"Relazione per
"Programmazione ad Oggetti"

Antonelli Giacomo, Garofalo Giorgio, Ghignatti Nicolò, Patrignani Luca

Indice

Capitolo 1

Analisi

1.1 Requisiti

Ispirato al celeberrimo "Super Mario Bros." del 1985, il nostro videogioco è un platformer 2D che segue il personaggio principale - da qui in poi chiamato anche protagonista - che deve essere guidato dall'utente fino alla fine di ogni livello, superando ostacoli e sconfiggendo nemici con differenti caratteristiche e punti deboli che appariranno durante il suo percorso.
All'avventura si possono unire fino ad un totale di 4 protagonisti.

Requisiti funzionali

- Il nostro protagonista potrà:
  - Muoversi all'interno del livello spostandosi a destra e sinistra, saltando verso l'alto o cadendo in basso per effetto della gravità;
  - Raccogliere potenziamenti ("power-ups") che modificano il suo aspetto conferendogli abilità speciali;
  - Sconfiggere i nemici o essere danneggiato e perdere vite;
  - Interagire e distruggere certi blocchi da cui possono fuoriuscire potenziamenti o monete, o che possono semplicemente rompersi;
- Sarà possibile ottenere delle statistiche come punteggio attuale e monete raccolte.
- Il livello viene completato se tutti i protagonisti raggiungono la bandiera finale.
- Sarà necessario giocarlo nuovamente dall'inizio in caso di sconfitta. Una sconfitta può avvenire se:

  - Il protagonista cade in un burrone;
  - Il protagonista viene colpito da un nemico, a meno che il tipo di contatto sia adatto a sconfiggere il nemico, senza un power-up attivo. Se si ha almeno un power-up attivo viene perso quello più significativo.

- Il protagonista potrà incontrare lungo il suo cammino alcuni tipi di nemici:
  - Goomba: un piccolo funghetto che si muove orizzontalmente. Può essere sconfitto saltando sulla sua testa.
  - Koopa: una tartaruga che si muove orizzontalmente. Saltando su di essa si chiuderà nel suo guscio che, se colpito di nuovo, inizierà a scivolare verso la direzione desiderata sconfiggendo tutti i nemici (e anche il protagonista) sul suo cammino.

- Il protagonista potrà raccogliere alcuni tipi di potenziamenti:
  - Fungo: rende il protagonista più alto e capace di rompere alcuni blocchi.
  - Fiore: permette al protagonista di sparare palle di fuoco che sconfiggono il primo nemico che incontrano.

- Il protagonista potrà incontrare lungo il suo percorso dei blocchi inanimati come:
  - Blocco sorpresa (punto interrogativo): può produrre un power-up (determinato dallo stato del protagonista) o monete e dopo essere stato distrutto si trasforma in un blocco generico.
  - Blocco distruttibile: viene distrutto una volta colpito da sotto mentre il protagonista è in possesso del power-up fungo o fiore.
  - Blocco generico/di ambiente: generico blocco indistruttibile.

Requisiti non funzionali

- L'applicazione dovrà essere in grado di girare su qualsiasi hardware, anche quelli più datati;

- L'interfaccia grafica deve essere in grado di ridimensionarsi automaticamente in base alla risoluzione dello schermo e alla grandezza dello schermo;

- Non dovranno insorgere errori durante il gameplay, quali il personaggio all'interno di un blocco o crash durante esso.

1.2 Analisi e modello del dominio

Il giocatore gioca consecutivamente a dei livelli. Ogni livello (`Level`) ha un suo mondo (`World`) il quale contiene delle entità (`Entity`), cioè ogni oggetto che ha una posizione e una dimensione.

Ogni mondo inoltre tiene traccia dei giocatori (`Player`) e ne gestisce i comportamenti quando vengono intercettati determinati input.
La difficoltà primaria sarà quella di permettere al giocatore di uccidere i mostri, schiantarsi contro i muri, saltare e tutte quelle azioni che scaturiscono da collisioni tra il `Player` e le altre entità.
Ciò è valido anche per le altre entità nemiche, le quali dovranno ostacolare il giocatore cercando di colpirlo, che dovranno essere dotate di un minimo di intelligenza. Altro punto importante sarà la gestione del punteggio del giocatore, il quale accumulerà punti in base all'uccisione di mostri (`Enemy`) o la raccolta di monete (`Coin`).

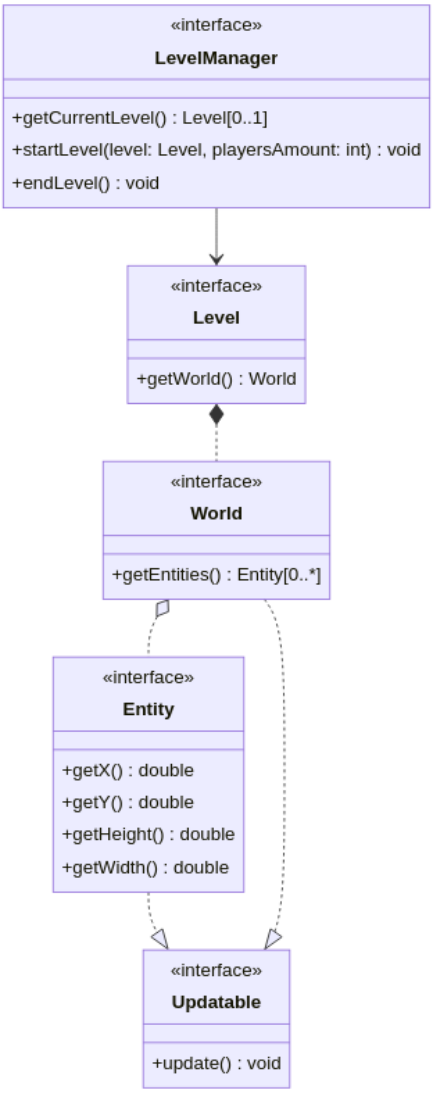

Figura 1.1: Schema UML dell'analisi di quello che dovrebbe essere la gestione del `Level` e del `World`, con le principali entità in gioco

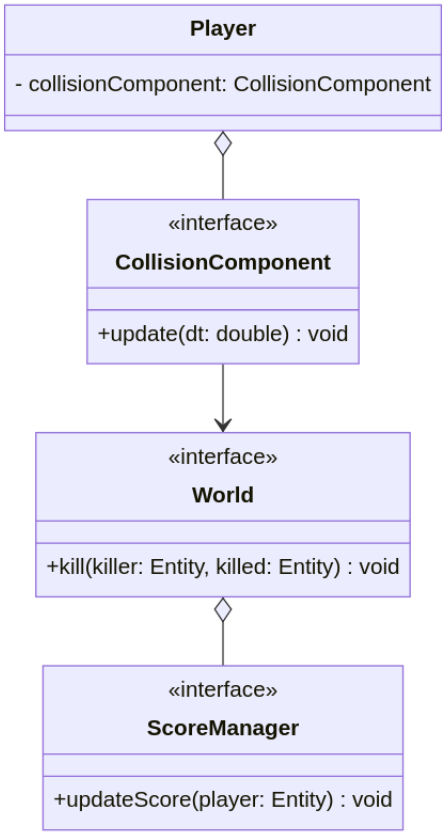

Figura 1.2: Schema UML dell'analisi di quello che dovrebbe essere la gestione dell'uccisione delle entità e l'accumulo di punti del player

Capitolo 2

Design

2.1 Architettura

L'architettura di Pixformer segue il pattern architetturale MVC. Poiché la ricezione degli input e la produzione dell'output sono due compiti distinti, si è deciso di creare due infrastrutture indipendenti, una responsabile di recepire gli input dell'utente da mouse e tastiera (view), mapparli in comandi e farli eseguire dai personaggi di gioco (model), ed un'altra che si occupa di portare i dati di output dal model alla view per la loro rappresentazione grafica. Per rispettare la correttezza del pattern utilizzato, tali comunicazioni tra model e view avvengono sempre passando per il controller.

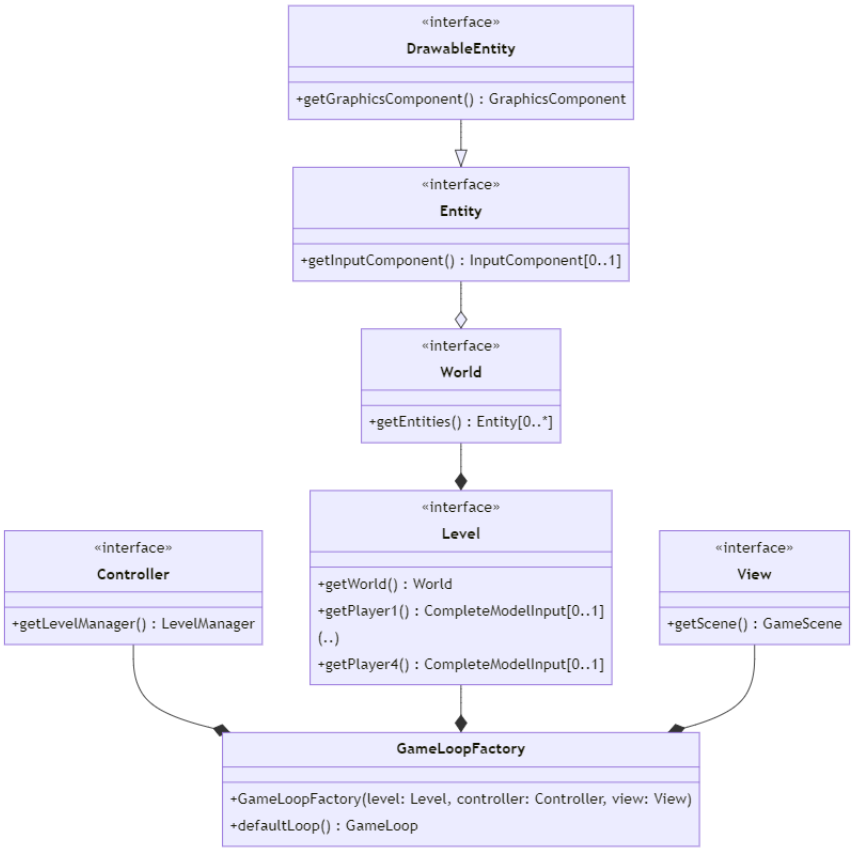

Figura 2.1: Diagramma UML dell'architettura MVC.

Nel model le entità sono caratterizzate dai loro componenti (`Component`) sulla base di un modello ECS[1]:

- l'`InputComponent` controlla i movimenti dell'entità;
- il `PhysicsComponent` si preoccupa di applicare le forze esterne sull'entità;
- il `CollisionComponent` gestisce gli eventi legati alle collisioni che riguardano l'entità;
- il `GraphicsComponent` determina la strategia per rappresentare l'entità. Vengono implementati dalla view e passati tramite *abstract factory pattern*. Questo permette il rispetto del MVC e permette di cambiare

[1] *Entity Component System*: `https://gameprogrammingpatterns.com/component.html`

le strategie di rappresentazione delle entità senza influenzare il model 2.2.

