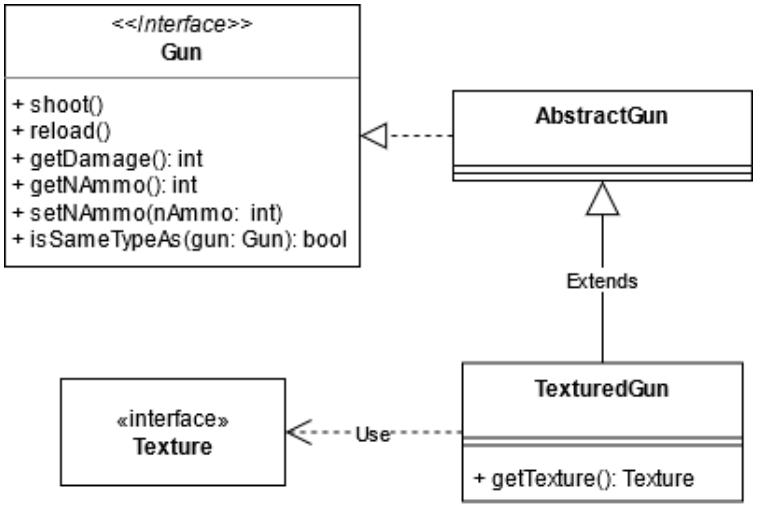
2.2 Design dettagliato

2.2.1 Matteo Belletti

Nella mia parte di progetto mi sono occupato della creazione e gestione dei proiettili e, dunque, delle pistole.
L'intento iniziale era implementare solamente i proiettili, ma procedendo con la progettazione delle classi ho pensato che la gestione di proiettili senza armi fosse parziale, manchevole di una parte integrante. Per questo ho cambiato più volte la struttura del codice, arrivando a definire armi con al loro interno anche le informazioni sui proiettili, con una implementazione il più possibile fedele alla realtà. La logica legata alle pistole è gestita nel package model, dove ho collocato classi indipendenti dalla libreria grafica scelta, mantenendo una divisione funzionale alla preservazione del codice in previsione di eventuali future modifiche.

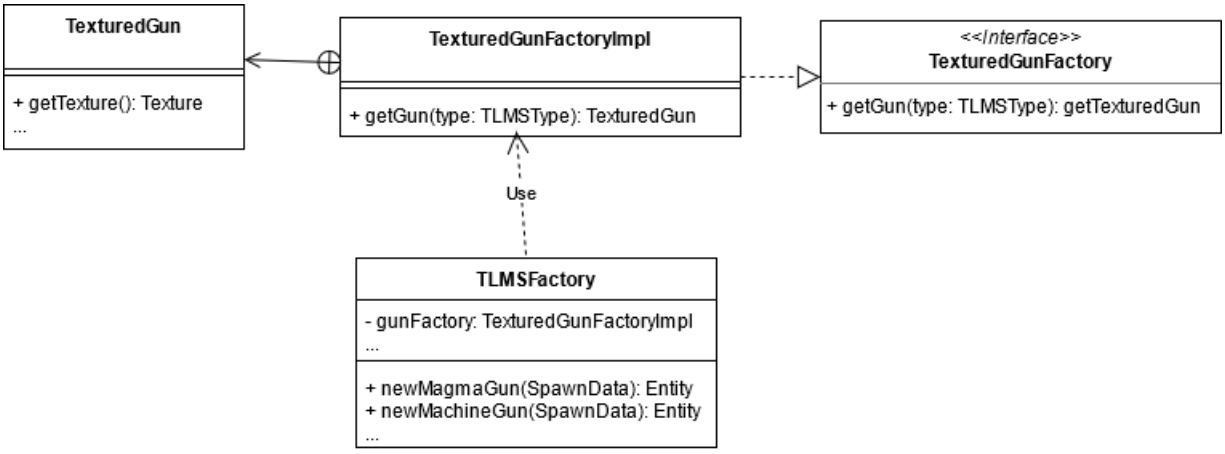
Model

L'interfaccia Gun modella concettualmente il funzionamento delle pistole.
Questa confluisce nella classe astratta AbstractGun, che ne implementa tutti i metodi meno shoot(), lasciato alla singola, specifica implementazione di pistola.
La AbstractGun diventa contenitore di tutte le informazioni che riguardano le munizioni, che possono essere rese illimitate con una specifica implementazione del metodo astratto (è il caso della MachineGun) e i proiettili, di cui memorizza danno e velocità.
Vi è poi la classe astratta TexturedGun che estende la AbstractGun, arricchendola con texture (gestione dei path delle immagini) e divenendo ponte con il package di creazione "factory".
Inizialmente le textures delle armi erano solamente due: quella della pistola e quella dei proiettili; è in vista di una futura estensione che ho integrato una mappa, lasciando la possibilità di aggiungerne di nuove in successive releases.
Nell'arricchimento delle texture ho valutato di applicare il pattern decorator, che ho poi scartato perché l'ho ritenuto inadatto a un contesto di estensione non multipla.



Factory

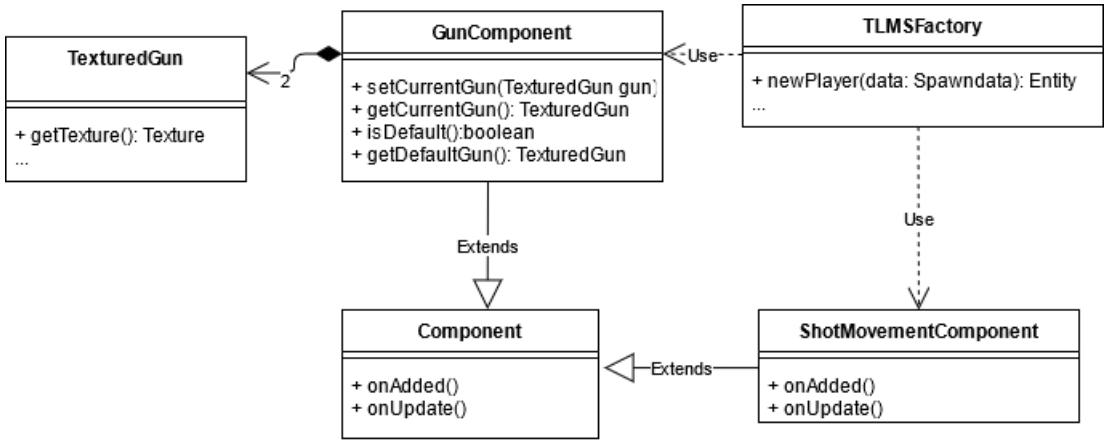
La creazione delle armi di gioco è gestita nel package "factory".
Le istanze di pistole di gioco vengono fornite tramite la classe TexturedGunFactoryImpl, implementazione dell'interfaccia TexturedGunFactory, creata seguendo il design pattern creazionale "Factory". Questa fornisce implementazioni specifiche delle armi utilizzando classi anonime, poi passate al metodo creatore, che restituisce l'arma a seconda del TLMSType passato come argomento.
In questo modo la definizione delle pistole lungo tutto il codice è univocamente decisa nella TexturedGunFactoryImpl, dove nuove armi potranno essere aggiunte con aggiornamenti futuri.



Ho utilizzato il pattern per la sua funzione di incapsulare responsabilità e fase di creazione degli oggetti, isolando l'App dalle classi concrete.
Tuttavia, il punto di unione vero e proprio con la libreria FXGL è in TLMSfactory, che fa uso della precedente Factory di armi per costruire Entities di gioco.

Componenti

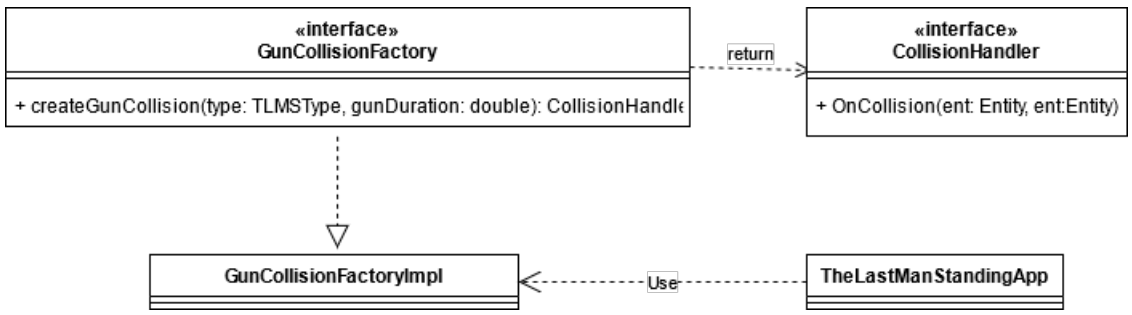
Il player interagisce con le pistole equipaggiabili attraverso la GunComponent. Questa componente tiene traccia dei cambiamenti di arma nel corso del gioco e si occupa della ricarica delle munizioni, azione eseguibile solo se l'arma è assegnata a un'entità.
La GunComponent è un punto di unione potenzialmente riutilizzabile per qualsiasi Entity, rendendo possibile e agevole la creazione di gregari o nemici con armi equipaggiate.



Questa componente, così come la PropComponent, è legata alle entità tramite FXGL, a cui vengono attaccate avvalendosi del builder method.
In questo modo sono rese componibili le unità di gioco, secondo il principio "composition over inheritance", a cui ho personalmente aderito.
Ho creato la classe PropComponent, per facilitare l'aggiunta di ulteriori potenziamenti o oggetti di gioco che sottostiano alle leggi della fisica: è sufficiente passarle una texture per ottenere un'entità, a cui poi assegnare eventuali effetti a seconda della collisione avvenuta.

