Super Ninja Frog

Casamenti Gianmaria, Leoni Lorenzo, Pasini Luca, Spahiu Marsild

25 aprile 2021

Indice

Capitolo 1

Analisi

L'applicazione mira alla realizzazione di un gioco 2D platform. Il personaggio principale deve superare una serie di ostacoli e nemici al fine di completare i livelli del gioco. Il gameplay sarà in stile Super Mario.

1.1 Requisiti

Requisiti funzionali

All'avvio dell'applicazione verrà mostrato un menù iniziale che permetterà la scelta di diverse opzioni, tra le quali: "PLAY", "SETTING" ed "EXIT". Con la scelta della prima opzione si accederà alla schermata di selezione di uno dei livelli di gioco disponibili, con la seconda si potrà disattivare l'audio e con l'ultima si uscirà dall'applicativo. Il gioco è sviluppato in modalità single player, con l'obiettivo di raggiungere la fine del livello selezionato. Durante il percorso si potranno incontrare diversi tipi di nemici i quali, se non evitati oppure uccisi, possono portare alla sconfitta. Si potranno trovare anche una serie di potenziamenti sotto le sembianze di "frutta" che potranno dare punti, abilitare la funzione di "doppio salto" oppure aumentare il numero delle vite del personaggio.

Requisiti non funzionali

Dovrà essere garantita una corretta implementazione dei movimenti e della fisica del gioco, oltre ad una generale fluidità dell'applicazione.

1.2 Analisi e modello del dominio

I due livelli di cui si compone il gioco sono strutturati in maniera differente, però gli elementi e le entità di cui sono composti sono le stesse.
In ciascun livello il personaggio principale (NINJA) entrerà in contatto con diversi oggetti non interattivi: si potrà muovere sul terreno di gioco (GROUND) e dovrà superare gli ostacoli (GROUND_OBJECT) al fine di raggiungere la fine del livello (FINISH). Lungo il corso della mappa potrà collidere con la testa con due diversi tipi di oggetti interattivi, i BRICK e i FRUITBOX, i quali sono molto simili tra di loro ma hanno un comportamento diverso al contatto. Il primo si distruggerà, mentre il secondo dovrà far comparire un frutto. La frutta, allo stesso modo, può essere di tre diversi tipi, con caratteristiche simili, i quali però avranno effetti diversi al contatto col personaggio principale: il primo tipo (ORANGE) aumenterà il punteggio di gioco, il secondo tipo (CHERRY) darà una vita in più al personaggio mentre l'ultimo tipo (MELON) attiverà, per un determinato tempo, l'abilità del doppio salto. Bisognerà evitare anche i nemici, inizialmente di due tipi diversi (RINO e TURTLE), i quali possono essere uccisi saltandoci sopra (stando attenti alle spine nel caso della TURTLE).
In conclusione i livelli di gioco sono composti da diversi tipi di entità (nemici, frutta, oggetti), come mostrato in fig. 1.1, i quali però sono tra di loro definiti in maniera simile. Bisognerà gestire anche il sistema di collisioni del gioco, tenendo in considerazione che diverse entità interagiscono con il personaggio, ma con effetti diversi.

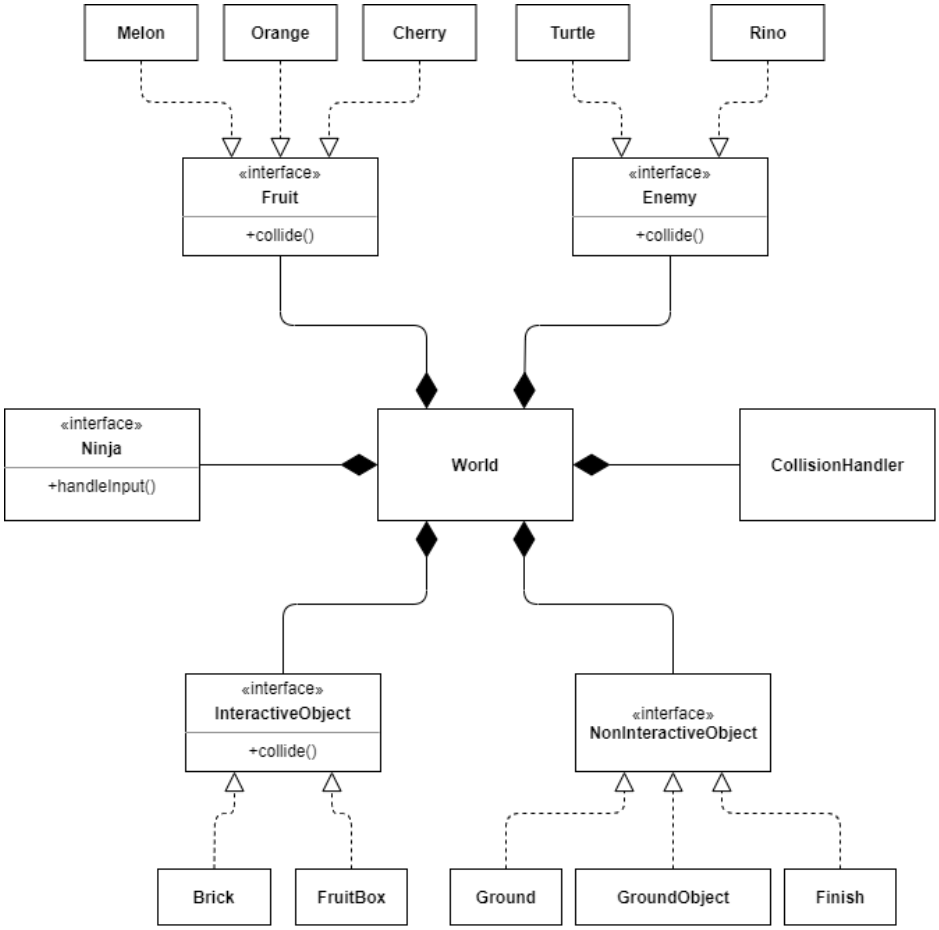

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro.

Capitolo 2

Design

2.1 Architettura

L'applicativo è stato realizzato mediante l'utilizzo della libreria libGDX, la quale permette la suddivisione del gioco a diverse schermate, tutte implementazioni dell'interfaccia "Screen", le quali vengono gestite e lanciate all'occorrenza dalla classe principale del gioco, NinjaFrogGame, la quale estende la classe di libreria Game. Quando una schermata viene selezionata, la sua gestione viene definita all'interno di tre principali metodi ereditati dall'interfaccia, show, render e dispose, i quali rispettivamente inizializzano gli elementi di cui è composto uno schermo (show), aggiornano continuamente (diversi frame per secondo) le componenti della schermata (render), e rilasciano le risorse utilizzate (dispose) alla chiusura del gioco o al cambio di schermata.
Per la realizzazione di questo applicativo sono risultate necessarie la realizzazione di sei diverse schermate, come mostrato in fig. 2.1: il MainMenu (menù principale), il LevelsMenu (scelta del livello di gioco), il SettingsMenu (impostazioni), il GameOverScreen (schermata di morte), il WinScreen (schermata di vincita) e il PlayScreen (schermata di gioco). Per informazioni dettagliate riguardo queste schermate vedere la successiva sezione di Design dettagliato.

