Programmazione ad oggetti

Relazione di progetto: Pyxis

Realizzato da :
Giovanni Antonioni 0000922658
Luca Rubboli 0000923420
Riccardo Traini 0000924216

Indice

Capitolo 1

Analisi

Il gruppo si pone come obbiettivo lo sviluppo di una replica del celebre videogioco Arkanoid.
Arkanoid è una tipologia di videogame a giocatore singolo, suddiviso in più livelli, che presenta, in ciascuno di questi, un'arena di gioco. Di quest'ultima possiamo distinguere nella sua parte superiore una serie di mattoncini di varia tipologia e, in quella inferiore, una piattaforma con una palla. La palla si muove all'interno di tutta l'arena, infliggendo, in caso di collisione, dei danni ai mattoncini, che dopo un determinato numero di colpi ricevuti, vengono distrutti. La condizione necessaria per superare un livello è quella di distruggere ciascun mattoncino distruttibile presente all'interno dell'Arena. La piattaforma, controllata dal giocatore, si muove orizzontalmente su una linea immaginaria posta al di sopra del fondo dell'area di gioco, e la sua posizione iniziale è al centro di questa linea. La piattaforma agisce da racchetta, l'impatto con la palla scaturisce un ridirezionamento di quest'ultima. Se la palla raggiunge il fondo dell'area di gioco, passando quindi la piattaforma, si perde una vita. All'inizio di ogni livello vengono concesse 3 vite e in caso di perdita di ognuna di queste, la partita si riterrà conclusa.

1.1 Requisiti

Requisiti funzionali

- Il giocatore può controllare il pad muovendolo nei limiti dell'arena di gioco.
- L'applicazione implementa la fisica della palla e ne gestisce il rimbalzo con i bordi dell'arena, con il pad e con i mattoncini.

- L'applicazione permette di generare, in maniera casuale alla rottura di un mattoncino, dei powerup e di applicarne gli effetti una volta che questi collidono con il pad.
- L'applicazione implementa il caricamento dei livelli di gioco da files di configurazione.
- Deve essere possibile selezionare un livello di gioco.
- A fine partita il giocatore dev'essere in grado di visualizzare il punteggio totale ottenuto.
- Il gioco deve gestire la perdita delle vite.

Requisiti non funzionali

- L'applicazione deve garantire un caricamento efficiente dei files media.
- L'applicazione deve essere in grado di garantire delle buone prestazioni durante la fase di rendering grafico.

1.2 Analisi e modello del dominio

Il gioco è composto da più entità "Livello" aventi un ordine crescente di difficoltà. Tali livelli contengono un'arena all'interno della quale vengono caricati determinati elementi: Mattoncino (brick), Palla (ball) e Piattaforma (pad). Un livello contiene anche informazioni riguardanti il punteggio accumulato e le vite possedute dal giocatore, inoltre permette la sincronizzazione degli elementi contenuti al suo interno e ne aggiorna stato e posizione (se modificabile). Le vite costituiscono il numero di tentativi concessi all'utente per completare un livello. Al termine di esse la partita si interrompe e il giocatore potrà scegliere se avviarne una nuova a partire dal primo livello, oppure selezionarne uno tra quelli già sbloccati.
L'elemento **mattoncino** si può presentare in due forme differenti:

- **Distruttibile** se, dopo una o più collisioni con la palla, viene rimosso dall'area di gioco.
- **Indistruttibile** qualora il mattoncino persista all'interno dello scenario indifferentemente dal numero di collisioni con la palla che questo riceve. La sua distruzione può avvenire come risposta alla sola collisione con la palla atomica.

Un **livello** è composto da più mattoncini di vario tipo disposti secondo un preciso schema nella parte superiore dell'arena. Quando ogni mattoncino del tipo "distruttibile" viene rimosso il livello può essere considerato completato e si può procedere a quello successivo.
La **piattaforma** è un elemento controllato dall'utente, questa risiede nella parte inferiore dell'arena ed effettua degli spostamenti orizzontali. Il suo compito è sia quello di direzionare la palla per la distruzione dei mattoncini, sia quello di non farle raggiungere il limite inferiore dell'arena, pena la decurtazione di una vita.
La **palla** è l'elemento che si occupa della distruzione dei mattoncini. Ha una velocità iniziale casuale e rimbalza all'interno dell'arena collidendo con i suoi elementi. Il rimbalzo con la piattaforma è scandito da regole differenti rispetto al classico rimbalzo su un qualsiasi altro elemento, la direzione della palla viene infatti modificata diversamente a seconda del rapporto tra le coordinate orizzontali del punto di collisione, calcolato sulla piattaforma, e la lunghezza stessa di quest'ultima, permettendo al giocatore un controllo migliore della dinamica della palla.
I **power-up** sono entità che modificano la palla e la piattaforma di gioco. Vengono generati in maniera casuale alla distruzione di un mattoncino e si muovono verso il basso mantenendo una velocità costante. Possono essere acquisiti dal giocatore attraverso la collisione con la piattaforma, altrimenti, al raggiungimento del limite inferiore dell'area di gioco, vengono rimossi. Tutti i powerup che influiscono sulla piattaforma possono sovrapporsi applicando in contemporanea il proprio effetto nei limiti consentiti. Caso contrario invece per quanto riguarda le tipologie di powerup che modificano il comportamento della palla, questi non possono essere sovrapponibili e l'applicazione dei loro effetti comporta l'interruzione degli altri.