Collisioni

Collisioni: Arriviamo così all'ultima parte fondamentale della gestione delle armi: le collisioni.



Per la gestione delle armi bonus ho fatto uso del design pattern "Factory" nell'interfaccia GunCollisionFactory. Nella sua implementazione viene resti-

tuita un'istanza di CollisionHandler (il gestore di collisioni di FXGL) diversa a seconda del tipo di arma da equipaggiare.
In questo modo viene delegato il processo di creazione, limitando dipendenze ed evitabili duplicazioni di codice.
Sarà quindi sufficiente creare, nella classe "TheLastManStandingApp", un nuovo collisionHandler con la tipologia di arma come parametro, per farsi che questa venga equipaggiata al player.
Oltre alla pistola, nella factory di collisioni, ho richiesto che venga passato un double per indicare la durata della stessa come equipaggiamento.
Nel gestire l'assegnazione e la rimozione a tempo, mi sono imbattuto in problemi di concorrenza, che ho risolto dopo numerosi tentativi, tenendo traccia di cambiamenti multipli di pistola nel gestore di collisioni.

2.2.2 Luca Cantagallo

La parte da me svolta si è incentrata sulla gestione del player, dell'input, dei power-up e delle implementazioni delle collisioni riguardanti il player. All'inizio di una nuova partita viene creato un player. Questa entità è rappresentata da diverse componenti inizializzate al momento della "costruzione" del player. Il primo componente che cito è il "PlayerComponent" che ha il compito di ricevere delle richieste e tradurle in azioni da fare svolgere al player. Le richieste provengono dalla classe "TheLastManStandingApp" che intercetta gli input da tastiera e lo notifica a questo componente, che lo traduce in un'azione da fare svolgere al player. (Figura 2.3) Questo componente gestisce la parte più logica dell'entità player e interroga il model circa i parametri veri e propri del player per reagire di conseguenza. Ad esempio se deve muoversi a destra, prima di farlo chiederà al "Player" il valore della velocità del player con cui dovrà spostarsi, e infine si sposterà con quella velocità.

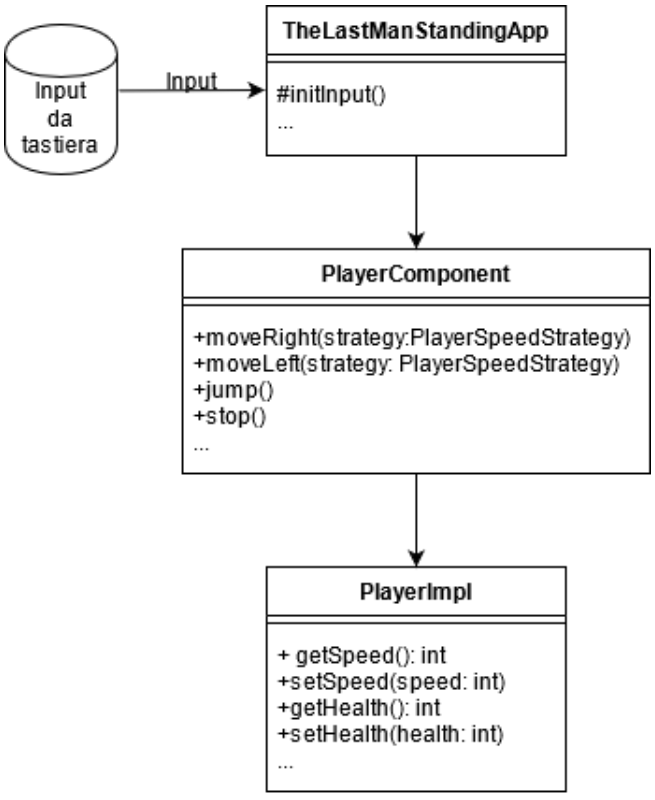


Figura 2.3: Input da tastiera e comportamento assunto dal player

NOTA: In realtà, oltre ai comandi da tastiera, anche nella classe della collisione tra il player e gli zombie è richiamato un metodo del PlayerCom-

ponent, che ha la funzione di fargli accusare il danno.

Il "TextureComponent" ha il compito di assegnare al player un'immagine specifica, date le circostanze in cui si trova. Ad esempio, se il player sta saltando, fa in modo di mostrare l'immagine del player che salta. Poi ci sono altri due componenti forniti da FXGL: "CollidableComponent" e "PhysicsComponent". Il primo fa sì che il nostro player possa collidere con altre entità collidabili, e il secondo fornisce al nostro player alcune leggi fisiche, tipo la gravità. Per la costruzione e la creazione effettiva del player con tutti questi componenti mi sono servito della "TLMSFactory", una classe utile alla creazione e costruzione di entità assemblando più componenti attraverso un pattern builder. Sottolineo di non aver progettato io in prima persona questa classe: mi sono limitato a utilizzarla per la creazione del player e dei power-up. Grazie a questi componenti il player potrà muoversi liberamente all'interno del mondo di gioco. I movimenti base principali del player sono la corsa e il salto.

C'è da approfondire un aspetto della "corsa" del player (o meglio: del suo "movimento orizzontale"). A progetto quasi terminato mi sono infatti reso conto di un piccolo problema dovuto a questo movimento: non era possibile fare girare il player verso la direzione opposta, facendolo rimanere però fermo nello stesso punto. Questo accade perché intrinsecamente il player dispone di una velocità che viene azionata non appena iniziata la pressione di un tasto della tastiera. Ho quindi scelto di differenziare due diversi tipi di movimento orizzontale: il movimento vero e proprio, e la semplice volontà del player di voltarsi dall'altra parte. Per risolvere questo problema e integrare questa nuova funzionalità mi sono affidato al pattern Strategy (Figura 2.4). Questo pattern infatti mi ha aiutato a decidere in un secondo momento la velocità da assegnare al player, facendola dipendere dal tipo di input. Infatti, se voglio che il player si giri senza muoversi, la sua velocità di movimento alla pressione di un tasto dovrà essere zero. Ho implementato quindi il pattern Strategy partendo dall'interfaccia "PlayerSpeedStrategy" e estesa successivamente da due classi "PlayerSpeed" e "PlayerSpeedTurnsAround". Viene creato quindi un nuovo strategy, in grado di restituire il valore della velocità, dipendente dal tipo di input, a ogni pressione di un tasto per il movimento orizzontale. Inoltre, creare questo pattern e tirare fuori la velocità dal movimento, trovo che sia molto utile anche in caso di eventuali estensioni future, potendo assegnare diverse velocità a fronte, ad esempio, di differenti input.

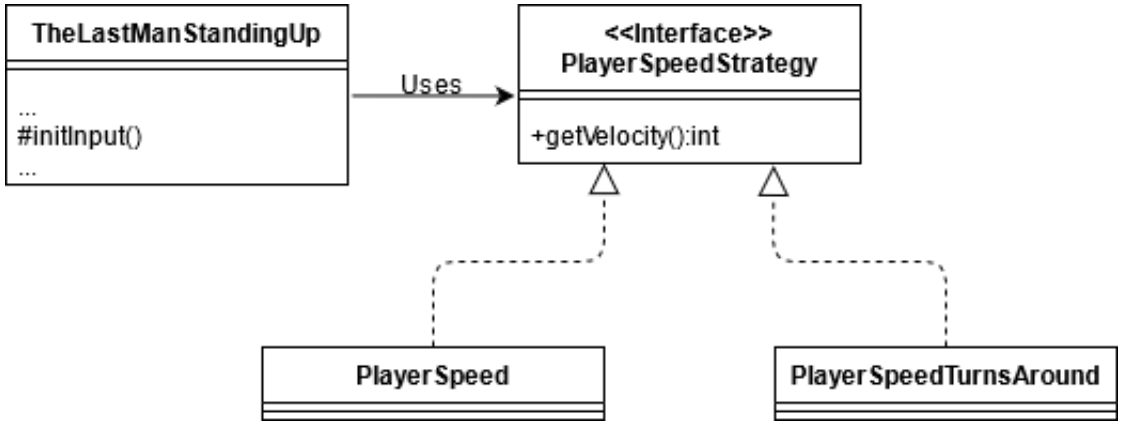


Figura 2.4: Gestione della velocità tramite il pattern Strategy

Interessante è il concetto di power-up: ho creato piccole e semplici sfere che cadono dall'alto. Una volta entrate a contatto con il player, sono in grado di fargli eseguire una "trasformazione". La trasformazione consiste nell'incremento di molti parametri del player, tra cui la velocità, più salti consecutivi a mezz'aria, il ripristino della salute, e una nuova abilità che consente al player di atterrare più velocemente. La trasformazione si manifesta sottoforma di cambio di colore del player. Se il player trasformato viene però colpito da uno zombie, questo torna ad avere le caratteristiche iniziali (il colore, la velocità, il numero di salti,...). La trasformazione avviene quindi sempre a fronte di una collisione: o con uno zombie, o con un power-up. A questo proposito ho richiamato il metodo della trasformazione nelle collisioni del player. Tra player-zombie accade infatti una "detrasformazione" (oltre che un decremento della vita), mentre tra player-power-up avveniva una nuova trasformazione. Inizialmente sussisteva il problema che veniva effettuata la trasformazione ogni qual volta che il player collideva: anche se prima della collisione era già trasformato si trasformava nuovamente, e viceversa. Questa problematica l'ho gestita introducendo un surrogato con il compito di controllare, gestire e limitare gli accessi al metodo della trasformazione. Per fare ciò mi sono servito del pattern Proxy (Figura 2.5). Ho creato un'interfaccia "PlayerPowerUp" con il metodo della trasformazione, e da questa ho implementato due classi: "PlayerPowerUpProxy", "PlayerPowerUpImpl". "PlayerPowerUpProxy" al suo interno ha un metodo privato che controlla se è necessario o meno eseguire la trasformazione, confrontandolo con lo stato del player precedente alla collisione. Ha inoltre un riferimento a PlayerPowerUpImpl, dove è effettivamente implementata la trasformazione, che viene richiamato dal Proxy solo se ritenuta necessaria. Così facendo la collisione rimane totalmente allo scuro del controllo dello stato del player e viene

quindi evitato di sporcare la collisione con metodi per la trasformazione e controlli vari. Inoltre questo sistema può essere esteso, volendo, anche a più trasformazioni.