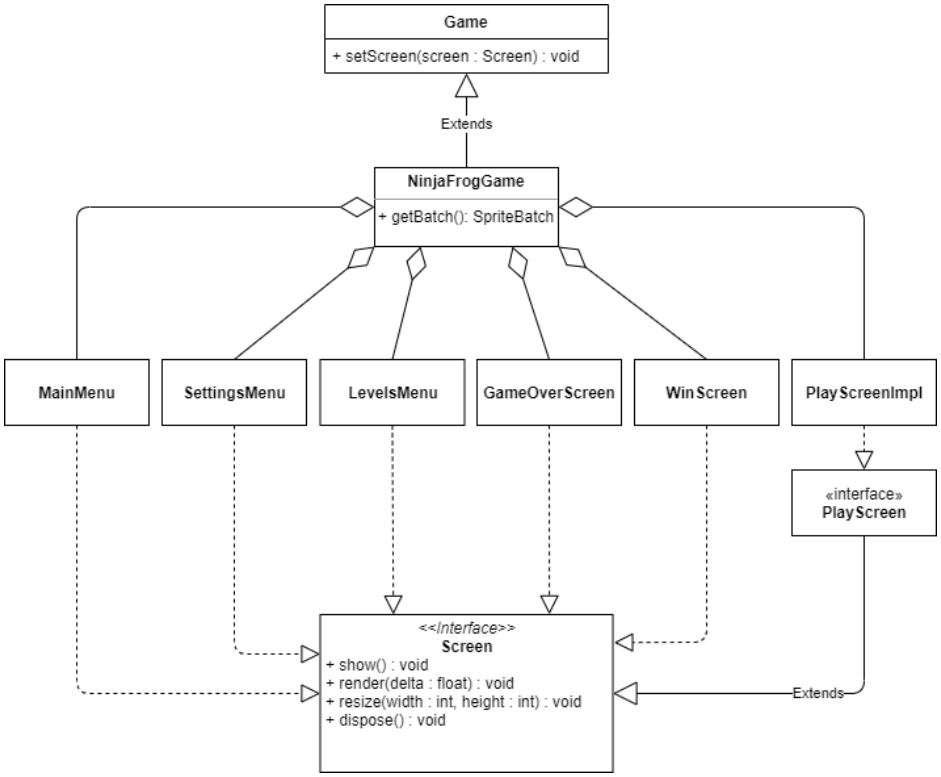

Figura 2.1: Schema UML dell'architettura.

Con l'utilizzo di questa architettura, la soluzione ai problemi visti durante la fase di analisi del problema (fig. 1.1) viene gestita nel seguente modo (fig. 2.2): la schermata di gioco, PlayScreen, gestisce la creazione e la coordinazione fra le diverse entità del mondo, tra cui la creazione del mondo fisico stesso. La suddivisione dei ruoli rimane comunque invariata. La gestione delle collisioni viene affidata alla classe worldCollisionListener, la quale rileva in autonomia la collisione tra due diversi corpi.

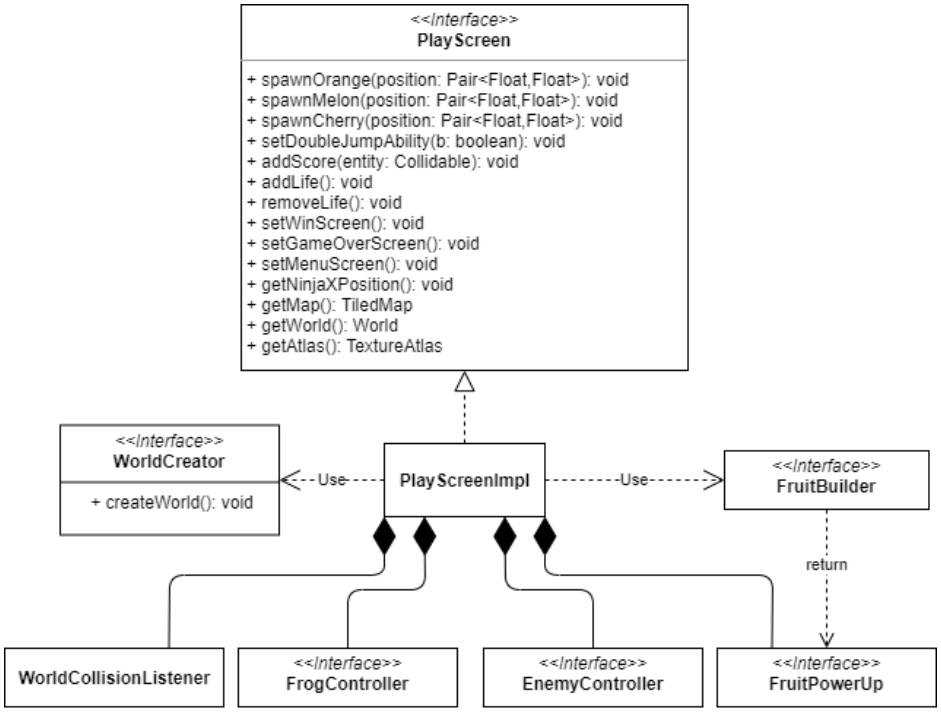

Figura 2.2: Schema UML dell'analisi con l'utilizzo di LibGDX.

Per informazioni dettagliate riguardo l'implementazione delle singole componenti, le interazioni tra le diverse interfacce e come viene definito il concetto di "entità che può collidere" vedere la successiva sezione di Design dettagliato.

2.2 Design dettagliato

2.2.1 Gianmaria Casamenti

All'interno dell' applicativo mi sono impegnato nello sviluppo delle seguenti entità:
Potenziamenti in gioco
I potenziamenti in gioco sono oggetti che fanno acquisire bonus o abilità supplementari al giocatore e vengono rappresentati sotto forma di frutti. All'interno dei livelli si può trovare:

- La ciliegia restituisce una vita aggiuntiva e 200 punti alla partita.
- Il melone attiva il potenziamento "DoubleJump" e aggiunge 150 punti alla partita.
- L'arancia è il bonus più comune e aggiunge 100 punti alla partita.

Durante lo sviluppo e l' implementazione di taluna parte, ho ritenuto idonea l'applicazione di 2 diversi GoF Design pattern, come si può notare nella fig. 2.3.

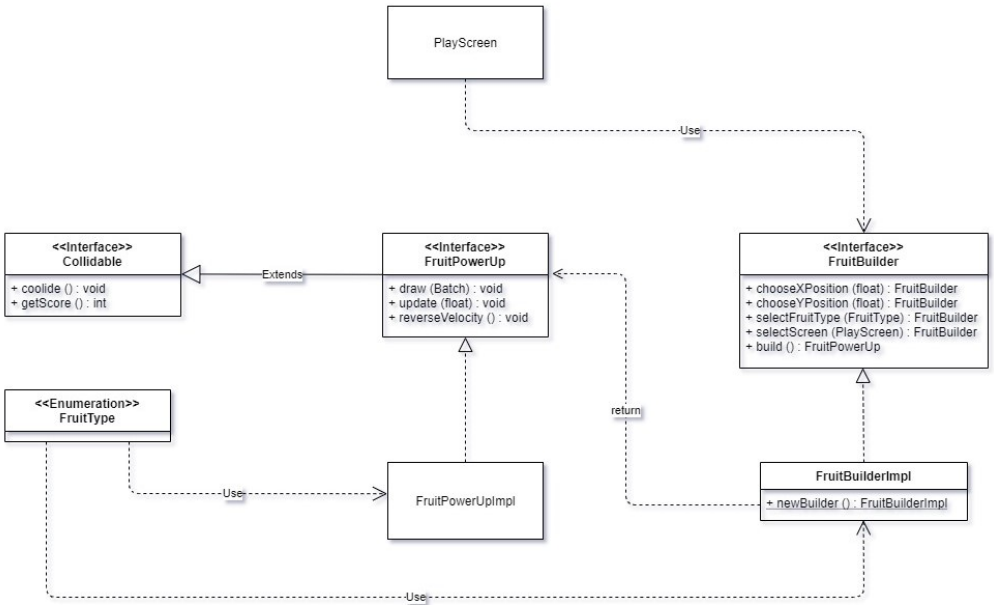

Figura 2.3: Rappresentazione UML completa dei potenziamenti in gioco

Il pattern Builder ha come obiettivo di separare la costruzione di un oggetto complesso come un FruitPowerUp dalla sua rappresentazione, cosicché un unico processo di costruzione possa creare diverse rappresentazioni della stessa entità.
Questo comporta una maggiore indipendenza dell'algoritmo di costruzione e rende più semplice il codice, permettendo alla classe Fruitbuilder di focalizzarsi su una corretta costruzione di un'istanza e lasciando che la classe originale si concentri sul funzionamento dell'oggetto.
Gli elementi costitutivi del pattern sopracitato sono visibili nella seguente figura (fig. 2.4).