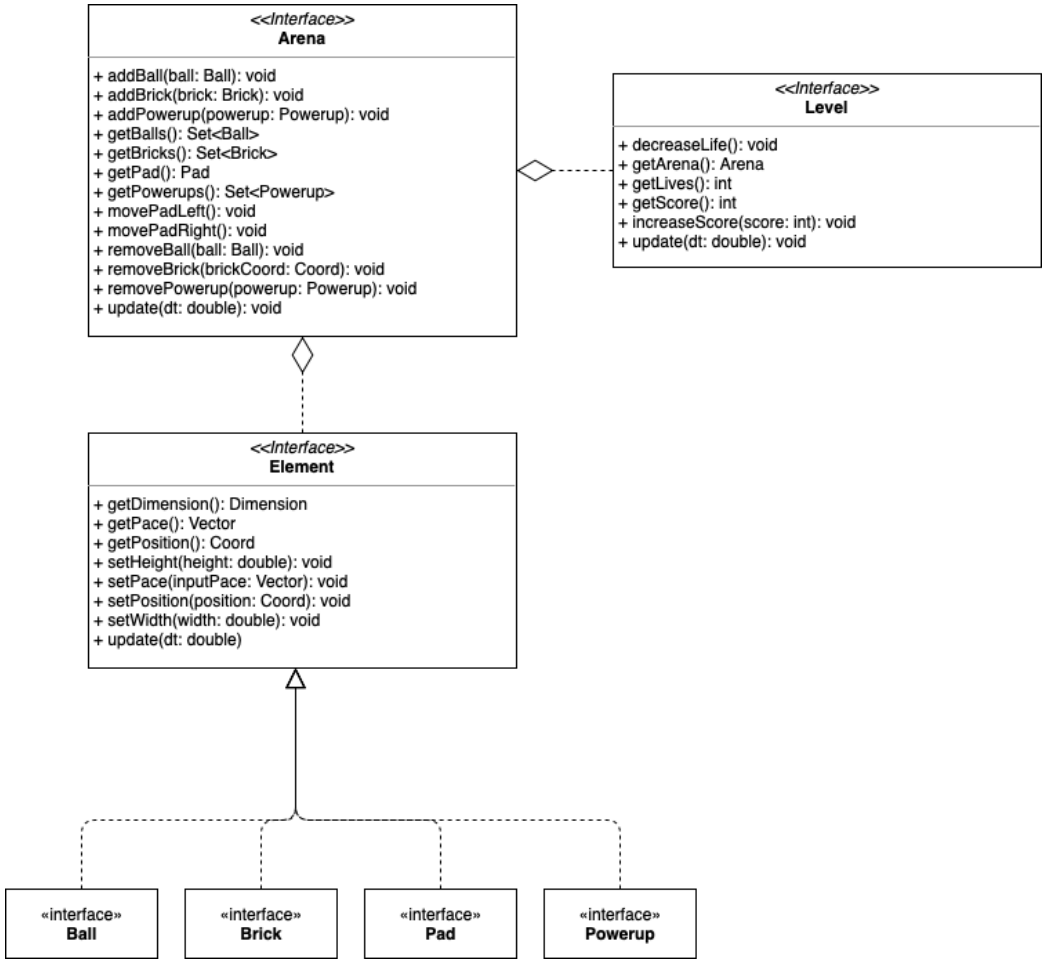

Figura 1.1: Schema UML di partenza

Capitolo 2

Design

2.1 Architettura

Per lo sviluppo dell'applicativo abbiamo deciso di adoperare il pattern **MVC** (Model-View-Controller). Questo ci ha permesso di dividere quella che è la logica del dominio dell'applicazione dalla sua effettiva rappresentazione grafica, garantendo in questo modo una maggiore portabilità ed estendibilità. Ci siamo posti come obbiettivo quello di marcare una linea di separazione netta tra le varie parti, permettendo in questo modo una sostituzione più agevole di librerie grafiche, che non devono andare ad intaccare in alcun modo le sezioni di model e controller.