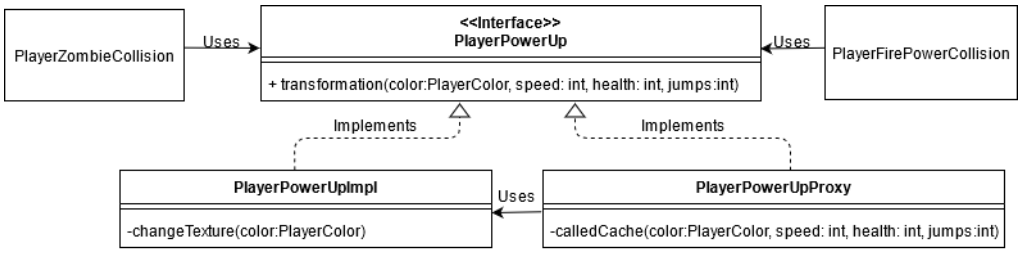


Figura 2.5: Gestione della trasformazione del player tramite il pattern Proxy

Come accennato precedentemente, a seguito di un attacco da parte di uno zombie, il player oltre che perdere l'eventuale trasformazione, perde anche punti vita. Se i punti vita scendono ad un valore minore o uguale a zero il player muore e il gioco termina.

2.2.3 Valerio Di Zio

Uno zombie è una entità (Enemy) che ha la possibilità di muoversi all'interno del mondo con una determinata velocità, ha una vita e può arrecare danno al player. Lo zombie ha una texture, definita dal genere, con cui viene visualizzato all'interno del gioco. Attraverso il pattern "Decorator" (Figura 2.6) è possibile decorare uno zombie con un particolare tipo di texture e di modificare le caratteristiche di danno e velocità. Dato che la maggior parte delle entità presenti nel gioco utilizzano texture, ho creato una classe Texture che ne permette la gestione.

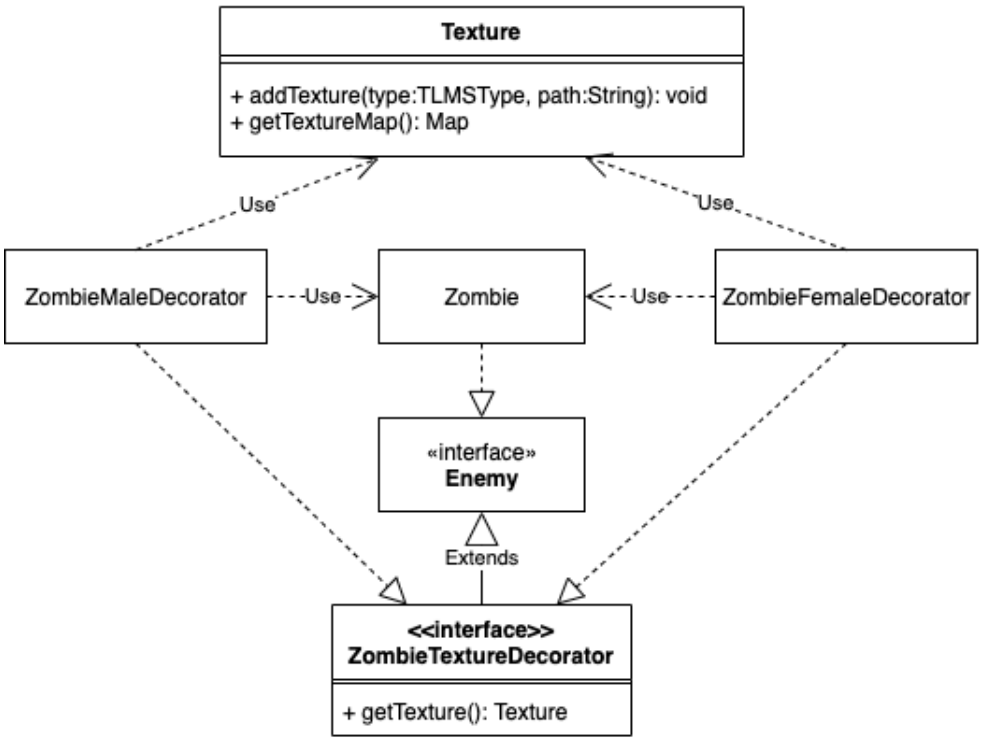


Figura 2.6: Descrizione funzionamento Zombie Decorator

La creazione di uno zombie è affidata a due metodi della classe TLMSFactory che utilizzano il pattern "Builder" per creare uno zombie "stupid" che si muove casualmente e uno "following" che insegue il player durante il gameplay: l'utilizzo del pattern permette la creazione step-by-step delle entità. Nello specifico a uno zombie è associato un tipo, è soggetto alla gravità e ha delle Hitbox utilizzate per gestire le collisioni; per quanto riguarda i componenti (Figura 2.7), ne ho creati svariati personalizzati:

- DamagingComponent che arricchisce un'entità con la possibilità di fare danno con uno specifico valore;
- RandomMovementComponent che, se assegnato ad una entità, la fa muovere in modo casuale all'interno del gioco;
- FollowPlayerComponent che, aggiunto all'entità, la farà muovere seguendo il player;
- ZombieTextureComponent che è utilizzato per l'assegnazione di una texture.

Lo zombie, inoltre, ha due componenti fornite dalla libreria FXGL, "HealthIntComponent" che rappresenta la vita di quest'ultimo, e "CollidableComponent" che indica se l'entità reagisce alle collisioni. I due component responsabili di definire uno zombie "stupid" o uno zombie "following" sono rispettivamente RandomMovementComponent e FollowPlayerComponent.

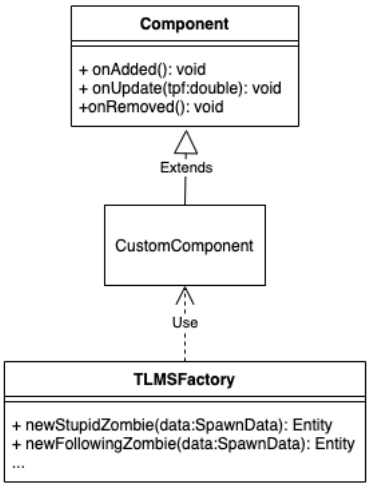


Figura 2.7: Descrizione funzionamento "component"

Per gestire l'apparizione degli zombie all'interno del gioco ho scritto una classe ZombieSpawner (Figura 2.8) che estende Thread che si occupa di generare degli zombie con:

- statistiche casuali, all'interno di un intervallo definito, di danno, velocità e vita;
- di un particolare genere;

- un tipo di movimento “stupid” o “following”.

Il pattern “Strategy” astrae la scelta del tipo di zombie da creare in base al suo tipo di movimento.

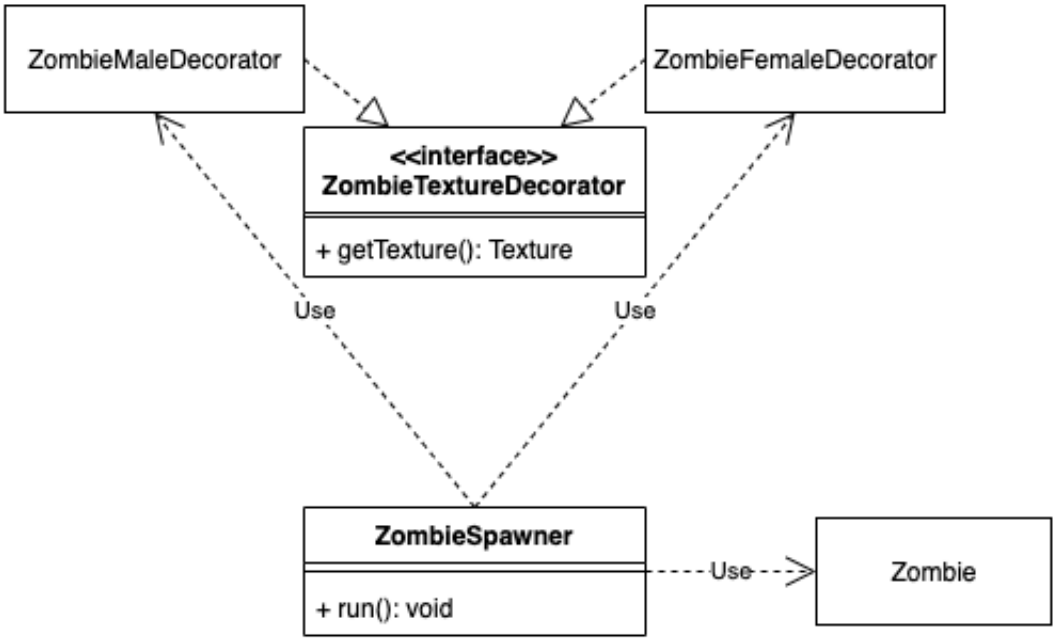


Figura 2.8: Descrizione funzionamento sapwner di zombie

La gestione delle collisioni, è stata implementata da tre nuove classi (Figura 2.10) che definiscono routine di gestione:

- PlayerZombieCollision: tra player e zombie che sottrae vita al player e che ne può causare la morte quando arriva a zero;
- ZombieWallCollision: tra muro e zombie che permette a quest’ultimo di saltare i gradini quando entra in contatto con un muro.
- ShotZombieCollision: tra zombie e proiettili che sottrae vita allo zombie e che ne può causare la morte quando arriva a zero;

I miei colleghi hanno aggiunto alle precedenti collisioni le loro specifiche implementazioni (Figura 2.9).