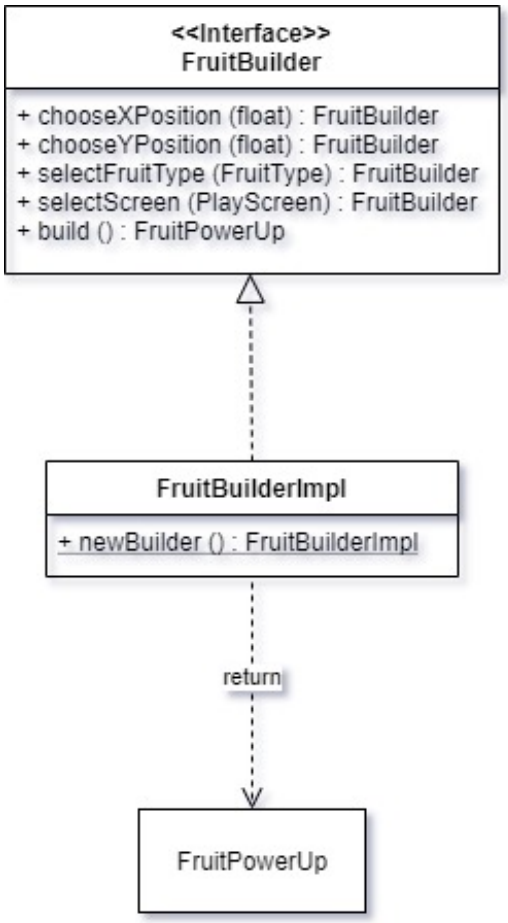

Figura 2.4: Rappresentazione UML dell'applicazione del pattern Builder.

Il secondo pattern utilizzato per l'implementazione dei power-up è **il pattern Strategy**, rappresentato nella Figura 2.5, che consiste nell' incapsulamento dell' algoritmo collide() all' interno della classe FruitPowerUpImpl mantenendo un' interfaccia Collidable, che rappresenta la strategia, comune a tutte le entità presenti nel gioco.

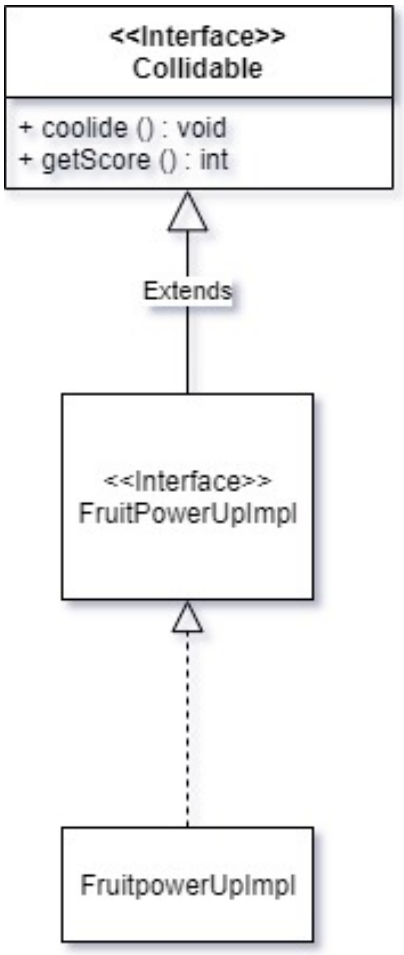

Figura 2.5: Rappresentazione UML dell'applicazione del pattern Strategy.

Menù di gioco

I menù sono le schermate iniziali in cui si possono impostare alcune configurazioni del gioco come audio e livelli.
Ciascuno di questi oggetti è caratterizzato da diverse "labels" posizionate al centro dello schermo, per rendere più intuitiva la selezione che si vuole effettuare ho aggiunto un selettore a lato dell'etichetta evidenziata.
Dopo un' attenta analisi della libreria, per la realizzazione dei menù ho deciso di utilizzare l'interfaccia "Screen" (fig. 2.6) fornita da LibGdx e di suddividerli in principale (MainMenu), di configurazione (SettingsMenù) e dei livelli (LevelsMenù).
I motivi della scelta sono stati dettati da una maggiore estensibilità degli oggetti e grazie a questo sistema aggiungere ulteriori menù e configurazioni sarebbe poco oneroso in termini di tempo e progettazione.

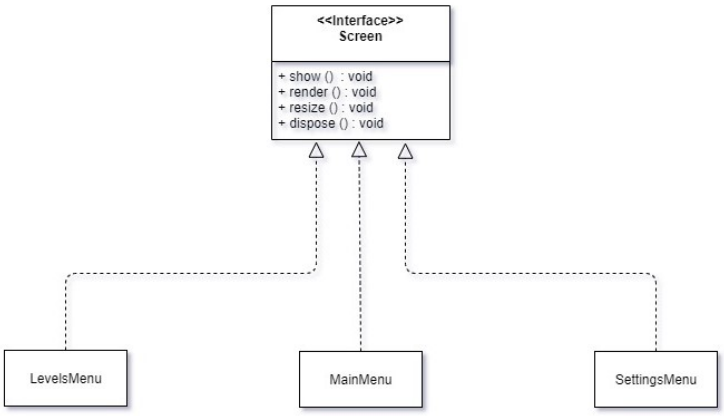

Figura 2.6: Rappresentazione UML dell'interfaccia Screen con i vari menu di gioco.

Schermate di fine gioco

Le schermate di fine gioco sono gli "Screen" che vengono lanciati in caso di vittoria (WinScreen) o sconfitta (GameOverScreen); all' interno di essi è possibile fare un'unica operazione di invio per tornare al menu principale. Nella fig. 2.7 è possibile esaminare l'UML dell'implementazione dell'interfaccia scelta.

Il passaggio tra i vari menu e schermate viene effettuato attraverso un apposito metodo della classe "NinjaFrogGame" che viene chiamato ogni qualvolta ci sia bisogno di cambiare la schermata corrente, questo ha garantito un corretto funzionamento dell' applicazione e una stesura del codice più pulita.

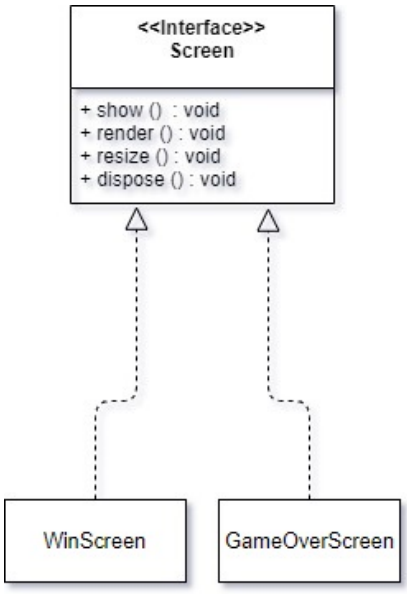

Figura 2.7: Rappresentazione UML delle implementazioni WinScreen e GameOverScreen

2.2.2 Lorenzo Leoni

Gestione Player
Per la gestione del player ho deciso di utilizzare il pattern architetturale MVC, come si può vedere nella fig. 2.8, avendo così una divisione più marcata tra la parte dei dati e di logica e la parte grafica.
I vari stati del player sono stati gestiti attraverso un'enumerazione che è stata poi utilizzata da tutte e tre le classi.
Le classi dell'MVC sono così suddivise:

- La **model** gestisce i dati e la logica dell'applicazione e li fornisce al controller su richiesta.
- La **view** oltre a disegnare il giocatore nel play screen, gestisce tutte le sue animazioni nei vari stati e in caso di potenziamento attivo cambia la skin del personaggio.
- Il **controller** gestisce l'input da tastiera per muovere il player all'interno del gioco e fa da tramite tra view e model modificando i dati in base a cosa accade nella view e viceversa.

Grazie all'utilizzo del pattern MVC, nel caso si volesse aggiungere successivamente un altro personaggio basterebbe implementare l'interfaccia della model già esistente, poichè abbastanza generica, e creare una nuova view passandola poi al costruttore del controller.