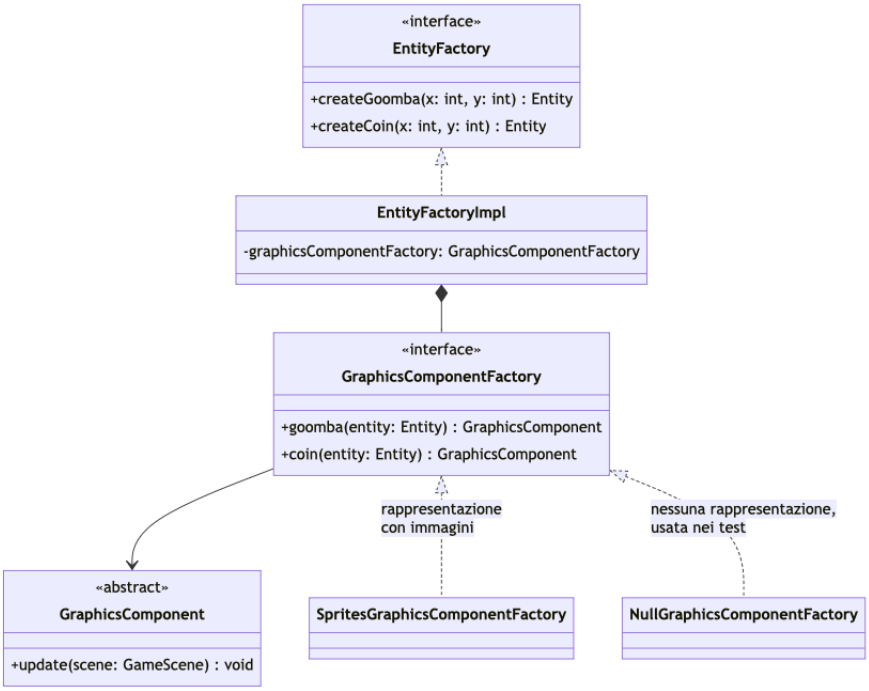

Figura 2.2: Diagramma UML che esplicita l'uso di *abstract factory* per la creazione dei `GraphicsComponent`.

2.2 Design dettagliato

2.2.1 Antonelli Giacomo

Giocatore principale

Problema Gestione dell'input del giocatore e comportamento del Player
È necessario gestire in modo efficace l'input proveniente dai giocatori, e far corrispondere queste azioni a un comportamento appropriato del Player nel gioco.

Soluzione Implementazione di un'architettura di tipo `Component Pattern`
Per questo problema è stata adottata un'architettura di tipo `Component Pattern`. Questo approccio prevede la suddivisione del carico di lavoro della

gestione del `Player` in diverse classi collegate tra loro, ognuna delle quali si occupa di un aspetto specifico. Ad esempio avremo una classe che gestisce l'input del giocatore, una che ne gestisce le collisioni ecc. Questo approccio modulare e ben strutturato ci ha consentito di gestire in modo efficiente l'input del giocatore e il comportamento del `Player`, mantenendo una chiara separazione delle responsabilità e facilitando la manutenzione e l'estensibilità del codice.
Nota: in questa sezione con gestione di input e collisioni non si intende la loro verifica nel gioco ma solamente della loro gestione una volta notificati al `Player`.

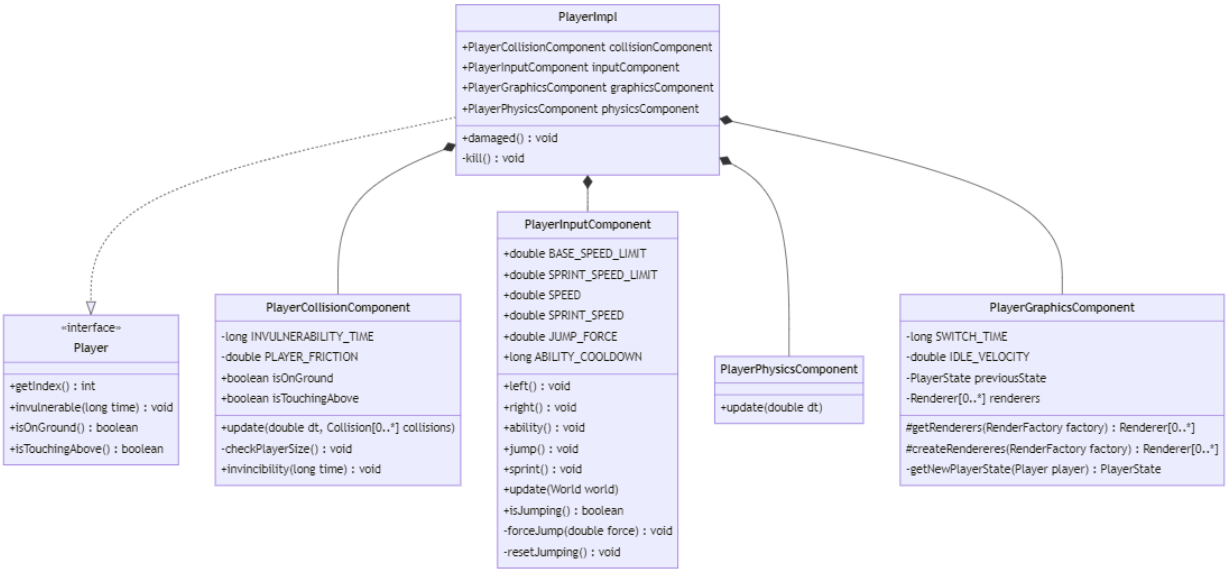

Figura 2.3: Schema UML del `Player` e dei suoi componenti.

Problema: Gestione dei Powerup
Uno dei nostri principali obiettivi è riuscire a schematizzare in modo semplice, efficiente e scalabile i `Powerup` del gioco e le loro peculiarità, considerando inoltre la gestione delle priorità tra di essi (ex. None < `Mushroom` < `FireFlower` = IceFlower(non implementato) < ?).

Soluzione: Implementazione dello Strategy Pattern
É stata adottata una soluzione basata sulla struttura del pattern di progettazione `Strategy`. É stata separata l'abilità del `Powerup` dall'oggetto `Powerup` stesso, in modo che il cambiamento di `Powerup` non necessiti la ricreazione dell'oggetto, ma piuttosto una modifica del comportamento associato all'oggetto esistente. In questo modo, l'oggetto `Powerup` rimane lo stesso, ma l'abilità (Behaviour) ad esso associato può cambiare.

Per la gestione della priorità fra `Powerup` è stato utilizzato invece, un approccio di incapsulamento, attraverso il quale, quando l'entità andrà ottenere un potenziamento di priorità superiore, questo incapsulerà al suo interno quello precedente, potendo così tornare, in caso di danno subito dall'entità, allo stato precedente. Ad esempio, se un personaggio possiede già un `Mushroom` e raccoglie un `FireFlower`, l'oggetto `Powerup` con il `Mushroom` viene inglobato all'interno di un campo del nuovo oggetto `Powerup` che avrà il behaviour del `FireFlower`, se invece il nuovo `Powerup` avrà la stessa priorità dell'attuale il behaviour del nuovo `Powerup` andrà a sostituire quello dell'attuale. Questo ci consente di gestire il cambio di `Powerup` con priorità diverse in modo coerente e senza complicare eccessivamente la logica di gioco.

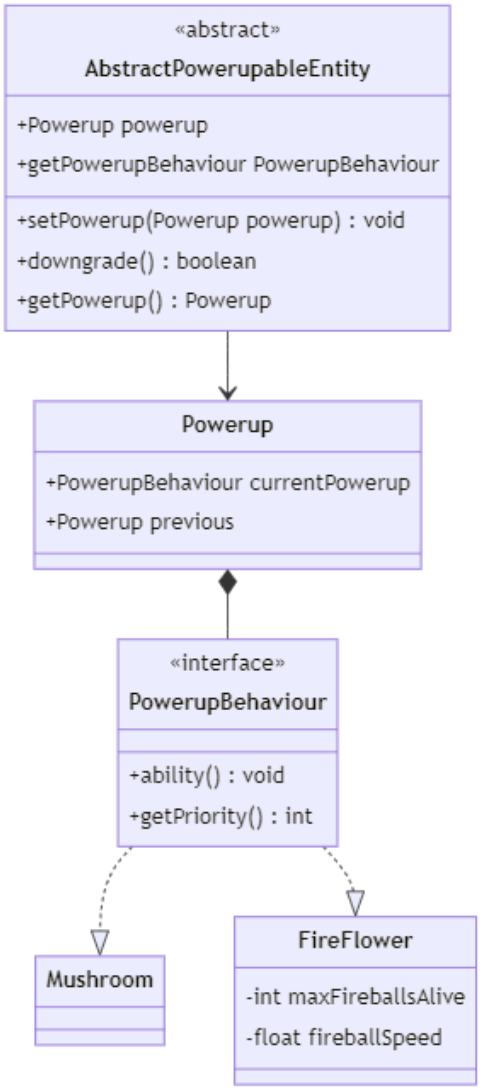

Figura 2.4: Rappresentazione UML della struttura logica dei `Powerup`.

Problema Gestione della visualizzazione del Player tenendo in considerazione dei numerosi stati che può avere.

Soluzione La gestione effettiva della visualizzazione del `Player` è affidata al suo `Graphics Component`. All'interno di questo componente vengono utilizzate una variabile `PlayerState`, che riassume alcune informazioni sullo stato del giocatore in un record per ottenere una visione più chiara, e una `PlayerSpriteFactory`, che genera una lista di `Renderer` in base alle informazioni ottenute dal `PlayerState`. Questi `Renderer` vengono poi passati alla view per essere visualizzati.