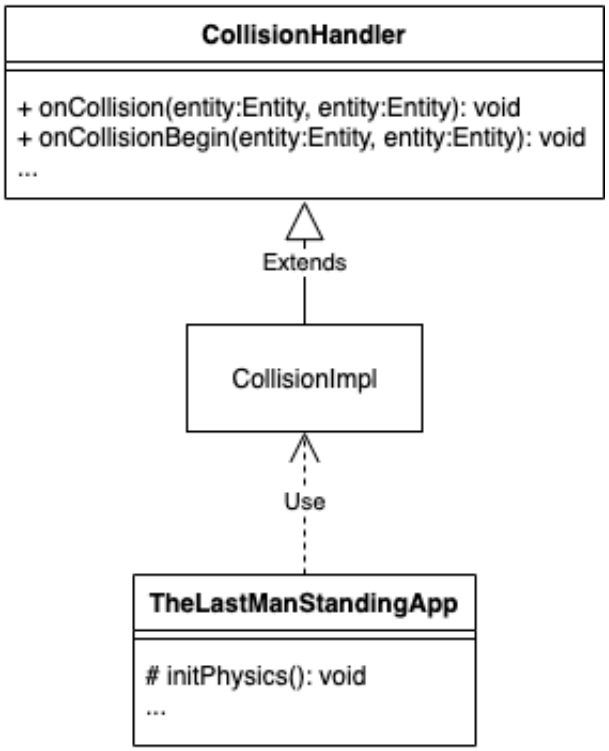


Figura 2.9: Descrizione funzionamento collisioni generico.

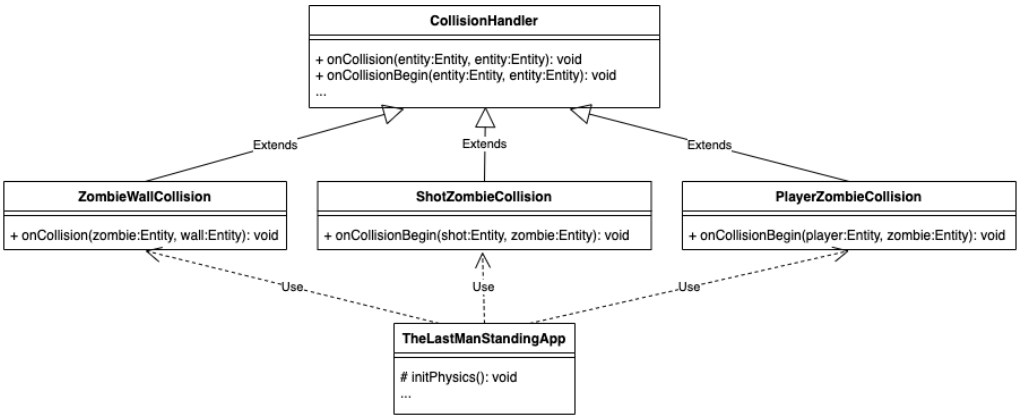


Figura 2.10: Descrizione funzionamento collisioni specifiche zombie.

2.2.4 Luca Morlino

Menù

Il framework FXGL avrebbe una propria gestione del menù e dei punteggi molto pratica e semplice che però non permette di utilizzare pattern architetturali tali da averne un uso indipendente dall'applicazione. È stato quindi disabilitato il menù di default e ne è stato sviluppato uno che all'occorrenza potrebbe offrire le stesse funzionalità fornite a "2121: The Last Man Standing" ad un qualsiasi altro gioco platform a punti. Il menù è stato costruito in modo tale che la scelta di una diversa libreria grafica porterebbe alla riscrittura della sola **View** in quanto i controller di gestione della classifica, mappe e user name ne sono totalmente indipendenti. In figura (Figura 2.11) viene mostrato lo schema UML delle interazioni della **View** del menù con i controller. L'utente ha la possibilità di scegliere a proprio piacimento il proprio user name e la mappa in cui giocare. I controller rispettivi svolgono il compito di creare i rispettivi file all'interno del pc al primo avvio di gioco e di scriverci le impostazioni scelte per le partite future. La visualizzazione delle classifiche delle diverse mappe è possibile grazie allo `ScoreController`, i cui funzionamenti verranno spiegati nel dettaglio in seguito.

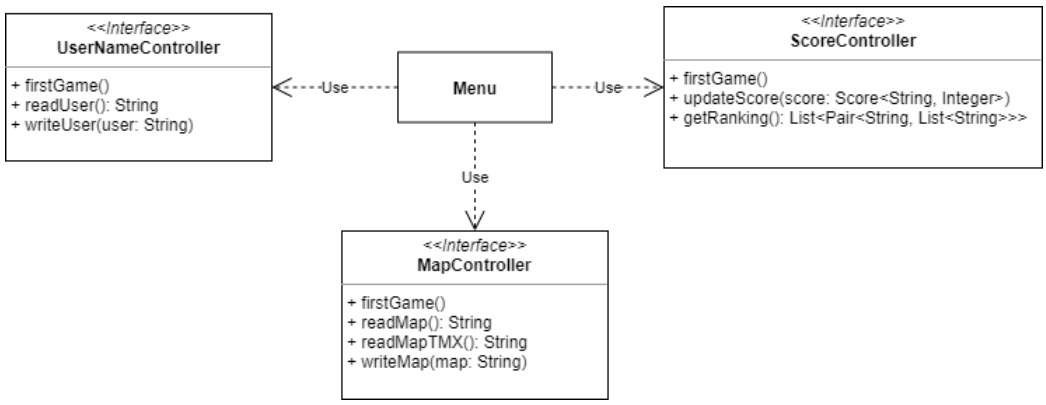


Figura 2.11: Il menu interagisce con i vari controller per soddisfare le richieste dell'utente

La View del menù è stata implementata utilizzando la libreria Swing, il framework nativo di Java. Per ottenere un'immagine sullo sfondo ed avere una visione più accattivante per l'utente è stato necessario estendere la classe `JPanel` con una classe `JPanelWithBackground`, classe in cui è stato utilizzato il pattern Static Factory per la creazione dei JPanel a seconda dell'argomento inserito, è stato inoltre necessario sovrascrivere il metodo **paintCompo-**

nent(). L'utilizzo di un `JPanel` con sfondo ha condizionato un intelligente uso di un LayoutManager per i pulsanti. Si è costruito quindi un proprio setting di pulsanti ma si è cercato di rimanere indipendenti da una qualsiasi immagine scelta come sfondo grazie ad un resetting automatico.

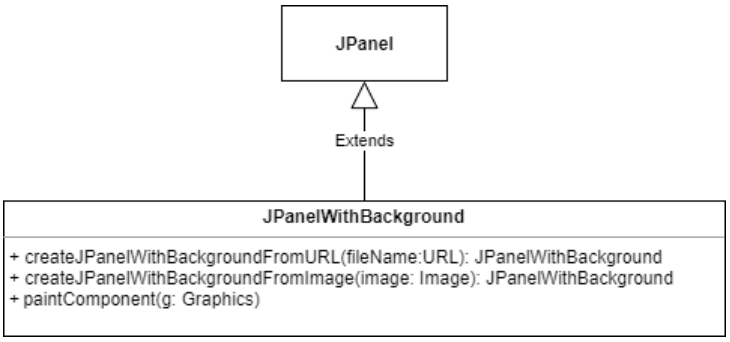


Figura 2.12: La classe JPanelWithBackground estende JPanel sovrascrivendo paintComponent().

Score

Per la gestione della visualizzazione e l'aggiornamento delle classifiche di gioco ho utilizzato un pattern **MVC**. La componente view è il menù citato precedentemente, nel caso di richiesta della visualizzazione della classifica. Lo `ScoreController` viene istanziato anche dalla classe `PlayerZombieCollisions` in quanto quest'ultima controlla la vita residua del player e in caso di morte notifica il controller dell'eventuale aggiornamento della classifica. Lo `ScoreController` viene notificato di un nuovo punteggio ottenuto da un dato utente. Questi chiede allo `ScoreModel` se lo Score ottenuto rientra nei primi tre relativi alla mappa giocata e in tal caso di ottenere la nuova classifica sottoforma di lista. Sarà il controller stesso ad occuparsi della scrittura e lettura su file.