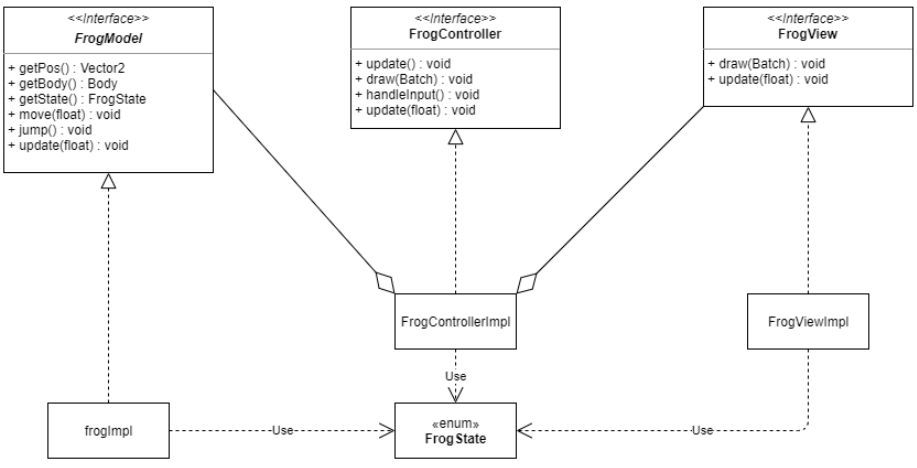

Figura 2.8: Rappresentazione UML del pattern MVC applicato al player.

Hud

L'Hud comprende tutte le informazioni continuamente visibili durante il gameplay, nel nostro caso: score, life e bonus. Ho quindi deciso di gestire questa parte attraverso una semplice interfaccia e, di conseguenza, la sua implementazione come in fig. 2.9. Al suo interno ci sono i metodi per tenere traccia del punteggio, delle vite disponibili.

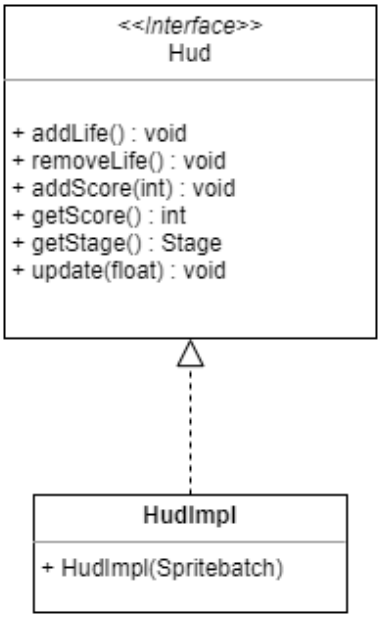

Figura 2.9: Rappresentazione UML dell'Hud

Sound Manager

Questa interfaccia è stata pensata per per gestire le tracce audio all'interno del gioco (fig. 2.10). Viene utilizzata da tutte le schermate, sia di gioco che di menù, per lanciare, stoppare le canzoni e anche mutare l'audio. Durante la sua implementazione ho sfruttato i metodi e i tipi forniti dalla classe Sound di libGDX.

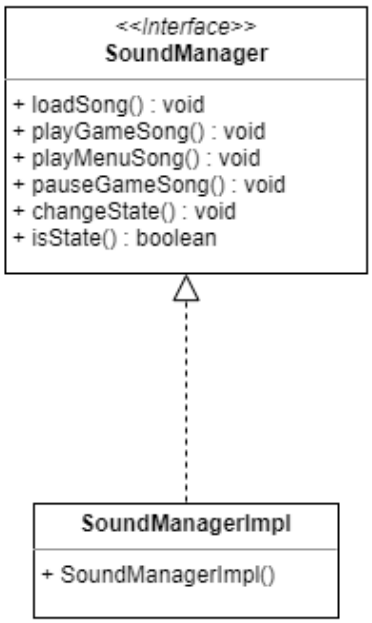

Figura 2.10: Schema UML dell'interfaccia SoundManager

2.2.3 Luca Pasini

La mia parte all'interno del progetto è stata quella legata allo sviluppo dei nemici. Durante lo progettazione ho deciso di utilizzare il pattern architetturale MVC cosi da rendere piu marcate le diverse parti che andranno poi a formare il nemico vero e proprio.

- La **Model** definisce le diverse parti da cui è formato il corpo del nemico e i metodi che lo contraddistinguono.
- La **View** è la parte che ha il compito di disegnare il nemico all'interno del gioco con la "texture" opportuna e di farla cambiare in base al suo stato e alle informazioni che riceve.
- Il **Controller** avrà il compito di creare i nemici all interno della mappa nelle apposite posizioni precendentemente definete ma soprattutto di far comunicare la model e la view tra di loro dato che queste classi sono totalmente separate, tra di loro non entrano mai in contatto.

In fase di progettazzione mi sono accorto che fondamentalmente le model dei nemici si possono divedere in due macroinsiemi: i nemici "dinamici" e i nemici "statici".

Dynamic Enemy
I nemici dinamici, come suggerisce il nome, sono quei nemici che durante il gioco hanno la possibilità di muoversi per la mappa interagendo con gli elementi che incontrano nel loro percorso cambiando così il loro stato(RINO).

Static Enemy
I nemici statici al contrario sono quei nemici che non avranno la possibiltà di muoversi, saranno gli altri elementi della mappa che interagiranno con loro(TURTLE).

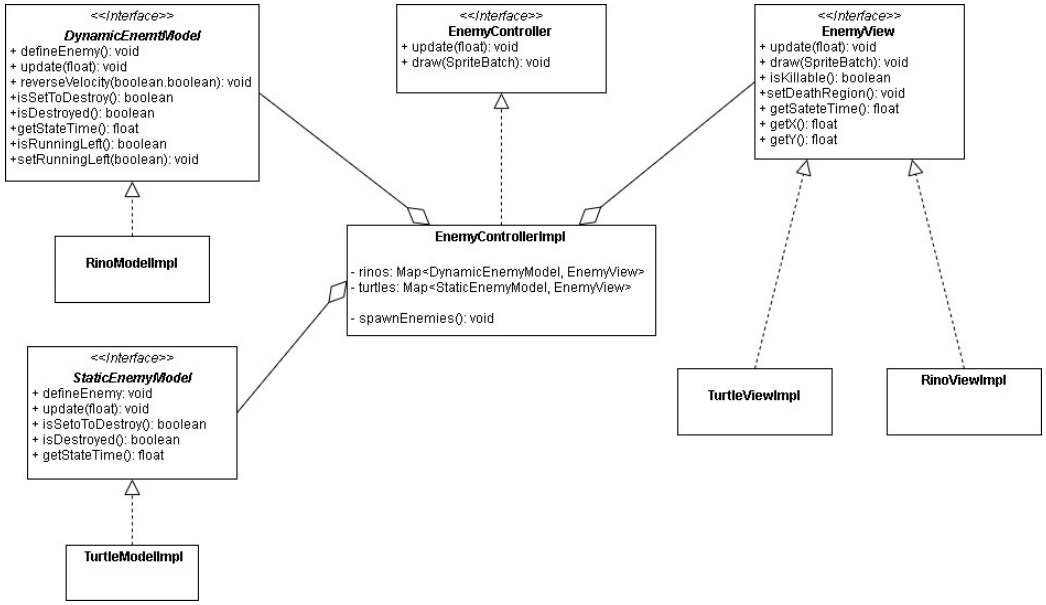

Figura 2.11: Schema UML dei nemici realizzato con il pattern MVC

Nella classe **EnemyController** della fig. 2.11 sono state volontarimente omessi numerosi metodi che passano informazioni tra la parte "model" e la parte "view" dei nemici per rendere più leggibile lo schema UML e perchè valutati di importanza secondaria.
Nella classe **EnemyControllerImpl** dove c'è la vera e propria essenza del controller ho deciso di gestire i due tipi di nemici in altrettante mappe, questo tipo di collezione mi è subito sembrato performante per gestire i gruppi di nemici dato che come chiavi abbiamo l'interfaccia model mentre come valore la corrispettiva interfaccia view. L'utilizzo della mappa ha anche semplificato notevolmente il passaggio di informazioni da model a view e viceversa.
Sempre nella stessa classe è stato inserito il metodo privato spawnEnemies il quale ha il compito di riempire le mappe con le classi che implementano le interfacce sopra elencate (il numero dei nemici e le loro rispettive posizioni sono stati concordati in fase di sviluppo e si possono rilevare direttamente dalla mappa). Come si puo' notare sempre dallo stesso schema il codice è sicuramente estensibile, in caso si voglia aggiungere un nuovo nemico bisognerà semplicemente decidere se lo si vuole di tipo dinamico, in tal caso la sua classe model dovrà implementare necessariamente l'interfaccia **"DynamicEnemyModel"** (ad esempio RinoModelImpl), in caso contrario se lo si vuole di tipo statico basterà implmentare l'interfaccia **"StaticEnemyModel"** (ad esempio TurtleModelImpl). Dopo aver fatto questa scelta in ogni

caso per la parte visiva si dovrà creare una classe che implementerà l'interfaccia **"EnemyView"** che è stata progettata in maniera molto generica per poter essere implementata da qualsiasi tipo di nemico. Come ultima operazione per fare in modo che il nemico progettato appaia nel mondo e che le sue parti "model" e "view" comunichino sarà necessario aggiornare la classe **"EnemyControllerImpl"** che appunto svolge questi compiti.