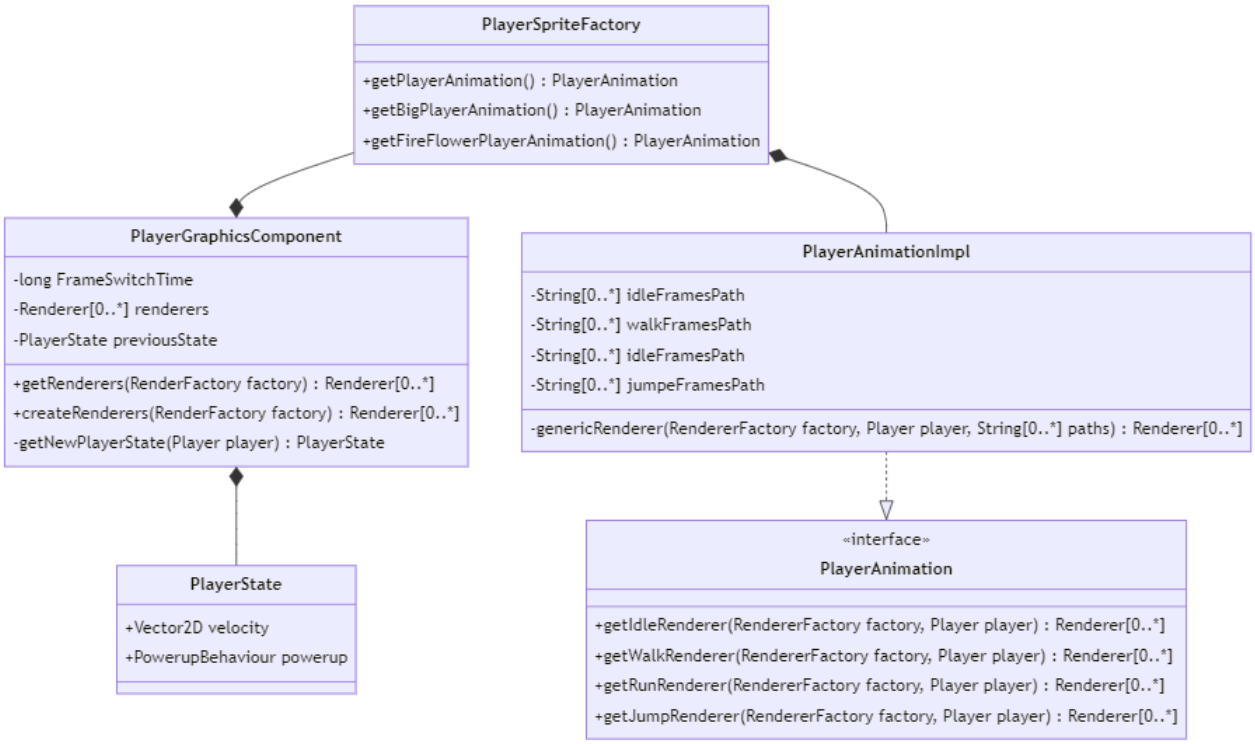

Figura 2.5: Schema UML della gestione del `GraphicsComponent` del `Player`.

2.2.2 Garofalo Giorgio

Engine grafico multi-framework

Problema Realizzare un game engine in grado di funzionare con più framework grafici. Una funzionalità importante del nostro game engine è la possibilità di disegnare la view utilizzando, a livello più basso, qualsiasi framework grafico per il raccoglimento degli input ed il rendering su schermo.

Soluzione È stato realizzata una struttura che, partendo da un `ViewLauncher` inizializza l'engine per il tipo di framework in uso. La suddivisione in rendering e input viene riconciliata da `GameScene`. La versione finale del progetto contiene solamente l'implementazione in JavaFX, ma è possibile supportare con facilità Swing, o anche il terminale con l'utilizzo di caratteri ASCII per il rendering.

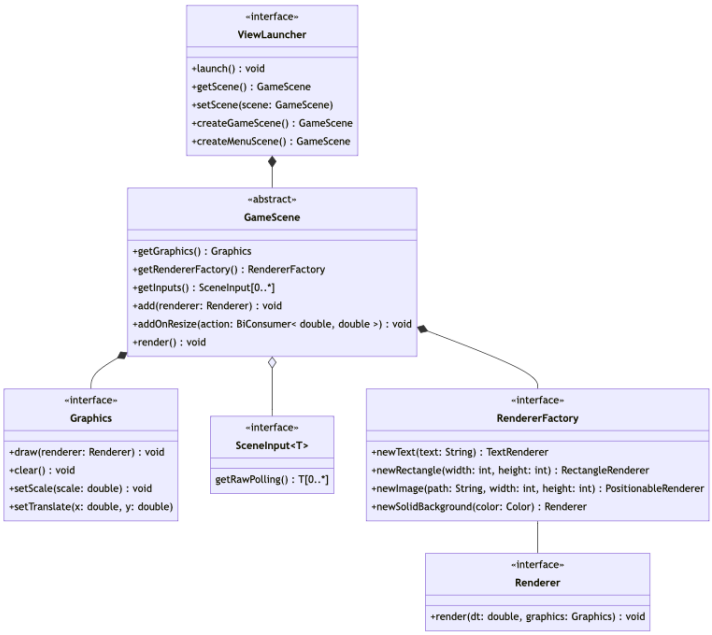

Figura 2.6: Schema UML dell'engine grafico

Ogni nodo della scena, qui chiamato `Renderer`, va istanziato unicamente dalla *abstract factory* `RendererFactory`. Altri design pattern utilizzati sono *Observer* con `GameScene#addOnResize` e *Singleton* con la classe `Lang` per l'internazionalizzazione.

L'aggiunta del supporto ad un framework grafico prevede l'implementazione delle interfacce presenti nello schema. Le implementazioni, ad esempio `JavaFXViewLauncher` e `JavaFXScene`, sono state omesse dallo schema per semplicità visiva.

Gestione delle collisioni

Problema La gestione delle collisioni è essenziale per la giocabilità dei livelli: le entità devono poter camminare sui blocchi solidi senza cadere per effetto della gravità, il protagonista sconfigge i nemici saltandovi sopra o può essere danneggiato da essi, raccoglie power-ups e monete e completa il livello quando tocca la bandiera finale. Si vuole quindi creare un sistema di collisioni riutilizzabile.

Soluzione La soluzione adottata associa ad ogni entità una bounding box in grado di determinare l'esistenza di una collisione con altre bounding box ed eventualmente anche il lato della collisione. Ad ogni chiamata del game loop le collisioni di ogni entità vengono passate al proprio `CollisionComponent` che, se esiste, si occupa di gestire le conseguenze di ogni collisione. Se un'entità viene marchiata come solida non può essere attraversata da altre entità. Come per le altre sottoclassi di `Component`, il suo utilizzo in `Entity` viene integrato nel pattern *Strategy*.

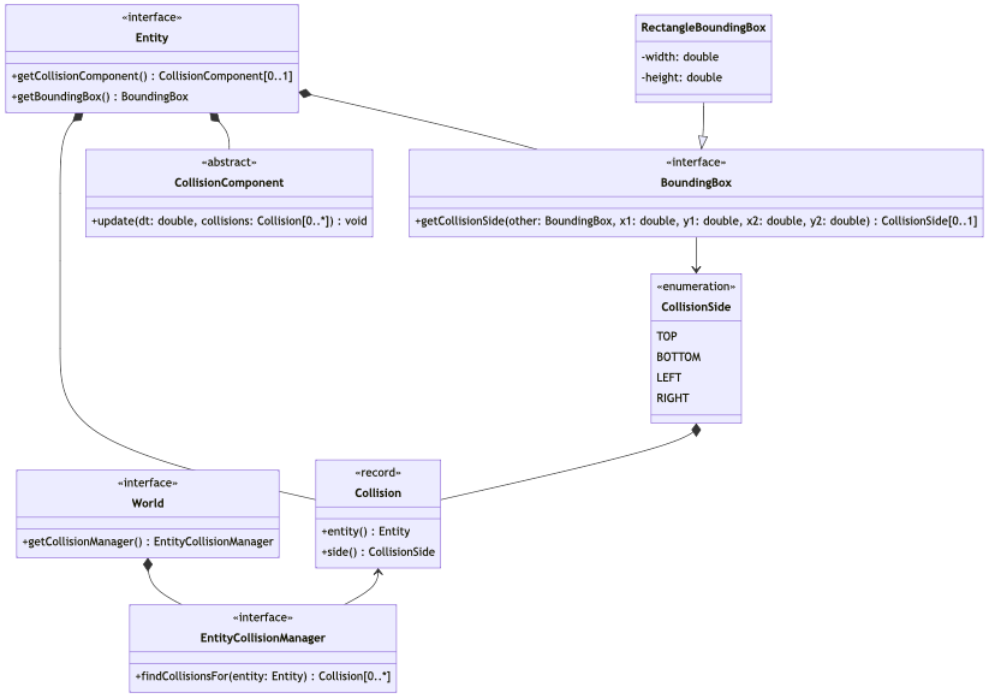

Figura 2.7: Schema UML del collision system

Alternativa L'approccio iniziale non prevedeva un `CollisionComponent`, bensì il controllo delle collisioni era gestito in zone sparse del codice. Ad esempio, un `InputComponent` poteva controllare se l'entità fosse a terra per poter resettare il salto. Abbiamo scartato questa soluzione al fine di garantire:

- un riuso più consistente, permettendo alle entità di "scambiare" i loro components. Ad esempio la maggior parte dei `CollisionComponent` delle entità dinamiche estendono `SolidCollisionComponent`, che impedisce all'entità di entrare in un blocco solido;
- prestazioni migliori, in quanto ogni controllo delle collisioni viene eseguito con complessità $O(n^2)$. Nella soluzione finale viene eseguito solo una volta per ogni entità ad ogni frame.

Lifecycle del livello

Problema Si vuole realizzare un sistema che:

- faccia vivere le entità nel mondo;
- permetta di iniziare e terminare l'esecuzione di un livello in modo indipendente dalla view nella sua implementazione, ma che riesca ad aggiornare la scena corrente (menu $\iff$ scena di gioco);

Soluzione Per quanto riguarda inizio e fine del livello, si è creato un `LevelManager` incluso nel `Controller` centrale che può aggiungere dei listener ai due eventi grazie al pattern *Observable*.

L'esecuzione del livello è invece gestita da un *game loop* che viene ripetutamente chiamato da un `GameLoopManager` la cui implementazione dipende dal framework grafico utilizzato. Nell'implementazione per JavaFX, il `JavaFXGameLoopManager` si basa su un `AnimationTimer`. Il mantenimento del frame rate all'interno del game loop manager è stato implementato da Nicolò ed è descritto da lui.

La *factory* `GameLoopFactory` fornisce le possibili implementazioni dell'interfaccia funzionale `GameLoop`. Al momento `defaultLoop()` è l'unica implementazione disponibile, ma non si esclude la possibilità di avere diversi tipi di game loop.