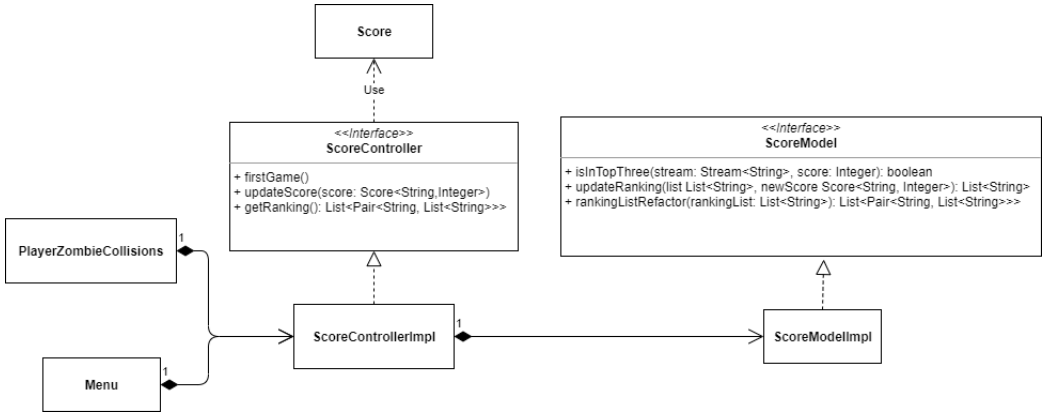


Figura 2.13: Il menu interagisce con i vari controller per soddisfare le richieste dell'utente

Il formato scelto è stato il JSON. La classe `JSonScore` costruisce un oggetto che tiene traccia del nome utente e del suo punteggio. `JSonScore` implementa un interfaccia generica `Score` che permette un proprio possibile riutilizzo con altri tipi di dato purché rispetti l'ordine (nome utente, punteggio). Per una più facile e intuitiva costruzione di un `JSonScore` è stato scelto di utilizzare il pattern **Builder**.

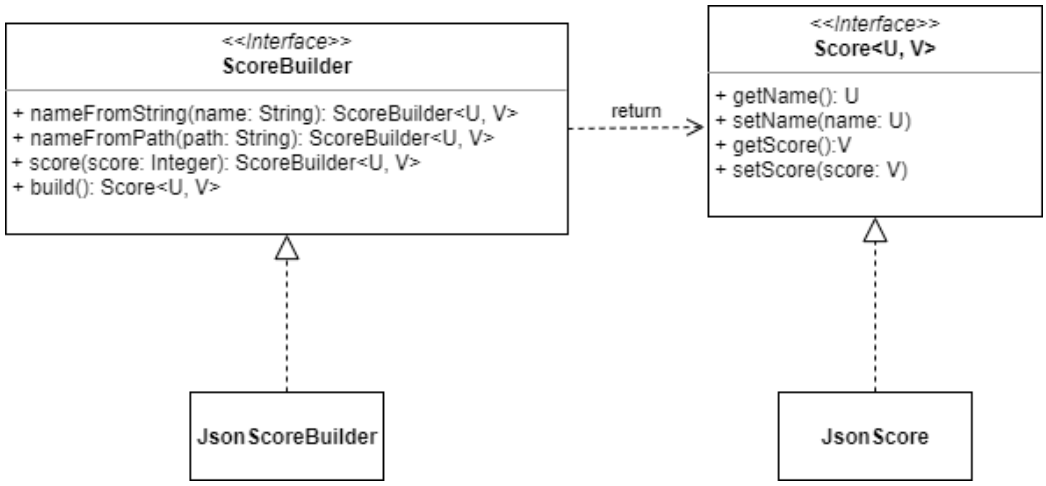


Figura 2.14: Il menu interagisce con i vari controller per soddisfare le richieste dell'utente

L'interfaccia `ScoreBuilder` definisce quelli che sono i metodi da implementare. Il `JSonScoreBuilder` fornisce quindi un JSonScore grazie ad una costruzione graduale ed intuitiva seguita dalla chiamata del metodo **build()**.

Mondo e HUD

Il framework FXGL fornisce un'efficiente creazione del mondo di gioco partendo da un file TMX, un formato che si basa su XML. Per quanto riguarda il HUD (Head-Up Display), e cioè la grafica a bordo schermo che tiene l'utente aggiornato sulla vita residua del player e sul punteggio accumulato, si è utilizzato un file FXML, anch'esso di base XML. Sono stati introdotti dei controller che forniscono i suddetti file alla classe `TheLastManStanding`, il vero "motore" del gioco, per avere un codice più snello e chiaro.

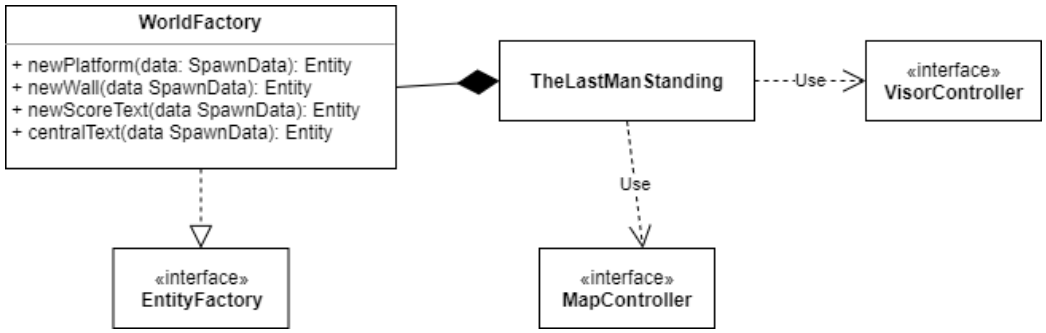


Figura 2.15: The Last Man Standing utilizza WorldFactory per lo spawn delle entità statiche

La creazione di quelle che sono le entità immobili del mondo è affidata alla classe `WorldFactory` che implementa la classe astratta di FXGL `EntityFactory`. Il nome suggerisce l'utilizzo del pattern **Factory**. Difatti questa classe si occupa di creare tutti quelli che sono gli oggetti inanimati del gioco, come muri, pavimenti ed alcuni particolari testi. In questa architettura ogni oggetto del mondo è visto come un'entità.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per il testing di 2121: The last man standing è stata scelta la libreria di JUnit 5. Gran parte delle funzionalità dell'applicazione si appoggiano sul game engine di FXGL. Non ci è risultato possibile con la nostra esperienza disaccoppiare componenti molto importanti di gioco dal framework scelto. Abbiamo quindi testato diverse classi che sono rimaste indipendenti dalla libreria.

3.1.1 Matteo Belletti

Avendo adottato FXGL come libreria, in fase di testing, ho risentito della struttura rigida e internamente codipendente della stessa, limitante per quanto riguarda il controllo di singole funzionalità delle entità. Tuttavia, la scelta di creare un model indipendente dalla libreria grafica mi ha permesso di testare la parte logica di codice su cui, di fatto, si regge tutto il restante software.
Ho infine testato parti di codice che, pur utilizzando FXGL, restano passibili di test autonomi dal motore di gioco.

- abstractGunTest;
- gunTypesTest;
- gunComponentTest;

3.1.2 Luca Cantagallo

Durante lo sviluppo del videogioco, ho testato alcune funzionalità di base avvalendomi della libreria JUnit. Purtroppo, da questo punto di vista, FXGL non è stato nostro amico siccome non ci ha dato modo di testare molte delle sue funzionalità, tra cui le entità: nel mio caso le entità player e power-up. Quello che ho testato con JUnit è nel "model", nella parte logica, quindi totalmente libera da FXGL:

- coloredTexture: questo controlla il corretto funzionamento dell'Enum "PlayerColor", essenziale affinchè il proxy non dia errori e tenga traccia del colore precedente alla collisione con un'altra entità: essenziale quindi alla trasformazione.
- initialPlayerParameters: questo controlla che il player venga creato correttamente, con i parametri giusti e che tutti vengano assegnati al valore atteso.

Le funzionalità che non sono riuscito a testare per mezzo di JUnit le ho testate manualmente. Quelle che mi hanno messo più in difficoltà sono le seguenti:

- Il cambio delle texture durante la trasformazione;
- Trasformazione durante le collisioni e proxy;
- Il fatto che le texture si comportassero coerentemente rispetto a ciò che eseguiva il PlayerComponent;
- Il despawn dei power-up una volta entrati in contatto con il player;
- Gestione Input da Tastiera e il corrispettivo funzionamento.

3.1.3 Valerio Di Zio

In fase di test mi sono reso conto della struttura monolitica della libreria e, dopo svariati tentativi, ho deciso di non scrivere alcun test sulla porzione di codice che utilizza FXGL. Durante la fase di progettazione, tuttavia, ho definito molte classi indipendenti dalla libreria, per esempio:

- creazione del modello dello zombie: testZombieCreation() ;
- gestione delle texture con una classe dedicata Texture: testTexture();
- creazione di uno zombie decorato: testZombieDecoration() ;

3.1.4 Luca Morlino

Tra i miei compiti c'era quello della gestione dei punteggi e delle classifiche di gioco. Testare classi che si occupano di verificare se la classifica va aggiornata è stato fondamentale. Nello specifico i miei test controllano se un punteggio ottenuto risulti nella Top 3 della relativa mappa e se l'aggiornamento della classifica sia corretto. Ho anche testato la giusta implementazione del pattern Builder della classe JsonScoreBuilder.

- testScoreModelTopThree
- testScoreModelUpdateRanking
- testJSonScoreBuilder

3.2 Metodologia di lavoro

Dopo aver sviluppato il modello del dominio la suddivisione dei compiti è stata rapida e, considerato a posteriori il tempo speso da ogni componente, efficiente.

È stato molto importante l'uso di **Git** per coordinare il lavoro svolto dai membri del gruppo. Sono stati creati un branch **master** che ospitasse solo le versioni funzionanti del gioco, un branch **develop** in cui unire il lavoro di ogni membro e un branch personale per ognuno.