L'interfaccia Collidable

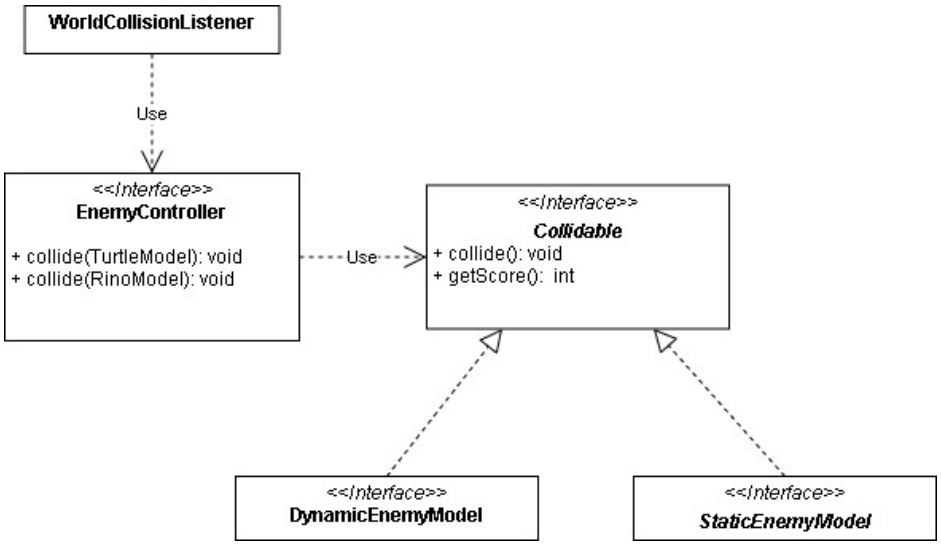

Figura 2.12: Schema dell'interfaccia collide applicata ai nemici.

Come mostrato nella fig. 2.12 le interfacce DynamicEnemyModel e StaticEnemyModel estendono l'interfaccia Collidable e di conseguenza prevedono nelle loro implentazioni la definizione dei metodi "collide", che definisce l'algoritmo di gestione della collisione e "get score" che ritorna il punteggio relativo alla data entità. Con l'estensione di tale interfaccia viene utilizzato il pattern Strategy, dato che viene definito il metodo collide (con uno specifico algoritmo) che sarà poi utilizzato all'oscuro della sua implementazione.

2.2.4 Marsild Spahiu

Sin dalla prima fase di analisi era chiara la necessità di dover creare un mondo di gioco nel quale i personaggi, i nemici ed i potenziamenti venissero generati e venissero a contatto oltre che tra di loro, anche con altre componenti di gioco tra cui, ad esempio, il terreno e gli ostacoli.
La parte della quale mi sono occupato riguardava appunto la creazione e la gestione di questi componenti del mondo fisico, la creazione di almeno due diversi livelli di gioco (background, grafica, posizionamento di ostacoli ed altre componenti) e l'implementazione del sistema di gestione delle collisioni tra le entità in gioco.

Level
Per rappresentare il concetto di livello inizialmente si era pensato all'utilizzo di una enum Level, ma questa soluzione andava contro alla definizione di "enumerazione" dato che esse andrebbero utilizzate per definire insiemi di valori che difficilmente cambieranno in futuro, inoltre nel caso si volesse aggiungere un nuovo livello, con una enum sarebbe difficile modificare il codice successivamente. Ho deciso quindi di utilizzare una interfaccia Level, con relative implementazioni (vedere fig. 2.13), che prevede la definizione di un metodo getMap(), il quale ritorna il nome del file della mappa sotto forma di stringa (utilizzato all'interno del PlayScreen). Con l'utilizzo di questo nuovo approccio viene consentita facilmente l'aggiunta di nuovi elementi e quindi viene favorita l'estendibilità del codice.

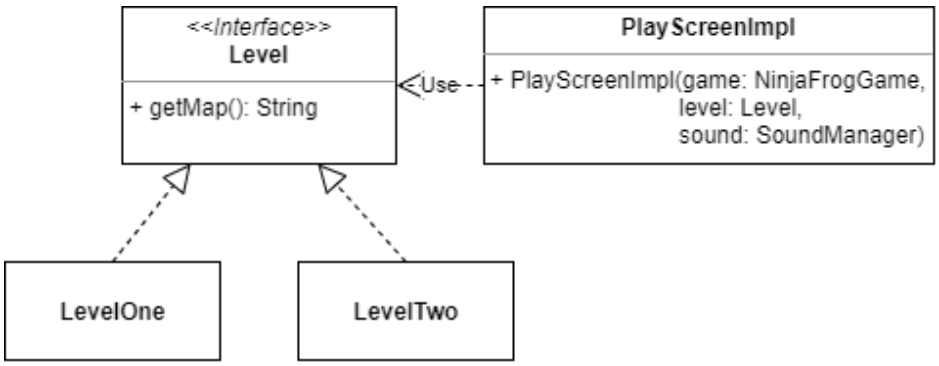

Figura 2.13: Schema UML dei Level.

WorldCreator

L'interfaccia WorldCreator e relativa implementazione, utilizzata nel PlayScreenImpl (vedere fig. 2.14) permette la creazione del mondo fisico a partire dai file .tmx delle mappe, presenti nella cartella "assets" del progetto Core. Da queste mappe vengono rilevati i diversi layer corrispondenti a diversi tipi di oggetto e, a secondo del tipo di layer, vengono generati gli oggeti desiderati, ai quali viene assegnato un bit il quale verrà poi utilizzato come identificatore nella gestione delle collisioni. Gli oggetti creati si possono dividire in due principali gruppi: "InteractiveObject" e "NonInteractiveObject" (descritti in dettaglio nelle successive sezioni), i quali, come anticipa il nome, sono differenziati dal fatto che i primi hanno un effetto alla collisione col personaggio principale, mentre gli ultimi no. Il WorldCreator genera gli oggetti non interattivi attraverso il NonInteractiveBuilder.

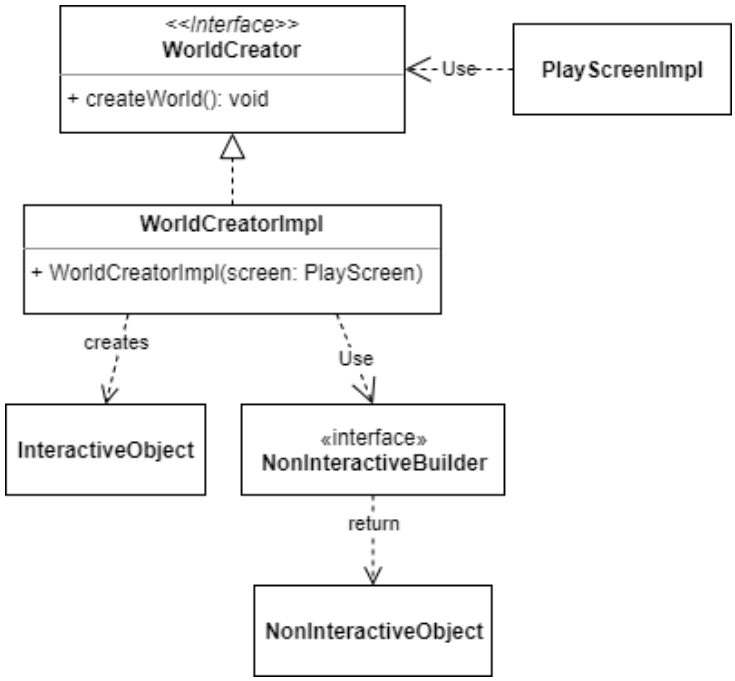

Figura 2.14: Schema UML del worldCreator.