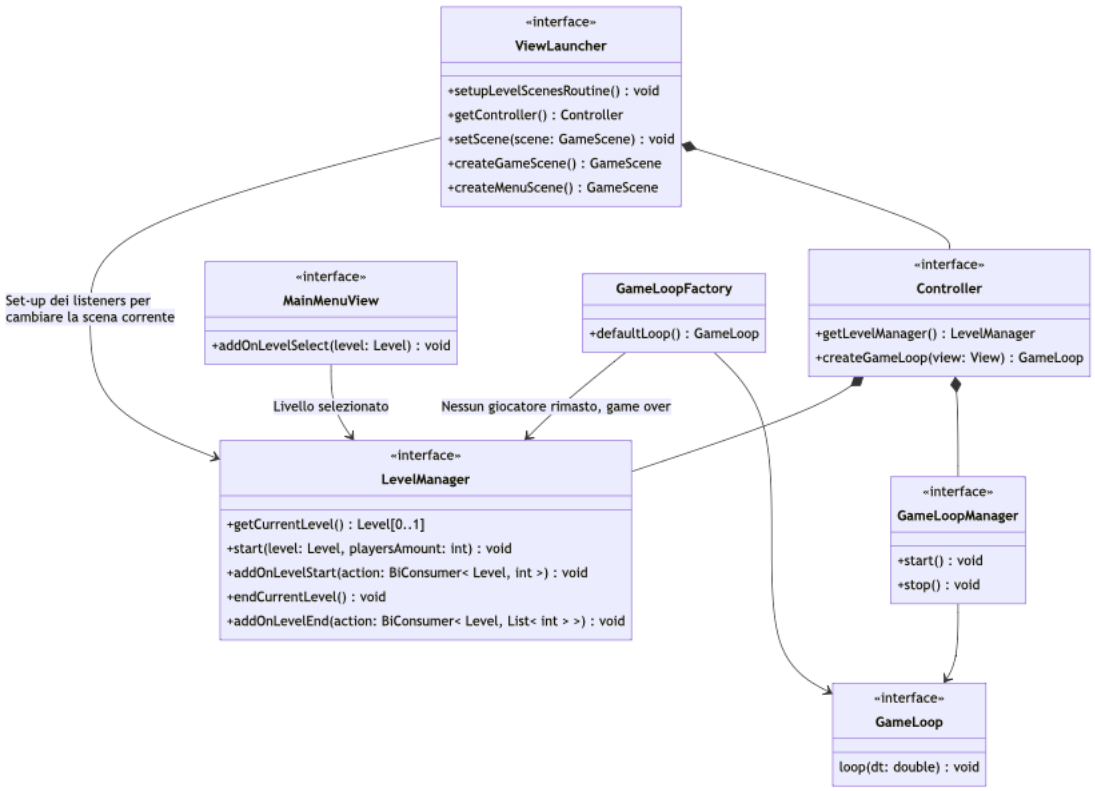

Figura 2.8: Schema UML del lifecycle del livello

Model input

Problema 1 Si vogliono eseguire dei comandi legati al mondo corrente azionati da un input umano da tastiera o mouse, senza però far notare al model l'esistenza delle periferiche di input.

Soluzione Si sono innanzitutto creati dei gruppi per ogni azione che un'entità può compiere: movimento orizzontale, salto, abilità e sprint, e si è creato un supergruppo `CompleteModelInput` che raccoglie tutti i loro comportamenti: in altre parole, un giocatore. A questo punto si possono esporre i giocatori di uno specifico livello tramite questo `CompleteModelInput`, e ad ogni nuovo input 'fisico' ottenuto dalla view, si invoca il suo mapping.

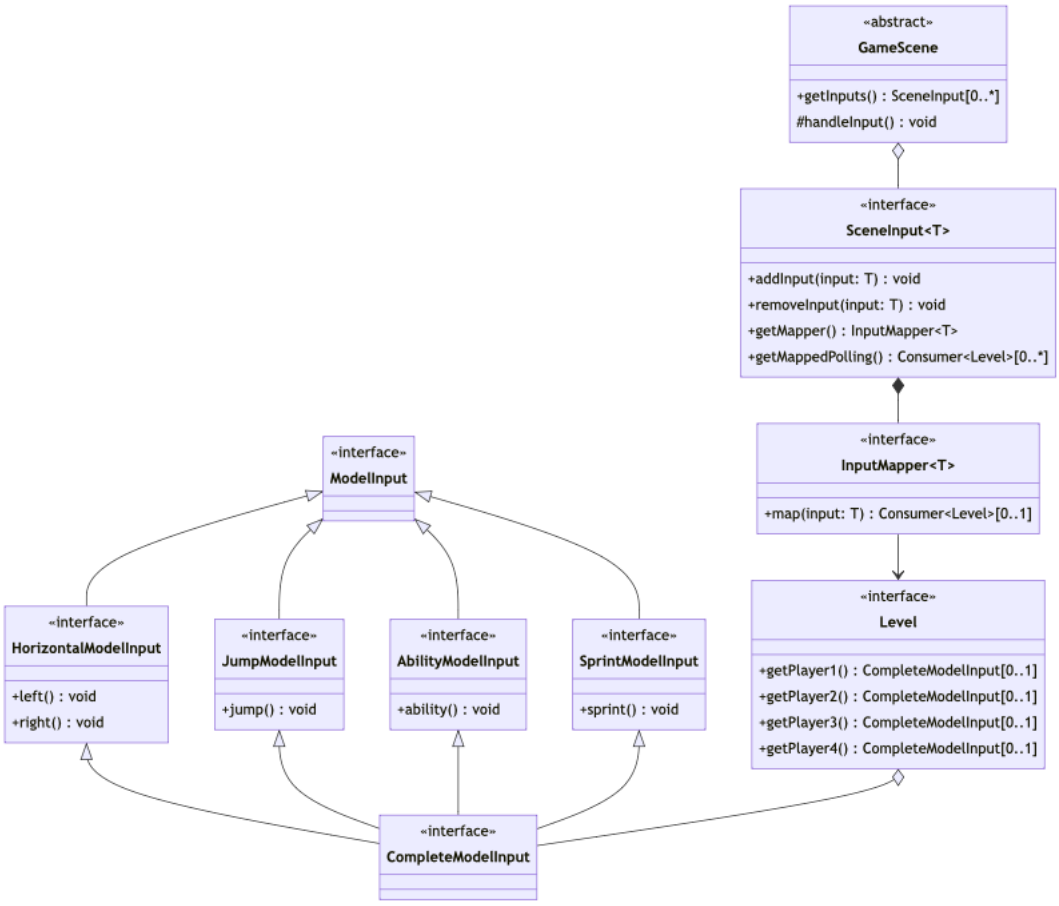

Figura 2.9: Schema UML della gestione dei model input

Problema 2 Risolta l'indipendenza del model della view, un problema che ci siamo posti concerne il concetto di giocatore: e se decidessimo di voler controllare un Goomba, invece che Mario? Si ricordi che le azioni di Goomba comprendono solamente il movimento orizzontale.

Soluzione Si può far credere al livello che un *giocatore* sia in realtà qualsiasi entità utilizzando il pattern *Adapter*: `ModelInputAdapter` conserva un `ModelInput` generale di riferimento ed espone tutti i metodi di `CompleteModelInput`. Ad esempio, nel caso di Goomba, la chiamata a `left()` sull'adapter richiamerà il `left()` del model input interno, mentre una chiamata a `jump()`, non supportato da Goomba, non farà nulla.

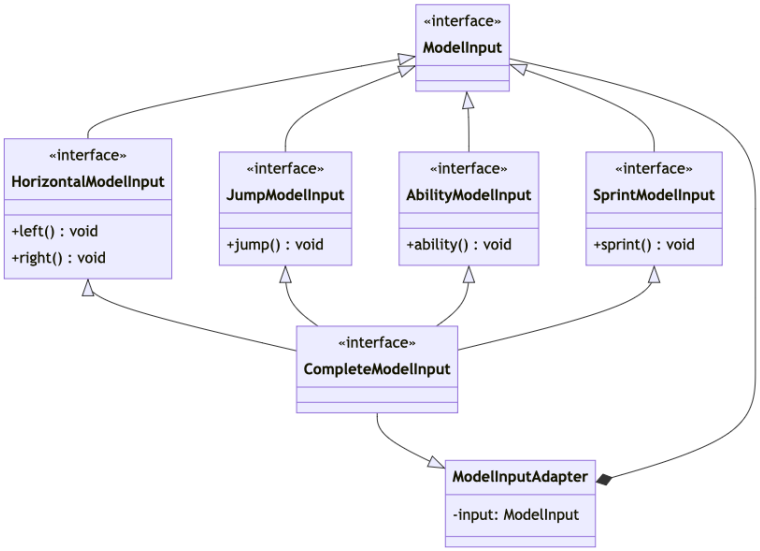

Figura 2.10: Zoom-in sull'utilizzo dell'adapter

Update range del mondo

Problema Un mondo di gioco può contenere anche decine di migliaia di entità, e aggiornare lo stato di tutte comporta un calcolo oneroso. Si vuole quindi limitare l'aggiornamento, ed il rendering, solo alle entità entro una certa distanza dai giocatori (`World#getUserControlledEntities`).

Soluzione Si è creato un record chiamato `WorldOptions` che contiene:

- la massima distanza da uno dei giocatori entro cui un'entità è aggiornata;
- la coordinata Y oltre cui un'entità viene rimossa dal mondo di gioco.

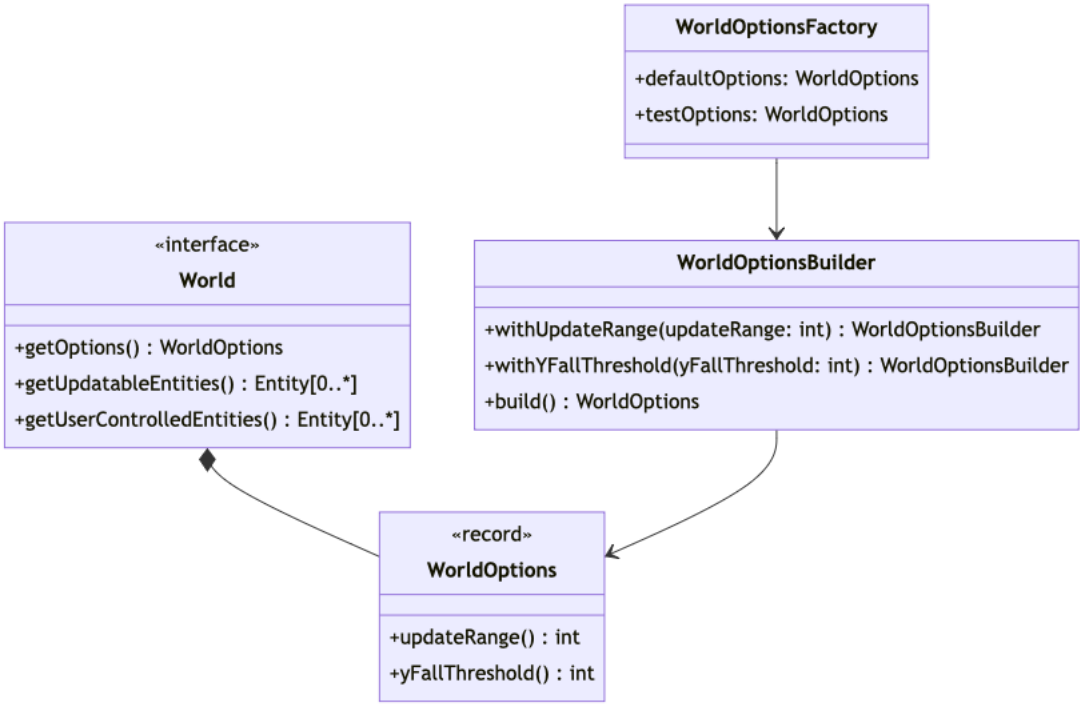

Figura 2.11: Schema UML dell'update range

Tale record può essere creato tramite il suo costruttore, tramite il *builder* `WorldOptionsBuilder` (probabilmente ridondante visto il numero di argomenti, ma si è deciso di tenerlo per possibili opzioni future) o dalla *static factory* `WorldOptionsFactory`.
L'insieme delle entità da aggiornare non sarà più `World#getEntities`, bensì un suo sottoinsieme `World#getUpdatableEntities`.