I problemi principali di integrazione delle parti di codice sviluppate separatamente sono stati dovuti all'inesperienza iniziale dell'uso di un DVCS. Tuttavia una buona suddivisione dei compiti e una buona organizzazione delle interfacce hanno permesso di ridurre drasticamente i tempi di merging. Il lavoro di gruppo è stato organizzato in modo incrementale e dettato da scadenze. Ciò ha permesso di avere sin da subito una versione funzionante del gioco e di evidenziare per tempo i problemi generati da una libreria a noi poco conosciuta.

La classe `TheLastManStanding`, il system dell'applicazione, è stata una classe sviluppata da tutti gli elementi del gruppo.

3.2.1 Matteo Belletti

- Gestione dei proiettili e, di conseguenza, delle armi.
- Implementazione delle armi di gioco: diverse tipologie, per cui variano textures, tipo di proiettili, danni ed altre caratteristiche.

- Spawning delle armi, come entità di gioco, equipaggiabili durante la partita.
- Implementazione delle collisioni con le armi potenziate, di un sistema di timing per la durata degli equipaggiamenti, nonché di gestione della concorrenza.
- Implementazione dei proiettili, le cui caratteristiche sono legate a quelle delle armi.
- Integrazione delle armi al player, con adeguati riferimenti e componenti ad hoc.

3.2.2 Luca Cantagallo

- gestione delle entità player e dei power-up (con tutte le loro funzionalità);
- gestione delle texture del player e dei power-up (in collaborazione con Valerio Di Zio che ha progettato e implementato la classe "Texture" nel model, classe che ho utilizzato in più occasioni);
- input da tastiera (eccetto il comando per sparare e ricaricare i colpi, concretizzato da Matteo Belletti);
- collisioni riguardanti il player (io ho implementato cosa accadeva al momento delle collisioni player-zombie e player-powerup, ma la classe vera e propria è stata progettata e impostata da Valerio Di Zio);
- gestione della musica in background;
- spawn del player e dei power-up nella TLMSFactory (ho costruito il builder delle mie due entità. La classe è stata impostata da Valerio Di Zio);

3.2.3 Valerio Di Zio

- Modellizzazione e creazione degli zombie.
- Creazione dei decorator per i due tipi di zombie.
- Creazione di una classe Texture utilizzata per l'assegnazione di texture alle entità presenti nel gioco.

- Creazione di una interfaccia Moveable utilizzata per il movimento delle entità all'interno del gioco.
- Creazione di ComponentUtils per agevolare la leggibilità del codice.
- Creazione di vari "component" custom:
  - DamagingComponent: permette ad una entità di fare danno.
  - FollowPlayerComponent: permette all'entità di muoversi seguendo il player all'interno della mappa.
  - RandomMovementComponent: permette all'entità di muoversi in modo casuale all'interno della mappa.
  - ZombieTextureComponent: permette l'assegnazione delle svariate texture e animazioni all'entità.
- Studio e implementazione delle classi e dei metodi per intercettare le collisioni e lanciare delle routine implementate da me, per le collisioni che riguardano gli zombie, e dai miei colleghi per i loro rispettivi funzionalità.
- Creazione di un Thread: ZombieSpawner che gestisce la creazione casuale e lo spawn degli zombie.
- Insieme ai miei colleghi ho contribuito alla creazione della TLMSFactory, aggiungendo due metodi per la creazione dei due tipi di zombie.

3.2.4 Luca Morlino

- Gestione dell'ambiente di gioco:
  - creazione dei vari mondi;
  - creazione del HUD che tiene traccia della vita del player, del punteggio accumulato e dei messaggi di testo che appaiono sullo schermo;
  - creazione di tutte le entità statiche;
- Generazione del menu con la libreria Swing
- Realizzazione del sistema di scrittura/lettura della classifica e delle preferenze dell'utente.

3.3 Note di sviluppo

3.3.1 Matteo Belletti

- **Lambda expressions**: utilizzo della programmazione funzionale per una maggiore leggibilità e agevolezza del codice.
  Programmazione di azioni posposte, come la scadenza delle armi potenziate, o di altre cadenziate, come lo spawn randomicamente prestabilito di armi, usando adeguatamente metodi di FXGL tramite espressioni lambda.

- **Stream**: utilizzo nella gestione delle texture delle armi, per estendibilità lasciate a libera scelta, appoggiandomi all'interfaccia Texture predispostami dal collega Valerio.
  Nella mia personale implementazione ho utilizzato gli stream, per ottenere un accesso fluido alle texture fornite con mappe, particolarmente apprezzati lavorando con un numero aprioristicamente indefinito di elementi.

- **JavaFX**: utilizzo per funzioni di base (es. gestione file .png).

- **FXGL**: Sul limbo tra libreria grafica e framework, il suo studio e utilizzo sono stati alla base del progetto.

3.3.2 Luca Cantagallo

- FXGL: questa libreria è stata davvero utile per quanto riguarda la costruzione del player e power-up in quanto entità e per il corrispettivo scorrimento delle texture.

- Lambda expression: utilizzate per rendere più sintetico e compatto il codice, in particolar modo per la gestione dei timer e la costruzione del componente fisico (tipo il sensore del terreno). Le lambda expression utilizzate, come dirò anche in uno dei prossimi punti, derivano principalmente da implementazioni e algoritmi specifici di FXGL.

- JavaFX: studiato ed utilizzato inizialmente per poi essere sostituito da FXGL. Ho lasciato qualche import e qualche metodo, come ad esempio la conversione dei png in immagini, e di conseguenza in animazioni.

Per studiare e apprendere al meglio l'uso di FXGL ho consultato tutto ciò che ho trovato in rete a riguardo, ho guardato tanti video e studiato diversi tutorial. In particolare ho preso come principale riferimento questo tutorial:

https://github.com/AlmasB/FXGL/wiki/FXGL-11

Per altri aspetti meno di libreria, mi sono ritrovato a consultare di nuovo le slide del corso ed esami passati valutati positivamente.

Una parte di codice che ho copiato, poi cercato di riadattare quanto fosse possibile, è stata lo scorrimento delle texture e il loro "update continuo". Lascio qui il link: https://github.com/AlmasB/FXGL/wiki/Adding-Sprite-Animations-%28FXGL11%29 Un altro blocco di codice copiato e riadattato è stato questo, utilizzato come strumento per scandire il tempo: https://github.com/AlmasB/FXGL/wiki/Timer-Actions-%28FXGL-11%29

3.3.3 Valerio Di Zio

- **Lambda expressions**: utilizzate per i timer di gioco, per ripetere porzioni di codice in un lasso di tempo o per far attendere il codice prima di procedere con l'esecuzione.
- **FXGL**: framework grafico.
- **JavaFX**: utilizzato per convertire i file png in Image nelle Texture.
- **Crediti**: Per lo studio della libreria ho letto la wiki del creatore, https://github.com/AlmasB/FXGL/wiki/FXGL-11

3.3.4 Luca Morlino

- **Stream**: per la gestione delle classifiche mi è stato molto utile lavorare con gli Stream in quanto il loro utilizzo, combinato alle lambda expressions, è una delle feature che più mi hanno affascinato durante il corso in quanto semplificano notevolmente la produzione del codice.
- **Lambda expressions**: ho fatto largo utilizzo delle lambda expressions sia nell'implementazione del menu di gioco sia nella gestione di scrittura/lettura su file.
- **Generici**: la creazione delle interfacce generiche Score e ScoreBuilder rendono possibile un loro eventuale riutilizzo con differenti tipi di dato.
- **FXGL e Javafx**: framework grafico.
- **Swing**: per la generazione del menu mi sono appoggiato alla libreria Swing
- **Crediti**: per la creazione della classe generica Score e di ScoreBuilder mi sono ispirato al progetto `Fight Avenge Guerrilla`.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Matteo Belletti

Riguardando alla mia parte di progetto, mi ritengo più che soddisfatto del lavoro svolto: mi impegnato ad andare oltre alla parte obbligatoria per sviluppare più funzionalità possibili per il progetto, al meglio delle mie capacità. Ho sottoposto il mio codice a più fasi di rifattorizzazione, revisione e modifiche, partendo da un'idea vaga del lavoro che sarei andato a svolgere, per poi affinarla sempre più, cercando di utilizzare al meglio pattern e buone pratiche, per produrre un codice il più possibile di qualità.
Per quanto riguarda la presenza nel gruppo, mi son reso al massimo disponibile ad offrire il mio aiuto, condividendo i risultati del mio lavoro e delle mie ricerche ogni qualvolta li ritenessi di possibile utilità al team, cercando di creare un ambiente di collaborazione che, non senza qualche piccola discussione interna (probabilmente fisiologica), si è riusciti a creare in modo propositivo.
Come ho cercato di aiutare i miei compagni, qualora ne avessero bisogno, non ho esitato a chiedere una mano a mia volta, quando mi sono ritrovato ad affrontare problemi da altri già affrontati, in modo da ottimizzarne la risoluzione e imparare da chi fosse in possesso di più informazioni di me in quell'ambito limitando il più possibile sprechi di tempo.
Sebbene io guardi con orgoglio al lavoro da me svolto, riconosco di aver avuto qualche criticità, essendo la gestione delle armi non indipendente nel gioco, ma strettamente connessa e funzionale al player e agli zombie.
Nella fase iniziale ho avuto qualche difficoltà ad avviarmi nella programmazione senza il codice a cui le armi avrebbero poi fatto riferimento, in particolare mi sono ritrovato a gettare una base generale delle classi, ritrovandomi

poi disorientato ad attendere il codice dei compagni necessario all'ecosistema delle armi; proprio in questa parte iniziale, i miei compagni hanno affrontato e risolto interrogativi, perlopiù riguardanti l'uso di FXGL, di fatto facilitandomi la strada nelle prime fasi di unione del codice.
Senza falsa modestia, ritengo però di aver dato successivamente il massimo per ricambiare e fare da apripista per la risoluzione di problematiche e scoperta di funzionalità.