InteractiveObject

Gli oggetti interattivi sono oggetti i quali, al contatto con il personaggio principale (ninja), cambiano "stato" e si comportano in un determinato modo, oltre ad incrementare il punteggio totale del gioco con uno specifico punteggio relativo alla propria entità. Per definire i due oggetti interattivi utilizzati in questo applicativo, cioè i FruitBox (che al contatto con la testa del ninja fanno comparire un frutto per poi diventare non interattivi) e i Brick (che una volta colpiti si distruggono) ho deciso di procedere nel seguente modo (fig. 2.15): le due sottoclassi estendono entrambe una classe chiamata "InteractiveObject", la quale definisce gli aspetti comuni ad ambedue gli oggetti, mentre singolarmente definiscono i metodi collide() e getScore() che ereditano dall'interfaccia Collidable, creata appositamente per esprimere il concetto di "entità" che può collidere col personaggio principale. Implementando questa interfaccia, viene applicato il pattern **Strategy**, dato che viene definito un algoritmo di gestione della collisione (metodo collide) il quale all'interno della WorldCollisionListener verrà chiamato all'oscuro della sua implementazione e indipendentemente dalla classe che aderisce al contratto.

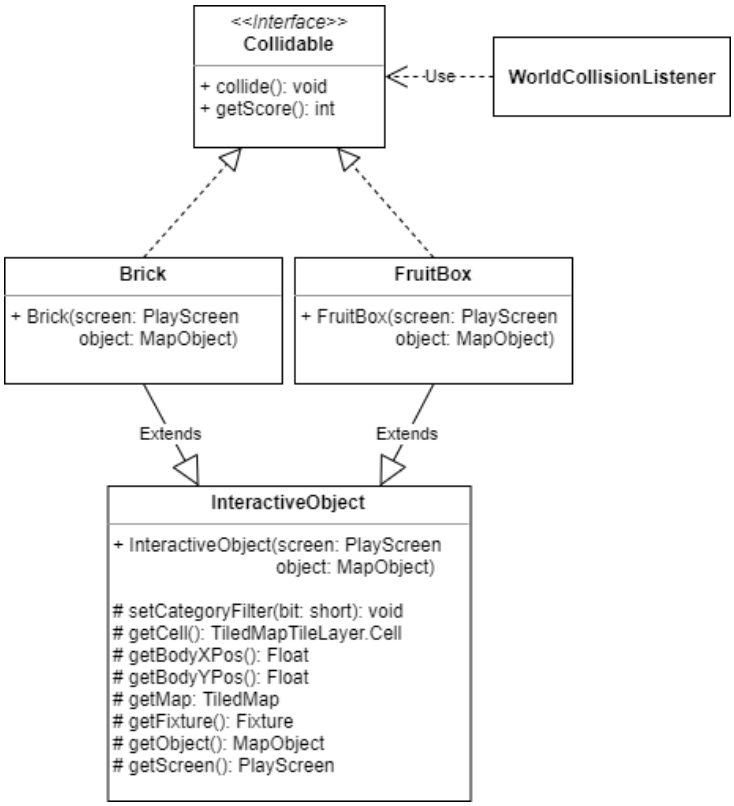

Figura 2.15: Schema UML degli oggetti interattivi.

NonInteractiveObject

Per la realizzazione di oggetti non interattivi, cioè il terreno (GROUND), gli ostacoli (GROUND_OBJECT) e il trofeo finale di vittoria (FINISH), ho deciso di utilizzare il pattern **Builder** (fig. 2.16). Questa scelta è stata effettuata per evitare le proliferazione di costruttori nella classe WorldCreatorImpl, definendo una strategia per la creazione step-by-step di questi oggetti. Essi, infatti, vengono praticamente generati allo stesso modo, con l'unica differenza del bit di schermatura (il quale permette di gestire in maniera differente le collisioni), il quale varia. Con il metodo chooseCategoryBit(short bit) è possibile impostare questo bit di schermatura, per poi generare l'oggetto definitivo col metodo build(). Come parametro del builder bisogna fornire il PlayScreen, mentre l'oggetto da generare si seleziona con il metodo selectObject(MapObject object). La scelta di utilizzare questo pattern permette l'estendibilità del codice, dato che il builder è riutilizzabile per definire altri oggetti non interattivi i quali magari possono dover essere gestiti in maniera differente nella collision listener.

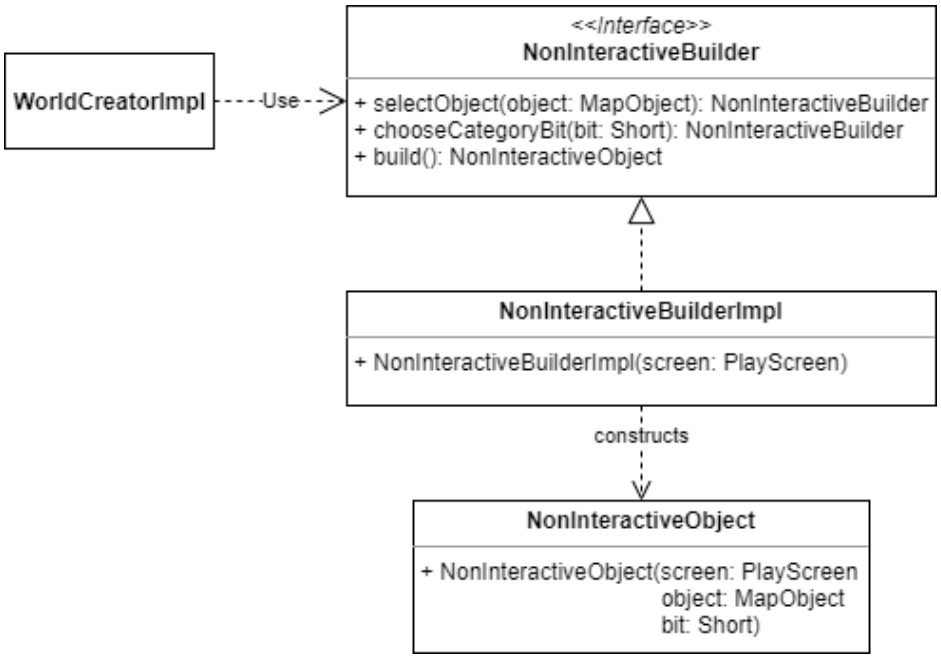

Figura 2.16: Schema UML degli oggetti non interattivi.

WorldCollisionListener

Il WorldCollisionListener, ovvero il gestore delle collisioni del mondo di gioco, è una implementazione dell'interfaccia di libreria ContactListener (fig. 2.17), la quale prevede la definizione di quattro diversi metodi "beginContact", "endContact", "postSolve" e "preSolve". Per realizzare la classe ho utilizzato solamente il primo di questi quattro metodi, che prevede di definire cosa deve succedere ad inizio di una collisione. Per gestire i diversi casi di contatto, prima ho rilevato le due fixture che sono entrate in collisione, e poi a seconda dei bit identificatori vengono gestite conseguentemente le azioni da compiere. In questa classe entra in gioco l'applicazione del pattern Strategy vista precedentemente, dato che in essa viene spesso chiamato il metodo "collide()" a prescindere dall'entità in gioco e quindi all'oscuro della sua implementazione, che viene lasciata alle singole classi.