Gestione della telecamera

Problema Si vuole fornire indipendenza tra le coordinate del mondo e le coordinate - in pixel - dello schermo. Inoltre, si vuole fare in modo che la telecamera segua i giocatori durante la loro avventura e che si rimpicciolisca o ingrandisca in base alle dimensioni della finestra.

Soluzione Ad ogni aggiornamento del game loop si crea un'istanza di `Camera` e la si imposta come telecamera corrente della view.
La telecamera di base, chiamata `SimpleCamera`, effettua operazioni di traslazione e scaling direttamente sul canvas di output e può essere istanziata o tramite il suo costruttore o tramite il *builder* `SimpleCameraBuilder`.

È importante, a mio parere, considerare la facilità con cui si potrebbero aggiungere degli elementi più complessi alla telecamera, come distorsioni dell'immagine o alterazioni dei colori, modificando solo l'implementazione di `Camera`.

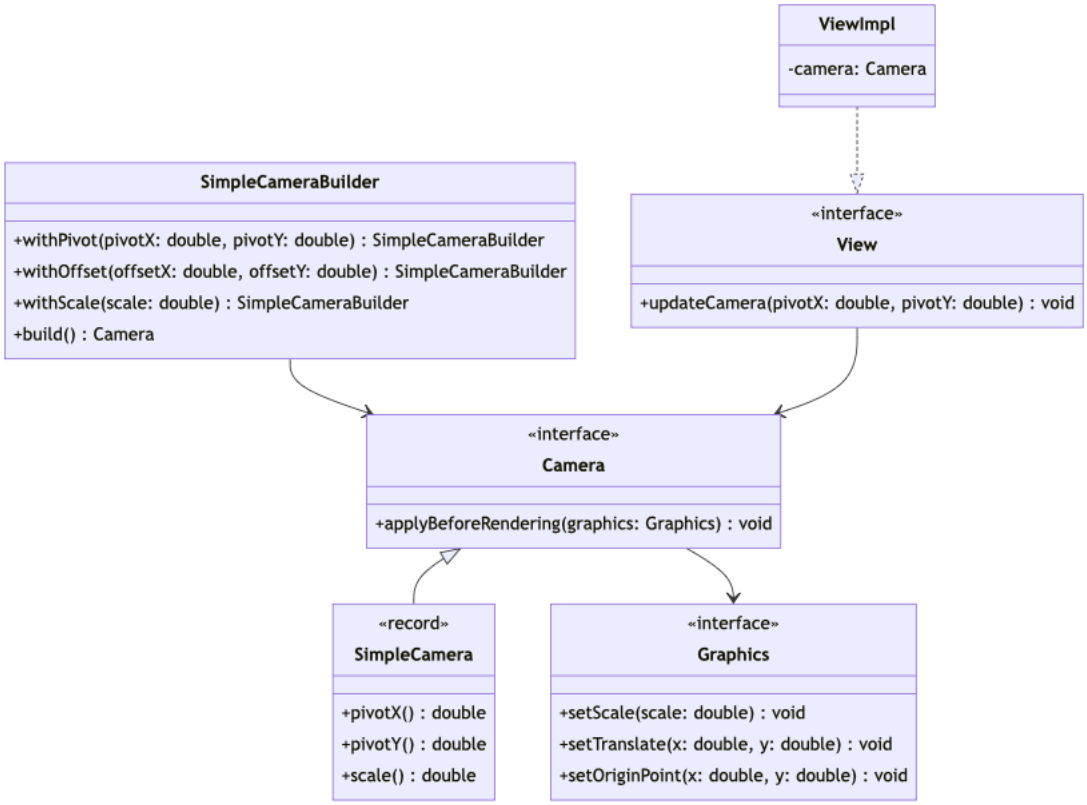

Figura 2.12: Schema UML della struttura della telecamera

2.2.3 Ghignatti Nicolò

Entità statiche

Problema Collisioni delle entità statiche e gestione dei comportamenti di quest'ultime.

Soluzione Attraverso l'utilizzo di un `CollisionComponent`, che ne rileva le collisioni è possibile gestire i comportamenti applicando il pattern `Startegy`, il componente prende una `Collection<Collision>` che riguardano le collisioni subite dall'entità a cui appartiene e ne specifica la strategia da applicare.

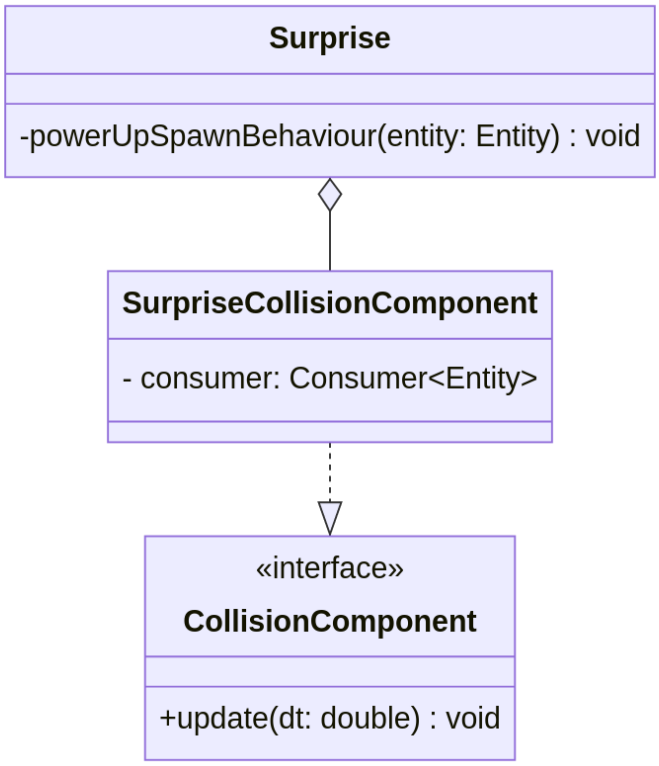

Figura 2.13: Schema UML dell'analisi del problema, con rappresentato un esempio di entità con il corrispettivo collision component

Problema Spawn di entità all'interno del gioco

Soluzione Utilizzo del *pattern Factory* per permettere la rapida creazione di ciascuna entità voluta

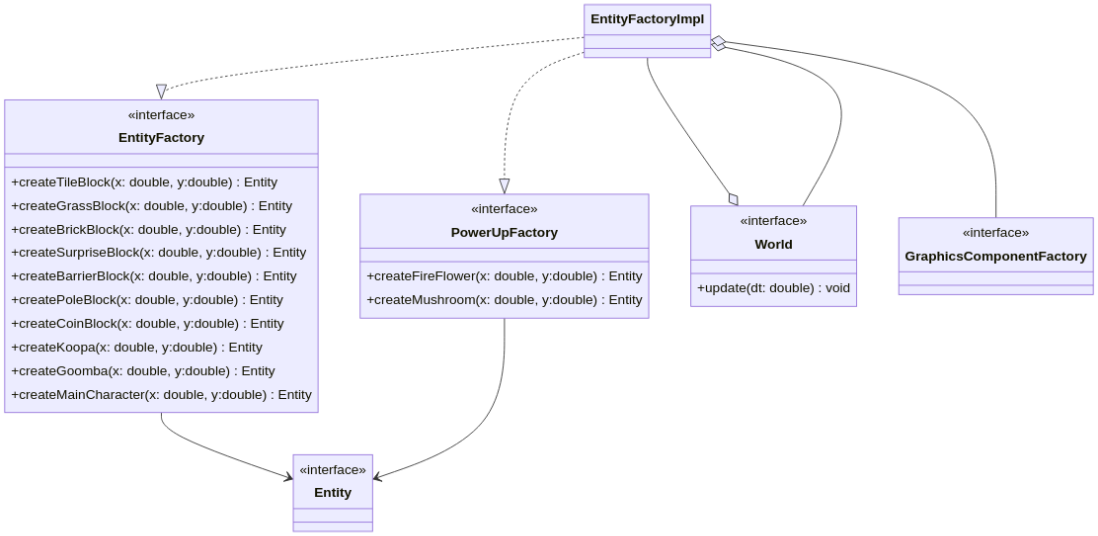

Figura 2.14: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Statistiche e punteggio

Problema Incremento del punteggio con l'uccisione di nemici o la raccolta di monete

Soluzione Il sistema utilizza il *pattern Observer* per applicare un `Conumer<Entity>` ogni qualvolta un entità muore, in questa situazione lo `ScoreManager` si iscrive all'evento morte di un entità passando il consumer da applicare, che verrà applicato dal `EventHandler` nel `World`.

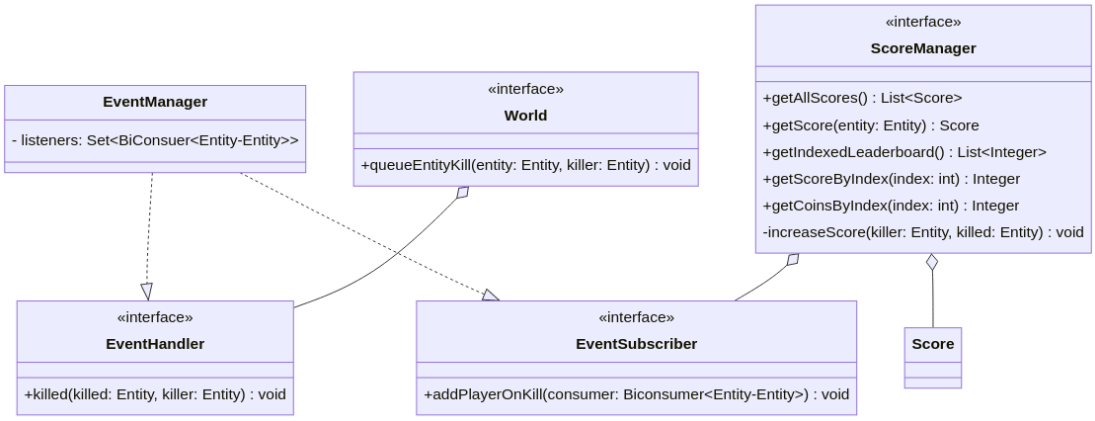

Figura 2.15: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

2.2.4 Luca Patrignani

Gestione degli eventi legati alle collisioni

Problema I nemici hanno reazioni alle collisioni simili fra loro.

Soluzione Creo una interfacccia chiamata `CollisionReactor` che accetta una `Collection<Collision>`. La sua più naturale implementazione è `SingleCollisionReactor`. `SingleCollisionReactor` fa uso di *strategy pattern* per permettere ai suoi utlizzatori di definire a quali collsioni deve reagire (`Predicate<Collision>`) e quale reazione vogliono (`Consumer<Entity>`), come mostrato in 2.16. Inoltre è possibile creare nuovi `CollisionReactor` combinanone altri tramite la static factory `CollisionReactorFactory` 2.17.