4.1.2 Luca Cantagallo

Questo è stato il primo progetto serio di tutta la mia vita. Se tornassi indietro al giorno in cui ho cominciato il codice, i consigli che darei a me stesso sarebbero di dare più peso alla fase di analisi, di fare utilizzo più maturo delle feature avanzate tenendole più in considerazione e iniziare ad utilizzare prima (temporalmente) i pattern di design. Devo dire che già il fatto che mi sentirei di dare consigli a un me stesso del passato mi fa capire che questo progetto mi ha fatto apprendere molto. Ho imparato senza dubbio a lavorare meglio in gruppo, e capito cosa voglia dire il "fare un lavoro di gruppo ma non essere in quattro nello stesso pc". Ho imparato a pormi scadenze per poi fare anche le tre di notte se necessario per rispettarle. Ho avuto modo di migliorare le mie capacità di programmazione, grazie anche all'utilizzo dei pattern della quale, messi ora in pratica, ne ho capito la reale importanza e da quali esigenze sono nati alcuni di questi. Ho approfondito lo studio e l'apprensione di particolari tools utili alla programmazione, come ad esempio il caro Git. Sono soddisfatto di tutto ciò che ho imparato, della mentalità con la quale ho affrontato il progetto in ogni sua sfumatura e del risultato finale. Sono sicuro che tutto ciò mi sarà davvero utile in un futuro, anche abbastanza prossimo, sia nella mia vita universitaria, sia, chissà, nella mia vita lavorativa. È stata senza dubbio un'esperienza costruttiva, sia dal punto di crescita personale, dal punto di vista accademico e dal punto di vista del lavoro personale e collettivo.

4.1.3 Valerio Di Zio

Secondo la mia opinione la parte che ho svolto nel progetto è ben strutturata; sono soddisfatto ed entusiasta del lavoro svolto. In questi mesi ho cercato di dare il massimo e fin da subito, ho provato ad individuare la maniera per poter svolgere il compito che mi spettava: nella fase di progettazione, l'ho analizzato cercando di adottare la miglior modalità possibile per realizzarlo. In un secondo momento mi sono soffermando sullo studio del framework FXGL per capire al meglio quello che potesse risultare utile a me e ai miei colleghi.

Durante tutta la fase di progettazione e di sviluppo sono stato molto attivo e ho messo a disposizione tutte le mie conoscenze, sia quelle apprese durante il corso, che quelle in ambito extra-universitario. In alcune fasi del progetto ho chiesto aiuto ai miei colleghi, senza alcuna esitazione; d'altro canto, anche io ho aiutato i miei compagni, con svariate problematiche da loro riscontrate. Il mio rapporto con i compagni è stato più che ottimo, nonostante alcune divergenze, che una volta risolte ci hanno fatto crescere professionalmente e personalmente. Dovendo trovare delle criticità al mio codice credo che FXGL non ci abbia permesso di crearlo totalmente indipendente da quest'ultima. In ogni caso, questa per me è stata una opportunità per mettermi alla prova e sperimentare in un certo senso "il mondo del lavoro" attraverso il lavoro in team, il rispetto delle scadenze e la creazione di un progetto complesso e corposo.

4.1.4 Luca Morlino

Questo progetto è stato per me molto utile per capire quanto effettivamente ho appreso e quanto invece devo e voglio approfondire riguardo la programmazione ad oggetti.

Altro aspetto non secondario è quello del lavoro di gruppo. Ci sono stati inevitabilmente degli scontri che hanno anche bloccato il lavoro per alcuni giorni ma sono stati sempre risolti. Credo fortemente dell'utilità pratica delle discussioni se ben argomentate. Immagino anche che tale tipo di problema sia propedeutico al corso e parte integrante della scelta di far elaborare un progetto agli studenti.

Il problema principale è stato quello della scelta del framework FXGL, una libreria il cui intento è facilitare il più possibile la creazione di un gioco "from scratch". Probabilmente anche un non addetto ai lavori riuscirebbe a creare un prodotto giocabile seguendo i tutorial. È stato quindi nostro intento scrivere codice applicando il più possibile gli insegnamenti avuti.

La nostra inesperienza progettuale ha fatto sì che non avessimo una solida impostazione architetturale e un relativo UML. Solo dopo "esserci buttati" sul codice (come qualunque docente universitario sconsiglia) ci siamo accorti che l'MVC non fosse applicabile: a quel punto la deadline era più vicina ed è stato optato per proseguire con il framework ma di rendere il più possibile indipendenti le varie componenti. Io personalmente sono riuscito ad utilizzare MVC (spero correttamente) in quanto il compito che avevo riguardava gli aspetti più esterni dell'applicazione. Ho usato un approccio incrementale, definendo inzialmente elementi base delle classi che col tempo hanno raggiunto la forma attuale.

Altra mia soddisfazione è derivata dall'uso di Git. Ben prima di lavorare al progetto ho ritenuto che uno studio approfondito delle funzionalità viste a lezione potesse evitare problemi futuri. Sono stato l'involontario responsabile di molti dei merging più ostici. Credo che Git sia tanto utile quanto pericoloso in mani insicure. Ho anche scritto un piccolo manuale Git per il gruppo usando il linguaggio Latex. Questo mi ha permesso anche di risparmiare tempo nella stesura della suddetta relazione.

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Matteo Belletti

In questa indimenticabile esperienza (nel bene e nel male), non sono mancate problematiche di rilievo.
Cercando di seguire le linee guida forniteci dai docenti, abbiamo esordito nel progetto cercando di creare schemi concettuali: UML e analisi preventive, ottenendo scarsi risultati soprattutto per quel che riguarda la progettazione, dovuti all'inesperienza e alla mancanza di riferimenti per quanto riguarda la progettazione di videogame.
Ritengo che, seppure formativo in giusta misura, il salto di difficoltà dall'esperienza a laboratorio e a lezione per poi passare al progetto, sia esagerata.
Ci siamo trovati a dover scegliere una libreria grafica, da cui il nostro codice sarebbe pesantemente dipeso, senza aver avuto una direzione specifica, ritrovandoci a "pescare" da un link fornitoci la libreria che ci sembrava potesse calzare per il nostro progetto: FXGL.
Dopo averla scelta e aver speso un grosso quantitativo di tempo a studiarla, ci siamo ritrovati a chiedere informazioni per quanto riguardasse i pattern architetturali, dato che il nostro intento di applicare MVC al progetto sembrava essere molto difficoltoso sulla libreria/framework; abbiamo così scoperto che FXGL è una libreria monolitica, poco versatile e poco elastica, dovendo così decidere se ripartire da zero con JavaFX, perdendo un quantitativo notevole di ore di studio, o proseguire con FXGL.
In questo frangente il mio team si è diviso a metà con opinioni opposte, entrambe sostenute da argomentazioni ragionevoli, arrivando infine alla decisione di proseguire con FXGL.
Riassumendo, credo che questo corso sia stato ottimo: sia per l'esperienza accumulata, che per la qualità delle lezioni fornite, ma abbia importanti criticità quando si parla di esame: ritengo che il carico di studio e lavoro autonomo sia estremamente sproporzionato portando me, e molti studenti

con cui mi sono confrontato, a rinunciare a ore di sonno e a dover sospendere l'ascolto delle lezioni del secondo semestre per lavorare al progetto, nonostante fossi partito in anticipo.
Il mio suggerimento è di fornire una transizione meno traumatica dal corso al progetto, fornendo qualche esempio di modus operandi per quanto riguarda progetti non giocattolo.
Consiglierei anche di dare direzioni più precise agli studenti per quanto riguarda le librerie grafiche, per evitare che altri come noi, per inesperienza e/o sfortuna, investano tanto tempo nello studio di una libreria, che si riveli poi limitante per il progetto e per l'applicazione delle pratiche di buona programmazione imparate nel corso. In questo caso, secondo me, restringere il campo di scelta a poche librerie selezionate dai docenti, potrebbe essere un fattore positivo.

4.2.2 Luca Cantagallo

Trovo che questo corso sia stato un corso davvero completo. Ho apprezzato il fatto che siamo partiti dalle basi a inizio corso: ho iniziato a programmare da poco prima aver cominciato l'università, e senz'altro sono nella categoria di "quelli scarsi". Eppure ciò che è stato spiegato è sempre stato spiegato sempre in un modo chiaro e semplice, mai entrando in discorsi troppo complessi o "per pochi intenditori". Fin dalle prime lezioni, inoltre, i professori sono stati in grado di coinvolgere anche la mia attenzione spiegando del perché è importante studiare la Programmazione a Oggetti e la differenza che si trova rispetto a tutto ciò che avevamo studiato fino a quel momento. Spiegare a cosa serve nella pratica ciò che si studierà per mesi, per quanto possa sembrare banale, è davvero importante e stimolante. A livello di progetto, come ho preannunciato nella mia prima sezione di autovalutazione, sono soddisfatto, anche di come è stato organizzato. Senza dubbio è stato il progetto che più si è avvicinato, fin ora, a farci capire nella pratica come viene sviluppato un vero e proprio software. L'unica critica che mi sento di fare è il numero di crediti per questo corso. Secondo me il numero di ore di studio e il carico di lavoro è decisamente sproporzionato per essere un corso a soli 12 crediti. Non ho basi né la giusta esperienza forse per questa affermazione. Parlo facendo il confronto con altri corsi da 12 crediti che risultano essere enormemente meno cariche di ore di studio. Il mio pensiero (per quanto ho la consapevolezza che concretamente ci saranno migliaia di burocrazie nel mezzo per realizzare ciò) non è quello di rendere più leggero il corso: come ho detto prima è davvero molto formativo! La soluzione per me sarebbe raddoppiare o comunque aumentare il numero di crediti per questo corso.