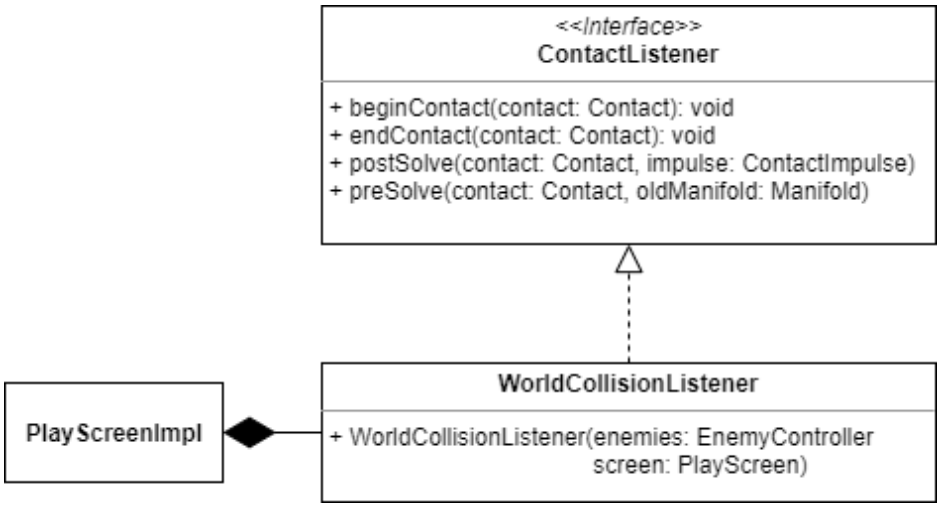

Figura 2.17: Schema UML del WorldCollisionListener.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

All'interno del progetto sono stati resi disonibili diversi test che consentono di verificare che non vi siano regressioni nelle funzionalità delle pricipali classi. Tutti i test sono stati raggruppati in un sottoprogetto chiamato "SuperNinjaFrog-tests" con relativo package e resi esguibili attraverso IDE utilizzando come unit test library pricipale JUnit, oppure attraverso il task di Gradle tests:test. Durante l'esecuzione di tali, si può andare ad esaminare la presenza di varie risorse fondamentali per il gioco e l'effettivo lancio di eccezioni come "IllegalStateException" o "NullPointerException" in caso di errata crazione di un oggetto.

3.2 Testing manuale

Sono stati effettuati anche test manuali per verificare il corretto funzionamento e la corretta eseguibilità dell'applicazione, trovando e risolvendo alcune incompatibilità tra i sistemi operativi Windows e Linux, soprattutto per quanto riguarda la gestione dei frame per secondo.
Per il testing manuale su sistemi Linux abbiamo utilizzato le seguenti distribuzioni: Lubuntu 32 bit LTS 16.04 e Ubuntu 64 bit LTS 20.4.

3.3 Metodologia di lavoro

Il progetto è stato svolto in modalità cooperativa, utilizzando git durante tutto lo sviluppo, come spiegato a lezione. Le fasi di inizializzazione del progetto, suddivisione del lavoro, analisi di dominio e la successiva di design architetturale sono state svolte in gruppo, attraverso piattaforme di comunicazione e collaborazione come Microsoft Teams oppure ove e quando possibile, in presenza. Allo stesso modo e attraverso anche altre applicazioni di messaggistica ci siamo aggiornati periodicamente sulla metodologia di lavoro e su come procedere nello sviluppo delle parti di lavoro comuni.
La suddivisione del lavoro, ideata per garantire la maggior indipendenza possibile nello sviluppo delle singole parti e definita dopo aver analizzato i requisiti dell'applicazione e le varie entità che lo compongono, risulta come segue:

3.3.1 Gianmaria Casamenti

Durante l'implementazione del programma ho gestito in maniera autonoma:

- Creazione e sviluppo dei vari menù e sottomenù di gioco con la relativa ricerca delle risorse necessarie come ad esempio asset, background, skin e font.
- Gestione dei potenziamenti durante il gioco, in cui ho analizzato i GoF Design pattern per trovare quelli più adatti al mio scopo e che avessero un impatto positivo sul mio codice.
- Realizzazione delle due schermate di sconfitta e vittoria.
- Implementazione di tutti i test, riguardanti le mie parti, con l'utilizzo di JUnit e l'apposito tool di Gradle.

3.3.2 Lorenzo Leoni

Durante lo sviluppo del progetto ho gestito in maniera autonoma:

- Hud.
- Sound Manager.
- Player.
- Test con Junit relativi alle mie classi.

3.3.3 Luca Pasini

Durante la realizzazione del programma ho gestito in maniera autonoma:

- Progettazione e creazione di interfacce e classi relative ai nemici.
- Test con Junit relativi alle mie classi.

3.3.4 Marsild Spahiu

Durante lo sviluppo del progetto ho gestito in maniera autonoma:

- Creazione e gestione dei livelli del gioco (interfaccia Level), con la conseguente ricerca delle relative risorse necessarie (background, asset e tutto il necessario per la creazione dei file delle mappe attraverso il map editor, vedere le Note di sviluppo).
- Gestione delle classi per la creazione del mondo fisico, cioè l'interfaccia WorldCreator, la sua implementazione, e le relativi classi ed interfacce da essa utilizzate (oggetti interattivi e non interattivi).
- Gestione delle collisioni del gioco, attraverso la classe WorldCollisionListener.
- JUnit test relativi alle mie classi.

Durante lo sviluppo dell'applicativo è stato necessario convenire in comune sulla definizione di diverse interfacce, per garantire la necessaria indipendenza nello sviluppo delle singole parti ai diversi membri del gruppo. La definizione dell'interfaccia PlayScreen e, di conseguenza, della sua implementazione, ad esempio, è stata realizzata in maniera cooperativa da tutti gli elementi del gruppo, così come la definizione della maggior parte delle interfacce utilizzate dal PlayScreen e l'implementazione con conseguente documentazione di diverse altre classi (ad esempio, GameConst). La loro effettiva stesura è stata però affidata a singoli componenti del gruppo per evitare problemi di coordinazione nella loro gestione.

3.4 Note di sviluppo

Data la nostra inesperienza riguardo lo sviluppo di progetti di gruppo, e più in particolare nello sviluppo di giochi/grafica 2D, abbiamo deciso, dopo esserci documentati online, di utilizzare la libreria LibGDX (servendoci, quando necessario, della sua documentazione), la quale presentava già un tool di setup, e abbiamo preso spunto da diversi video tutorial (Brent Aureli).
Per lo sviluppo dei test siamo partiti da uno scheletro di partenza di test JUnit e Mockito per progetti LibGDX (TomGrill/gdx-testing).

3.4.1 Gianmaria Casamenti

Nella creazione della classe FruitBuilderImpl ho impostato il costruttore privato, per non renderlo più visivibile e ho aggiunto un nuovo metodo statico newBuilder() che ritornava l'oggetto stesso. Questa scelta ha portato al vantaggio di non dover più istanziare la classe FruitBuilderImpl. Durante tutto lo sviluppo del codice ho fatto uso della libreria LibGdx per la visualizzazione dei menù e ampio utilizzo di Gradle e SpotBugs.

3.4.2 Lorenzo Leoni

Sviluppando la parte del Player ho sfruttato i metodi e i tipi forniti dalla libreria libGDX, per la parte relativa all'Hud ho deciso di utilizzare le Label e i Table di libGDX invece di quelle di javaFX poichè più comode da utilizzare per interfacciarmi meglio con gli altri colleghi.
Oltre a questo per la parte audio, so che non conta nella valutazione e nel conteggio delle ore, ho composto la canzone del menù e del gioco con l'aiuto di mio fratello e l'ausilio di Ableton.

3.4.3 Luca Pasini

Durante lo sviluppo del mio codice ho constatato come la decisione di utilizzare il pattern architetturale MVC sia stata a mio avviso corretta, dividere il codice del nemico in tal maniera lo rende implementabile e di facile comprensione.
Come feature avanzate ho utilizzato:

- Lambda expressions nella classe EnemyControllerImpl come "consumer" per i cicli "foreach"
- Reflection nella classe EnemyControllerImpl per la ricerca all'interno del layer mappa dei nemici

3.4.4 Marsild Spahiu

Per la creazione dei due livelli di gioco ho utilizzato il map editor Tiled, documentandomi online su come utilizzarlo al meglio e su come importare le due mappe (file .tmx) all'interno del codice.
Per la gestione delle collisioni del gioco, quindi per la creazione della classe WorldCollisionListener, mi sono servito dell'interfaccia di libreria ContactListener.
Come feature avanzate ho utilizzato:

- Lambda expressions nella NonInteractiveBuilderImpl all'interno del metodo filter() e in Brick per la definizione di un nuovo Thread.
- Reflection nella WorldCreatorImpl per ricercare la presenza di determinati tipi di oggetto all'interno dei layer della mappa.
- `Optional` nella NonInteractiveBuilderImpl.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Gianmaria Casamenti

Sono felice del progetto creato e di aver avuto la possibilità di poter programmare un gioco per la prima volta. Mi sono accorto che un videogioco è un esercizio progettistico che richiede diverse skill di programmazione.
Se dovessi analizzare i lati positivi indicherei, sicuramente, il fatto che ho avuto modo di e approfondire diversi argomenti ed aver imparato a collaborare con un team di lavoro.
Al contrario, invece se dovessi elencare i lati negativi sarebbero che, in un altro progetto, userei la libreria LibGdx in maniera differente; ma credo che questo errore di valutazione sia stato dettato anche dall'inesperienza.