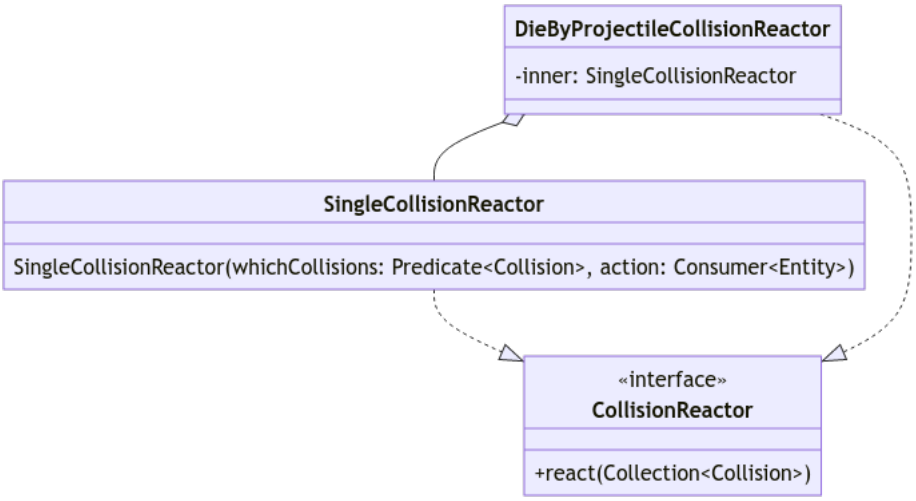

Figura 2.16: Schema UML del pattern strategy per le reazioni alle collisioni

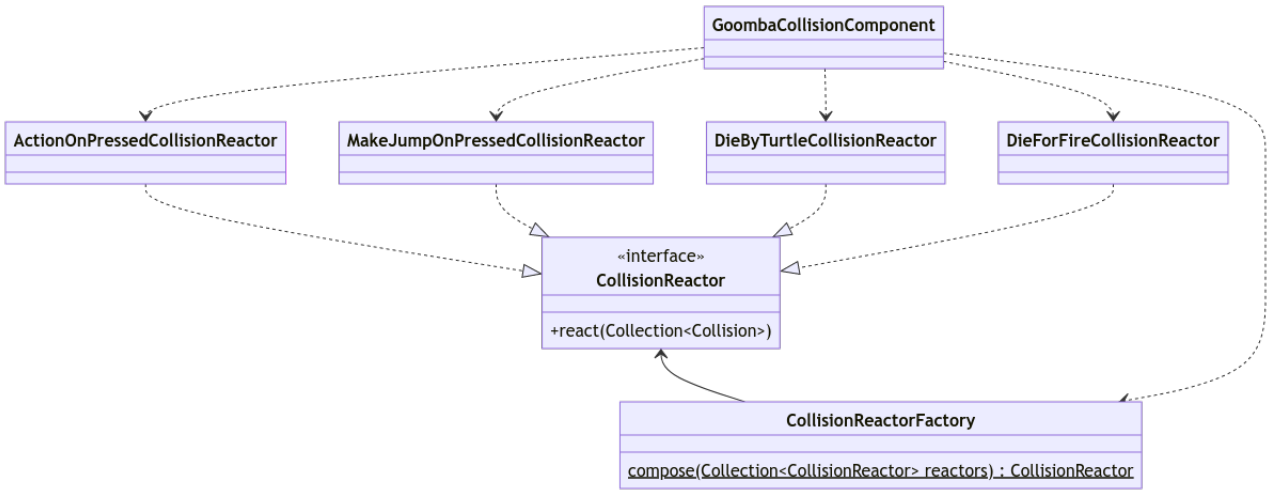

Figura 2.17: Schema UML dell'utilizzo della static factory `CollisionReactorFactory`

Intelligenza dei nemici

Problema I nemici hanno intelligenze simili.

Soluzione Creo una unica intelligenza che permette di controllare i movimenti orizzontali. Passando al costruttore un `Consumer<HorizontalModelInput>` *strategy pattern* posso decidere il comportamento iniziale della IA (in pratica

la direzione iniziale dell'entità controllata). Inoltre è possibile sempre decidere tramite *strategy pattern* a quali collisioni la IA deve reagire passando al costruttore un `Predicate<Collision>`.

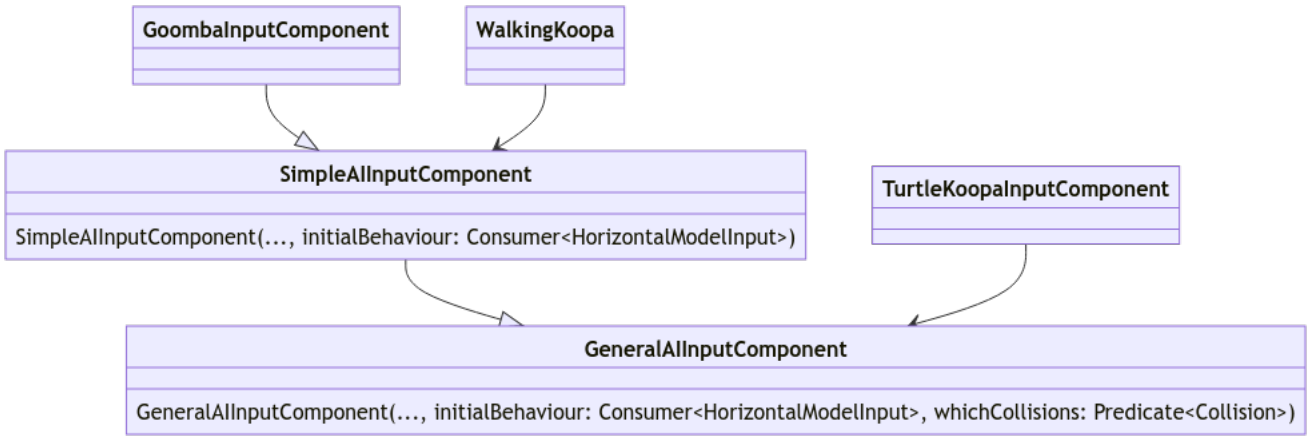

Figura 2.18: Schema UML per mostrare la riutilzzabilità di `CollisionReactorFactory` tramite strategy pattern

Eventi che portano all'uccisione dei nemici

Problema Isolare la parte di logica che gestisce la morte di una entità dalla parte che controlla l'evento che porta alla morte.

Soluzione Tramite *strategy pattern* è possibile passare al `ActionOnPressedCollisionReactor` un `Consumer<Entity>` che accetta l'assassino dell'entità controllata. È possibile anche passare dei azioni diverse dalla morte della entità 2.19, permettendo perciò anche il riutilizzo di questa classe.

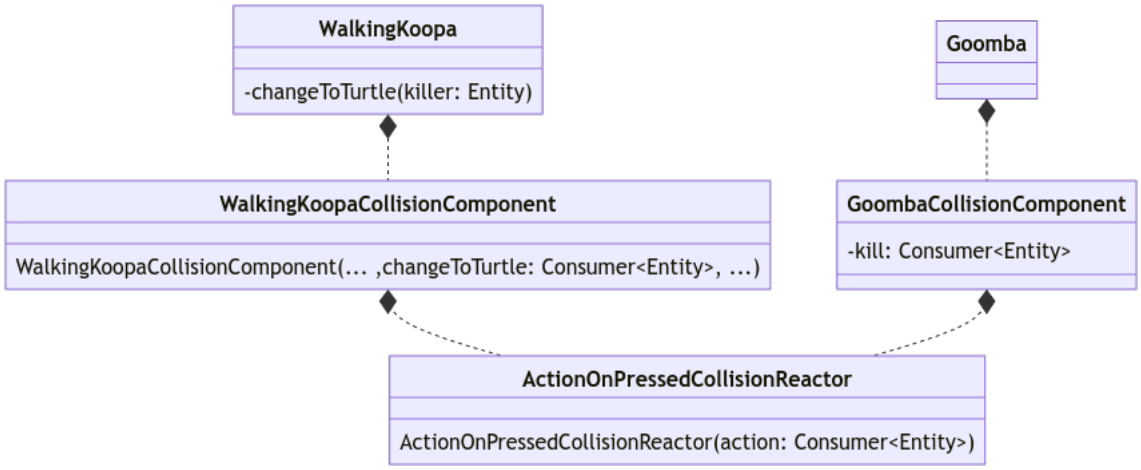

Figura 2.19: Schema UML per mostrare la riutilzzabilità di `ActionOnPressedCollisionReactor`

Koopa e i suoi stati

Problema Il nemico Koopa ha due stati: prima cammina e poi diventa una tartaruga vera e propria.

Soluzione Il primo approccio adottato è stato quello di applicare *state pattern* come in figura 2.20. Ma col progredire del progetto mi sono reso conto che il cambio di stato del Koopa è più riconducibile alla morte di un nemico (ad esempio nella assegnazione dei punti). Ho quindi implementato il cambio di stato uccidendo il `WalkingKoopa` e spawnando un `TurtleKoopa` nella stessa sua posizione 2.21.

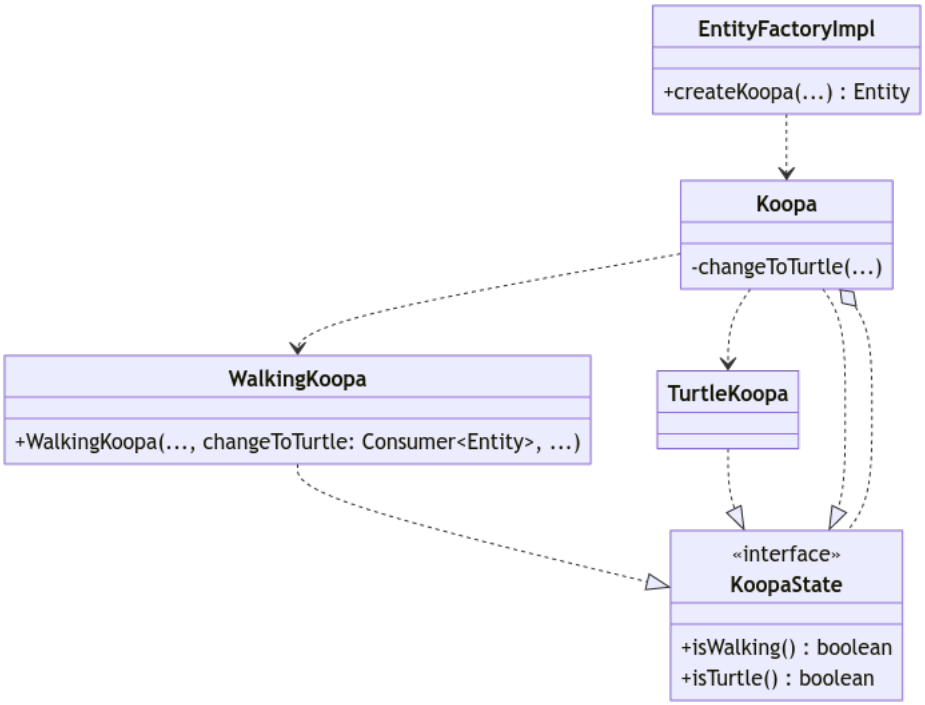

Figura 2.20: Primo approccio poi scartato per la modellare il cambio di stato di Koopa