4.2.3 Valerio Di Zio

Questo corso è stato uno dei migliori che abbia seguito, non ho riscontrato dei problemi nella comprensione degli argomenti trattati, anzi da essi ho appreso nuove pratiche di programmazione. Per quanto riguarda la creazione del gioco, nonostante io ne sia stato davvero esaltato, nel concreto abbiamo trovato delle difficoltà non indifferenti. Per creare un gioco abbiamo dovuto scegliere, inevitabilmente, una libreria grafica su cui appoggiarci, cosa non facile in quanto non avendo esperienza, il margine di errore poteva essere elevato. Successivamente allo studio della libreria ci siamo resi conto che era monolitica e non permetteva (per quanto ne sappiamo) di creare una buona parte del codice indipendente da essa. In questa fase della creazione del gioco sarebbe stato utile avere maggiori direttive. Quindi, sulla base di queste mie considerazioni, consiglieri di dare direttive più specifiche per quanto riguarda la scelta di quest'ultima.

4.2.4 Luca Morlino

Al momento della scrittura di questa parte sono tanto soddisfatto di questo corso quanto stremato fisicamente. Ciò però non è per me un difetto. L'impegno speso nel seguire le lezioni, nel partecipare attivamente a tutti i laboratori e nell'elaborare questo progetto mi hanno cresciuto tantissimo.

Il mio primo "Hello World" l'ho visto lo scorso anno durante le prime lezioni di "Programmazione in C". Vedere ora il lavoro prodotto per questo progetto mi riempie di soddisfazione e mi fa ritenere il corso di Programmazione ad oggetti non solo come il più valido tra tutti quelli seguiti finora ma anche costruito in modo tale che chiunque (con la giusta dose di volontà) possa cimentarcisi ed imparare tutto ciò che viene insegnato. Una critica con difficile soluzione la do al numero di crediti dedicati a questo corso. Sarei bugiardo se non dicessi che secondo me ne vale almeno il doppio. La quantità degli argomenti affrontati è notevole, soprattutto quelli in laboratorio. Ne hanno conseguito per me alcune questioni trattate velocemente o in momenti in cui l'esperienza dello studente non è tale da poter apprendere bene i concetti. Le mie difficoltà maggiori durante il progetto sono state dovute all'inesperienza di una stesura di un UML e dell'utilizzo dell'MVC. Più lezioni di laboratorio finalizzati su questi temi (come quello del robot) potrebbero aiutare i futuri studenti.

Appendice A

Guida utente

All'avvio della applicazione viene mostrato il menù (Figura A.1) principale da cui l'utente potrà scegliere tra diverse opzioni:

- **Username**: qui è possibile inserire un nome utente, che verrà utilizzato successivamente per la memorizzazione del punteggio della partita successiva.
- **Maps**: in questa sezione è possibile scegliere una mappa di gioco per la partita successiva. Se non dovesse venire selezionato nulla, il gioco partirà con la mappa di default (Cemetery).
- **Ranking**: in questa sezione compariranno, per ogni mappa, la classifica con i 3 username che hanno ottenuto il punteggio più alto.
- **Controls**: cliccando questo pulsante apparirà una finestra che illustra i vari comandi del gioco.
- **Start**: lancia una nuova partita.
- **Exit**: chiude la finestra di gioco.

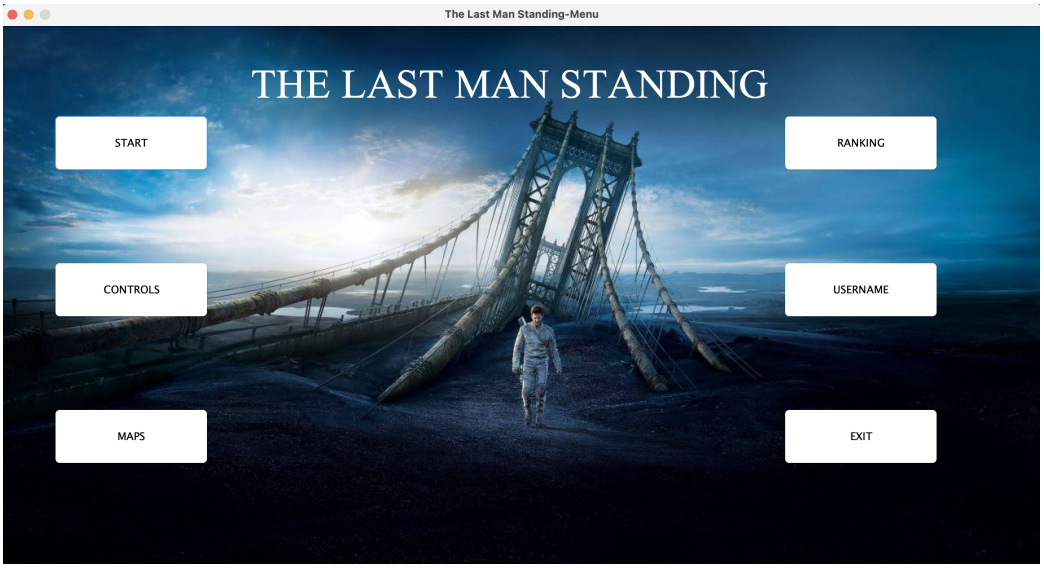


Figura A.1: Menù di gioco

Appena iniziata una nuova partita, comparirà il player (Figura A.2). Il player può muoversi utilizzando i tasti:

- **W**: Salta.
- **A**: Vai a sinistra.
- **D**: Vai a destra.
- **L**: Spara.
- **R**: Ricarica le munizioni.
- **Q**: Si gira a sinistra.
- **E**: Si gira a destra.
- **S**: Atterra più velocemente (abilità sbloccabile con la trasformazione).

Figura A.2: Player mentre salta

Dopo alcuni secondi inizieranno ad apparire degli zombie (Figura A.3) in modo casuale: l'utente dovrà cercare di ucciderli! Gli zombie non sono tutti uguali, avranno statistiche diverse l'uno dall'altro, inoltre alcuni ti seguiranno altri invece si muoveranno casualmente. Il compito dell'utente è lo stesso, rimanere in vita e totalizzare più punti possibili.

Figura A.3: Zombie che invadono la mappa

Durante il gioco appariranno svariati oggetti che aiuteranno l'utente a sopravvivere e a ottenere punti:

- **FirePowerUP (Figura A.4)**: prendendo il powerup e aspettando 1 secondo il player si trasforma:
  - Il colore del giocatore diventerà rosso.
  - Ripristina la vita.
  - Incrementa la velocità.
  - Incrementa il numero di salti che può effettuare a mezz'aria.
  - Aggiunta di una nuova abilità, utilizzabile con un nuovo comando ("S") chiamata "Aereodinamicità", che consente di atterrare più velocemente.
  - Se vieni colpito da uno zombie quest'ultimo ti toglierà tutte le abilità dovute al FirePowerUp!

Figura A.4: FirePowerUp

- **MagmaGun (Figura A.5)**: raccogliendo quest'arma da terra il player utilizzerà una potente arma che spara grandi palle di fuoco che uccidono gli zombie con un solo colpo.
  Disponibile per pochi secondi.

Figura A.5: MagmaGun

- **MachineGun (Figura A.6)**: raccogliendo quest'arma da terra il player utilizzerà una mitragliatrice in grado di sparare colpi infiniti, senza dover ricaricare.
  Disponibile per pochi secondi.

Figura A.6: MachineGun

Appendice B

Esercitazioni di laboratorio

B.0.1 Matteo Belletti

- Laboratorio 07: `https://github.com/MatteBelle/OOP-Lab07.git`
- Laboratorio 08: `https://github.com/MatteBelle/OOP-Lab08`
- Laboratorio 09: `https://github.com/MatteBelle/OOP-Lab09`

B.0.2 Luca Cantagallo

- Laboratorio 04: `https://virtuale.unibo.it/mod/forum/discuss.php?d=62685#p121265`
- Laboratorio 05: `https://virtuale.unibo.it/mod/forum/discuss.php?d=62684#p121268`
- Laboratorio 06: `https://virtuale.unibo.it/mod/forum/discuss.php?d=62579#p121270`
- Laboratorio 07: `https://virtuale.unibo.it/mod/forum/discuss.php?d=62582#p121277`

B.0.3 Valerio Di Zio

- Laboratorio 04: `https://virtuale.unibo.it/mod/forum/discuss.php?d=62685#p102967`
- Laboratorio 05: `https://virtuale.unibo.it/mod/forum/discuss.php?d=62684#p102969`

- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=62579#p102970
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=62582#p102972
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=63865#p103980
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=64639#p105843
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=66753#p121263

B.0.4 Luca Morlino

- Laboratorio 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=62684#p101508
- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=62579#p101509
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=62582#p101916
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=63865#p104055
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=64639#p104315
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=66753#p121302