4.1.2 Lorenzo Leoni

Essendo il primo progetto di medio/grandi dimensioni ho avuto difficoltà soprattutto nella parte di progettazione, poichè anche la parte meno affrontata all'interno del corso, ma nonostante questo mi ritengo abbastanza soddisfatto. La parte positiva del progetto è stata sicuramente il poter applicare e approfondire quando visto durante le lezioni e i laboratori. L'aspetto negativo invece, dettato per lo più della mancanza di esperienza, è stato della scelta della libreria, ottima per la resa finale del gioco ma meno per la progettazione iniziale inoltre avrei potuto utilizzare più feature avanzate di java dove possibile.
Dopo quanto fatto in questo progetto di esame mi troverei sicuramente più preparato e pronto ad affrontare un nuovo lavoro di questo genere, avendo così più consapevolezza di cosa fare e di cosa, invece, evitare.

4.1.3 Luca Pasini

Mi ritengo piuttosto soddisfatto del risultato essendo questo tipo di progetto molto piu sofisticato di tutti gli altri esguiti fin ora. Le parti iniziali di "analisi" e "progettazione" sono state le più lunghe e complesse al fine di dare buone basi al codice data la nostra inesperienza in questo tipo di lavori. Nonstante ciò abbiamo capito in un secondo momento che la nostra decisione iniziale di utilizzare libGDX ci avrebbe vincolato per quanto riguarda l'utilizzo nella maniera più opportuna dei pattern. Facendo questo progetto ho capito quanto siano importanti parti della programmazione come la "java doc" per rendere i tuoi metodi e le tue classi piu chiari anche a tutti quelli che non le hanno scritte, il fatto che una buona progettazione dettata dalle tue conoscenze sia la base poi di tutto il progetto, di utilizzare i pattern per rendere più leggibile, estendibile ma anche modificabile il codice. La comunicazione con tutto il team di sviluppo è stata costante in questi due mesi, creando così un clima di lavoro stimolante e produttivo. Mi sento appagato dall' aver potuto utilizzare per la prima volta un linguaggio di programmazione per creare un qualcosa di sofisticato e non un esercizio fine a se stesso.

4.1.4 Marsild Spahiu

Essendo il primo progetto di gruppo che ho svolto in ambito universitario utilizzando git/github, mi ritengo abbastanza soddisfatto del lavoro che siamo riusciti a portare a termine, data la mia, e in generale dei miei colleghi, inesperienza in questo campo. Penso di aver imparato molto da questa esperienza, su diversi fronti: ho capito il vero significato di "analisi" e "progettazione", ho imparato a coordinarmi con i miei colleghi nell'utilizzo del DVCS, ho utilizzato per la prima volta Gradle comprendendo la sua utilità e, soprattutto, ho compreso l'importanza dei pattern ed il vero significato delle interfacce, che prima avevo superficialmente visto e compreso solo ad ambito "didattico", mentre ora ho capito il fondamentale ruolo che assumono nella cooperazione dei lavori di gruppo. Sono inoltre soddisfatto del risultato che abbiamo ottenuto nella realizzazione di questo gioco, soprattutto a livello estetico, anche se mi rendo conto che l'utilizzo di LibGDX non è stato proprio limitato alla view, come invece auspicavano i professori. Reputo comunque questo progetto un ottimo punto di inizio per progetti futuri, essendo diventato più esperto sia per quanto riguarda gli aspetti tecnici di DVCS, gradle, ecc..., sia per quanto riguarda le fasi di analisi e progettazione che precedeno l'effettivo inizio della programmazione vera e propria.

Probabilmente in futuro, tempo ed esami permettendo, procederemo a sviluppare maggiormente il gioco, creando inizialmente nuovi livelli, e poi magari aggiungendo nuovi suoni, nuove animazioni o addirittura nuove feature come nuovi personaggi, potenziamenti o nemici, giusto per lo sfizio di mostrare a parenti ed amici un gioco bello, completo ed avvincente realizzato da noi.

4.2 Difficoltà incontrate e commenti per i docenti

- **Lorenzo Leoni**: fino ad ora questo corso è stato il più interessante e stimolante, mi è dispiaciuto però il poco tempo utilizzato per affrontare la parte relativa ai pattern, che si è dimostrata poi fondamentale durante la fase di sviluppo del progetto, essendo anche molto più difficile da appronfondire in autonomia.

Appendice A

Guida utente

A.1 Avvio e Menù

All'avvio dell'applicativo viene mostrato il menù principale nel quale ci si può muovere con i tasti "WS" oppure con le frecce direzionali verso l'alto e verso il basso, ed è possibile selezionare la propria scelta con il tasto "Enter". Le istruzioni sopraelencate sono valide anche per gli altri sotto-menù.

Figura A.1: Rappresentazione del menù principale all'apertura del gioco

A.2 Gameplay

Confermando le scelte precedentemente selezionate si darà inizio alla partita. All'interno del livello è possibile muoversi con la sequenza di tasti "WAD" oppure con le frecce direzionali.
All'interno della mappa si possono incontrare nemici, i quali possono essere evitati oppure uccisi saltandoci sopra (evitando le spine nel caso delle Turtle) e sono presenti anche oggetti bonus come ciliegie (aggiungono una vita), meloni (attivano il potenziamento del giocatore) e arance (aumentano il punteggio). Il gioco termina con una sconfitta se si viene uccisi da un nemico oppure se si cade, mentre termina con una vittoria se si raggiunge il trofeo finale. All'inizio del primo livello è comunque presente un mini tutorial del gioco.
Mentre si sta giocando, se necessario, si può passare allo stato di pausa premendo la barra spaziatrice e, per riprendere oppure per tornare alla schermata dei menù, ripremere la barra spaziatrice oppure premere ESC (seguire, in ogni caso, le istruzioni indicate all'interno della schermata che compare).

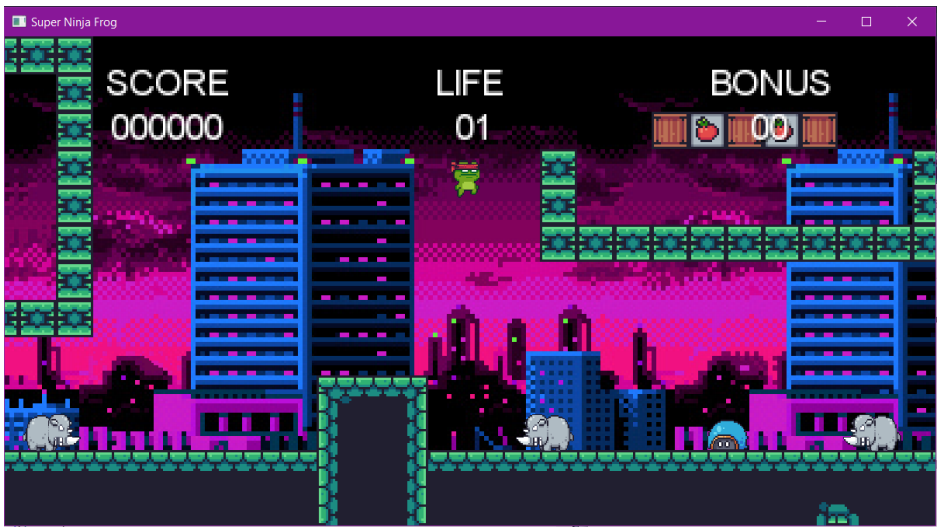

Figura A.2: Esempio di schermata visibile durante il gioco

Appendice B

Esercitazioni di laboratorio

B.1 Casamenti Gianmaria

- Laboratorio 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=62684#p105203
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=62582#p105205
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=63865#p105206
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=64639#p105208
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=66753#p106754

B.2 Lorenzo Leoni

- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=62582#p121213
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=63865#p121212
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=64639#p121211
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=66753#p121214

B.3 Luca Pasini

- Laboratorio 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=62684#p121204
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=62582#p121202
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=63865#p121200
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=64639#p121203

B.4 Marsild Spahiu

- Laboratorio 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=62684#p102899
- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=62579#p102902
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=62582#p102906
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=63865#p102908
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=64639#p103778
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=66753#p106613