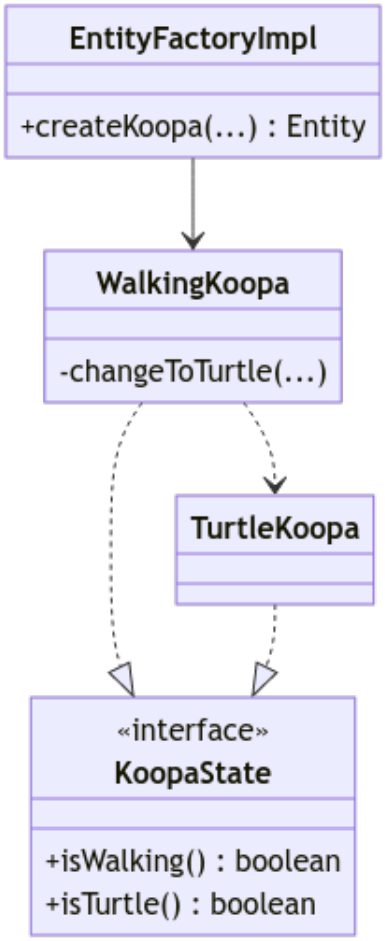

Figura 2.21: Approccio finale per la modellare il cambio di stato di Koopa

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Sono stati testati i comportamenti dell'applicazione sotto diversi punti di vista come:

- I movimenti delle entità, e dei rispettivi comportamenti in situazioni specifiche, come nel caso di collisioni o la loro morte per causa del giocatore o di una caduta;
- I punteggi assegnati al player dopo l'uccisione di entità o la raccolta di monete;
- Il componente fisico creato per le forze in gioco, come la gravità e l'attrito;

Tutti i test sono stati sviluppati attraverso l'utilizzo di JUnit 5, framework di testing per Java.

3.2 Metodologia di lavoro

3.2.1 Antonelli Giacomo

Giocatore principale

Mantenento tra le entità un'architettura coerente, anche la classe del Player è stata sviluppata attraverso l'uso del Component Pattern, i vari componenti sono stati modellati in maniera tale che ci fosse la più totatle indipendenza l'uno dall'altro, nei casi in cui questa non è potuta essere stata rispettata, per necessità, vi è stato utilizzato il Pattern Observer per accedere a funzionalità di altri componenti.

Powerup (livello logico)

Per quanto riguarda i powerup, mi sono occupato della parte logica di essi, quindi come essi vengono gestiti internamente alle entità, per questo è stata definita l'abstract class `AbstractPowerupableEntity` che viene estesa da ogni entità che vuole poter utilizzare i powerup (attualmente nel gioco è stata estesa solo da `Player` ma questo non esclude poterlo fare lo stesso con un `Enemy` o altre `Entity`). Mentre il funzionamento interno dei Powerup è stato spiegato nella sezione precedente, per il collegamento fra Powerup "logico" e Powerup "fisico" è stata realizzata un'interfaccia `PhisicalPowerup`, implementata all'interno dei corrispettivi Powerup fisici tramite cui è possibile ottenerne la versione logica.

3.2.2 Garofalo Giorgio

Engine multi-framework

Ho sviluppato interamente in autonomia l'engine grafico implementato in JavaFX che permette di renderizzare del contenuto e di raccogliere gli input da tastiera.

Deserializzazione dei livelli

Ho creato un deserializzatore dei livelli a partire da file JSON con l'utilizzo della libreria `Gson`. Il deserializzatore comprende anche la possibilità di usare delle *macro* sugli oggetti JSON che rappresentano un'entità: ad esempio, una *fill macro* prende in parametro lunghezza ed altezza così da replicare l'entità in oggetto $w \times h$ volte in una certa area, ed è risultato particolarmente utile per costruire il terreno dei livelli.

NB: i file dei livelli sono copiati all'avvio nella directory {`userhome`}`/.pixformer/levels` solo se il file corrispondente non esiste, per motivi di prestazioni. Dunque non è possibile modificare esternamente un file per poter alterare un livello esistente, ma lo si può copiare e modificare sotto un altro nome.

Lifecycle del livello

Ho sviluppato da solo il sistema di inizio-fine dei livelli tramite il `LevelManager`, ed il game loop (`GameLoopFactory.defaultLoop`).

Gestione delle collisioni

Ho sviluppato autonomamente il sistema di collisioni basato su `CollisionManager` e `CollisionComponent`, e la sua specializzazione `SolidCollisionComponent`.

Gestione della telecamera

Ho implementato da solo lo sliding sui giocatori e lo scaling della telecamera. Il punto di pivot della telecamera è dato dal punto medio delle coordinate dei giocatori, ed è fornito da `Controller#calcEntitiesCommonPointX/Y`. Il fattore di scale è ricavato in base alla lunghezza della finestra.

Controller

Ho costruito buona parte dell'interfaccia `Controller` e la corrispondente implementazione `ControllerImpl`, e l'ho integrata come ponte tra model e view.

Menu principale

Ho progettato, disegnato e sviluppato il menu principale rappresentato da `MainMenuView` e implementato in JavaFX da `PixformerJavaFXMainMenuScene`, con vari componenti come selettore del livello, selettore del numero di giocatori e scoreboard post-partita.

Parti in collaborazione

Ho gettato la base del `World` come contenitore di entità. Ho inoltre contribuito alla risoluzione di vari bug, tra cui uno per cui il giocatore non riusciva a raccogliere dei power-up.
La versione finale della gestione dei model input è stata implementata da me, ma collaborando con Luca per arrivare alla migliore soluzione.

3.2.3 Ghignatti Nicolò

Entità statiche

Come dichiarato nella richiesta di progetto mi sono occupato delle entità statiche nella loro più completa gestione, partendo dal model dove ne specifico i comportamenti quando avviene una collisione, fino alla view coi componenti grafici. La loro creazione avviene attraverso l'utilizzo di una factory per tutte le entità in gioco, suddivisa logicamente in `EntityFactory` e

`PowerUpFactory`, la quale finale implementazione è stata svolta in cooperazione.

Statistiche

Inoltre della gestione delle statistiche in gioco, attraverso il collezionamento di punti con l'uccisione dei nemici, la raccolta di monete o di power-up in più, assieme alla creazione della possibile leaderboard del livello (package `pixformer.model.score` e `pixformer.model.event`).

Fisica

Per quanto riguarda la fisica ho sviluppato un componente (`PhysicComponent`) a libero utilizzo delle entità per applicare la gravità ad esse, mentre l'attrito invece è stato applicato e gestito nei collision component, venendo applicato quando un'entità "cammina" su un'entità solida, come possono essere i blocchi d'erba nel gioco.

Frame sync e pausa

Altra parte da me sviluppata è stata la sincronizzazione per il framerate del gioco, assieme alla pausa all'interno di tutto il progetto, dal mapping dell'input (solo per determinati tasti, che ho definito tasti di controllo) allo stop dell'update del mondo. Inoltre anche tutto quanto riguarda il tempo è stato svolto in collaborazione (package `pixformer.common.time`)

Parti in collaborazione

Ho contribuito anche alla creazione dei componenti grafici di alcune Entità all'interno della `GraphicsComponentFactory` e nella implementazione di alcune di esse, a piccole parti del `Controller` e del `World`

3.2.4 Luca Patrignani

Entità dinamiche

Il mio lavoro si è principalmente focalizzato nella creazione degli `InputComponent` e `CollisionComponent` delle entità che si muovono autonomamente, ad esempio i nemici (`pixformer.model.entity.dynamic.enemy`). All'interno del progetto abbiamo chiamato queste entità *dinamiche*.

Potenziamenti di gioco come entità

Mi sono inoltre occupato dei `InputComponent` e `CollisionComponent` dei potenziamenti di gioco per quanto riguarda la loro presenza nel mondo, cioè da quando spawnano a quando vengono raccolti dal giocatore (`pixformer.model.entity.dynamic.powerup`). Per evitare incomprensioni preciso che *non* mi sono occupato interamente dei potenziamenti di gioco (i powerups), ma solo delle loro entità prima di essere raccolti dal giocatore. In pratica le mie entità implementano l'interfaccia `PhysicalPowerup`.

CollisionReactor

Per riutilizzare al meglio il codice ho sviluppato l'interfaccia `CollisionReactor` e alcune sue implementazioni (`pixformer.model.entity.dynamic.reactor`).

3.2.5 Lavoro collettivo

Per quanto riguarda l'uso del DVCS abbiamo optato per un modello più semplice rispetto a git-flow: abbiamo un branch principale (`master`) che da cui di diramano dei branch indipendenti, ognuno per lo sviluppo di una feature. A feature pronta viene effettuato il merge del branch su `master`. Abbiamo tenuto i nostri feature branch il più sincronizzati possibili con il branch `master` per evitare conflitti di merge complessi da risolvere. Il maintainer della repository remota è Garofalo, tutti i componenti del gruppo lavorano su copie locali della repository.

3.3 Note di sviluppo

3.3.1 Antonelli Giacomo

Utilizzo di Record

`https://github.com/iamgio/OOP22-pixformer/blob/d28d73f2a200568d8cccfc752a8611e3f`
`src/main/java/pixformer/view/entity/player/PlayerState.java`

Utilizzo di Stream

`https://github.com/iamgio/OOP22-pixformer/blob/d28d73f2a200568d8cccfc752a8611e3f`
`src/main/java/pixformer/view/entity/player/PlayerAnimationImpl.java#`
`L73`

Utilizzo di Optional

https://github.com/iamgio/OOP22-pixformer/blob/d28d73f2a200568d8cccfc752a8611e3f
src/main/java/pixformer/model/entity/powerup/PowerUp.java#L44

3.3.2 Garofalo Giorgio

Utilizzo di JavaFX

Utilizzata per la rappresentazione grafica del mondo di gioco e del menu iniziale:
https://github.com/iamgio/OOP22-pixformer/tree/b91d3babaf2ebc30d4497eb156dd4c1c2
src/main/java/pixformer/view/engine/javafx

Utilizzo della libreria Gson

Utilizzata per caricare i livelli da file JSON:
https://github.com/iamgio/OOP22-pixformer/blob/b91d3babaf2ebc30d4497eb156dd4c1c2
src/main/java/pixformer/controller/deserialization/level/JsonLevelDataDeserializ
java#L39-L105

Utilizzo di ResourceBundle

Utilizzato per l'internazionalizzazione dell'interfaccia grafica:
https://github.com/iamgio/OOP22-pixformer/blob/b91d3babaf2ebc30d4497eb156dd4c1c2
src/main/java/pixformer/view/engine/internationalization/Lang.java#
L40-L53

Creazione di una nuova annotazione

Utilizzata per ricavare le entità a partire da un ID nella deserializzazione dei livelli:
https://github.com/iamgio/OOP22-pixformer/blob/b91d3babaf2ebc30d4497eb156dd4c1c2
src/main/java/pixformer/controller/deserialization/level/EntityType.
java#L14
Utilizzo:
https://github.com/iamgio/OOP22-pixformer/blob/b91d3babaf2ebc30d4497eb156dd4c1c2
src/main/java/pixformer/model/entity/EntityFactoryImpl.java#L47

Utilizzo di reflection

Utilizzata per creare nuove entità da una `EntityFactory` nella deserializzazione dei livelli, con l'aiuto dell'annotazione descritta nel punto precedente:

https://github.com/iamgio/OOP22-pixformer/blob/b91d3babaf2ebc30d4497eb156dd4c1c2
src/main/java/pixformer/controller/deserialization/level/EntityFactoryLookupDeco
java#L37-L64

Realizzazione di interfacce generiche

https://github.com/iamgio/OOP22-pixformer/blob/b91d3babaf2ebc30d4497eb156dd4c1c2
src/main/java/pixformer/view/engine/InputMapper.java#L14
https://github.com/iamgio/OOP22-pixformer/blob/b91d3babaf2ebc30d4497eb156dd4c1c2
src/main/java/pixformer/view/engine/SceneInput.java#L16 https://
github.com/iamgio/OOP22-pixformer/blob/b91d3babaf2ebc30d4497eb156dd4c1c28608c03/
src/main/java/pixformer/common/wrap/ObservableWritableWrapper.java#
L11

Utilizzo di Record

https://github.com/iamgio/OOP22-pixformer/blob/8978269bf040af88861026c0b9a054054
src/main/java/pixformer/common/Vector2D.java#L9
https://github.com/iamgio/OOP22-pixformer/blob/b91d3babaf2ebc30d4497eb156dd4c1c2
src/main/java/pixformer/model/LevelData.java#L16
https://github.com/iamgio/OOP22-pixformer/blob/b91d3babaf2ebc30d4497eb156dd4c1c2
src/main/java/pixformer/model/entity/collision/Collision.java#L11
https://github.com/iamgio/OOP22-pixformer/blob/b91d3babaf2ebc30d4497eb156dd4c1c2
src/main/java/pixformer/model/WorldOptions.java#L9
https://github.com/iamgio/OOP22-pixformer/blob/b91d3babaf2ebc30d4497eb156dd4c1c2
src/main/java/pixformer/view/engine/camera/SimpleCamera.java#L12

Utilizzo di Stream e lambda expressions

Utilizzate spesso, ad esempio:
https://github.com/iamgio/OOP22-pixformer/blob/b91d3babaf2ebc30d4497eb156dd4c1c2
src/main/java/pixformer/view/engine/SceneInput.java#L46-L50
https://github.com/iamgio/OOP22-pixformer/blob/b91d3babaf2ebc30d4497eb156dd4c1c2
src/main/java/pixformer/model/entity/collision/EntityCollisionManagerImpl.
java#L41-L46

Utilizzo di Optional

Utilizzato spesso, ad esempio:
https://github.com/iamgio/OOP22-pixformer/blob/b91d3babaf2ebc30d4497eb156dd4c1c2
src/main/java/pixformer/controller/level/LevelManagerImpl.java#L26-L28

https://github.com/iamgio/OOP22-pixformer/blob/b91d3babaf2ebc30d4497eb156dd4c1c2
src/main/java/pixformer/model/LevelImpl.java#L69-L71

Codice di terze parti

L'individuazione del lato in cui una collisione è avvenuta (https://github.
com/iamgio/OOP22-pixformer/blob/b91d3babaf2ebc30d4497eb156dd4c1c28608c03/
src/main/java/pixformer/model/entity/collision/RectangleBoundingBox.
java#L30-L43) è stato riadattato da https://stackoverflow.com/a/29861691.

3.3.3 Ghignatti Nicolò

Utilizzo dei Record

Utilizzati nella classe Score:
https://github.com/iamgio/OOP22-pixformer/blob/3dc6788d9401c4b2c15e58554971f5ee4
src/main/java/pixformer/model/score/Score.java#L9

Utilizzo di Stream e lambda expression

Usate spesso, i seguenti ne sono esempi:
https://github.com/iamgio/OOP22-pixformer/blob/3dc6788d9401c4b2c15e58554971f5ee4
src/main/java/pixformer/model/WorldImpl.java#L157
https://github.com/iamgio/OOP22-pixformer/blob/3dc6788d9401c4b2c15e58554971f5ee4
src/main/java/pixformer/model/entity/statics/surprise/SurpriseCollisionComponent
java#L35

Utilizzo della reflection

Usata spesso, ad esempio:
https://github.com/iamgio/OOP22-pixformer/blob/bf27713af15e5b3a847caeb0d31490689
src/main/java/pixformer/model/entity/statics/pole/PoleCollisionComponent.
java#L31

Utilizzo degli Optional

Usati spesso ad esempio:
https://github.com/iamgio/OOP22-pixformer/blob/5203a286c5e329999e970ff462f20eec1
src/main/java/pixformer/view/javafx/PixformerJavaFXKeyboardInputMapper.
java#L56

3.3.4 Luca Patrignani

Utilizzo di Stream

https://github.com/iamgio/OOP22-pixformer/blob/fafbd682460e18acce92b95e6a3a068a0
src/main/java/pixformer/model/entity/dynamic/reactor/CollisionReactorFactory.
java#L35
https://github.com/iamgio/OOP22-pixformer/blob/fafbd682460e18acce92b95e6a3a068a0
src/main/java/pixformer/model/entity/dynamic/reactor/SingleCollisionReactor.
java#L35-L38
https://github.com/iamgio/OOP22-pixformer/blob/99105f22dba28ddffdccd438a88b415c5
src/test/java/pixformer/OnPressedTest.java#L70-L73

Uso di lambda expressions

Usate spesso, alcuni esempi di seguito:
https://github.com/iamgio/OOP22-pixformer/blob/99105f22dba28ddffdccd438a88b415c5
src/main/java/pixformer/model/entity/dynamic/enemy/koopa/walking/
WalkingKoopa.java#L49
https://github.com/iamgio/OOP22-pixformer/blob/99105f22dba28ddffdccd438a88b415c5
src/main/java/pixformer/model/entity/dynamic/reactor/CollisionReactorFactory.
java#L12
https://github.com/iamgio/OOP22-pixformer/blob/99105f22dba28ddffdccd438a88b415c5
src/main/java/pixformer/model/entity/dynamic/reactor/CollisionReactorFactory.
java#L22-L25
https://github.com/iamgio/OOP22-pixformer/blob/99105f22dba28ddffdccd438a88b415c5
src/main/java/pixformer/model/entity/EntityFactoryImpl.java#L132
https://github.com/iamgio/OOP22-pixformer/blob/99105f22dba28ddffdccd438a88b415c5
src/main/java/pixformer/model/entity/EntityFactoryImpl.java#L40-L41
https://github.com/iamgio/OOP22-pixformer/blob/99105f22dba28ddffdccd438a88b415c5
src/test/java/pixformer/OnPressedTest.java#L52

Parti di codice prese da altri progetti

L'interfaccia funzionale `TriConsumer` (`https://github.com/iamgio/OOP22-pixformer/blob/99105f22dba28ddffdccd438a88b415c59ba0431/src/main/java/pixformer/common/TriConsumer.java`) è stata creata partendo dall'implementazione di `BiConsumer` di OpenJDK (`https://github.com/openjdk/jdk/blob/dc81603cbf223c3ac6b413src/java.base/share/classes/java/util/function/BiConsumer.java`).

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Antonelli Giacomo

Personalmente riflettendo sulle fasi iniziali del progetto, pensavo di avere parti più corpose, mi sarebbe piaciuto sviluppare un po' di più, ma oltre alla quantità di cose è fondamentale la qualità del codice, ed è una qualità che ho imparato a sviluppare soprattutto grazie ai ragazzi del gruppo che mi hanno aiutato a vedere finezze che ignoravo. Adesso, con una visione più ampia e chiara di Java e dei pattern di programmazione, avrei sviluppato alcune cose in modo differente, tuttavia sono abbastanza soddisfatto del mio lavoro e penso che questo bagaglio di esperienza mi sarà sicuramente molto utile in futuro.

4.1.2 Garofalo Giorgio

Sono molto soddisfatto dell'engine grafico multi-framework che ho sviluppato e penso che lo potrei integrare in progetti futuri. Non sono invece entusiasta di alcune parti del model e credo potrebbero essere state progettate in modo più esaustivo. In generale sono soddisfatto del risultato, anche se ci sono stati ripetuti problemi di comunicazione tra di noi che mi hanno portato a lavorare poco più del previsto in modo da finire il progetto in tempo.

4.1.3 Ghignatti Nicolò

Penso che il codice che ho scritto sia semplice da capire e facilmente utilizzabile, personalemente non mi piace l'utilizzo, a mio parere un po' eccessivo, del metodo `instanceof` causato dal continuo passaggio di `Entity` ai metodi

all'interno del progetto. Personalmente, nonostante il carico lavorativo sia stato rispettato, avrei voluto poter fare di più.
Inoltre penso che non si sia attribuita abbastanza importanza alla parte di analisi e alla decisione dell'architettura all'inizio del progetto, che è stata un po' diversa dalle esperienze avute in precedenza.

4.1.4 Luca Patrignani

Valuto il mio lavoro in questo progetto sufficiente. Credo che aver lavorato per la prima volta in un progetto di discrete dimensioni mi abbia fatto capire molte cose su come si programma e alla luce di ciò avrei improntato tutto il mio lavoro differentemente. Alla fine di questo percorso mi sento di dire che l'analisi del dominio ha alcune lacune. Per quanto riguarda il dominio ad esempio ho riscontrato diversi problemi (anche durante lo sviluppo) riguardanti l'uso del costrutto `instanceof` per riconscere il tipo di entità.
Non credo che se volessi creare di nuovo un platformer userei come base questo progetto ma ricomincerei dalle fondamenta.

Appendice A

Guida utente

All'avvio del programma apparirà il menu principale, da cui è possibile impostare il numero di giocatori usando i bottoni + e −. Cliccare su un bottone avvia il livello corrispondente al file {`userhome`}`/.pixformer/levels/`{`nomelivello`}`.json`.

Comandi generali

Tasto	**Comando**
P	Pausa
O	Continua
ESC	Ritorno al menu

Comandi dei giocatori

	Salto	**Destra**	**Sinistra**	**Abilità**	**Sprint**
Player 1	W/SPACE	D	A	Shift	Ctrl
Player 2	Up arrow	Right arrow	Left arrow	M	N
Player 3	Y	J	G	C	X
Player 4	Numpad 8	Numpad 6	Numpad 4	Numpad 2	Numpad 1

Appendice B

Esercitazioni di laboratorio

B.1 luca.patrignani3@studio.unibo.it

- Laboratorio 3: `https://virtuale.unibo.it/mod/forum/discuss.php?d=112846#p168303`
- Laboratorio 5: `https://virtuale.unibo.it/mod/forum/discuss.php?d=114647#p170059`

Appendice C

Crediti a terzi

Gli sprites delle entità non sono protetti da copyright e sono stati presi da: `https://webfussel.itch.io/more-bit-8-bit-mario`.