2.1.1 Model

Il model rappresenta la parte dell'applicazione che gestisce le logiche relative al suo dominio, fornendo i metodi necessari all'accesso e alla modifica dei dati e garantendone una totale coerenza in ogni suo stato. Questo viene adoperato dal `Controller` sia per mantenere aggiornato lo stato del gioco ed ottenere quindi le eventuali informazioni richieste, sia per applicare delle modifiche in base all'input utente registrato dalla sezione view ed elaborato dal `Controller` stesso.

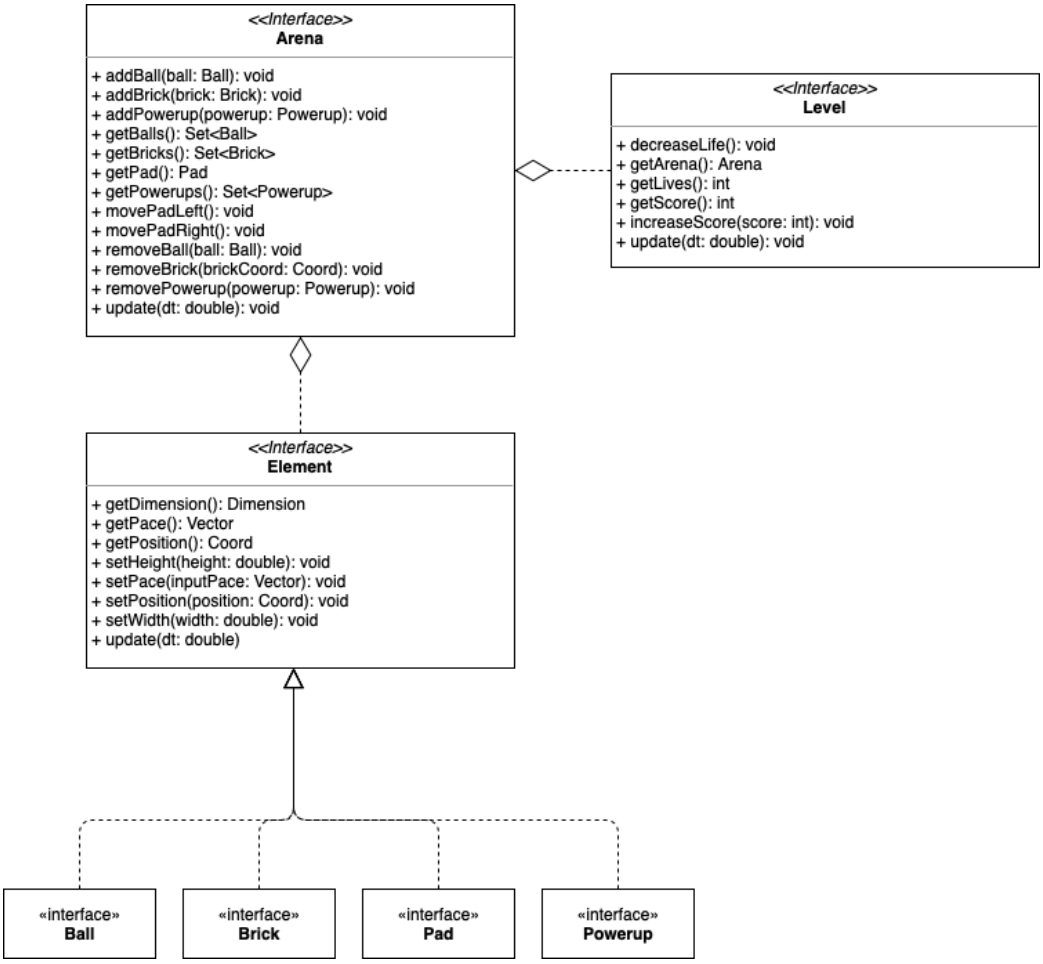

Figura 2.1: Model dell'applicazione

2.1.2 View

La view rappresenta l'interfaccia di gioco che l'applicativo offre all'utente.
Il gioco si sviluppa su diverse fasi, pertanto abbiamo reputato opportuno implementare differenti viste relative ad ognuna di esse. Per ogni vista è stato introdotto un `Controller` personalizzato, che elargisce ad essa tutte le informazioni necessarie per garantire una corretta visione di gioco.
La view si occupa inoltre di gestire l'interazione con l'utente, informando il `Controller` ad essa associato in caso di input.

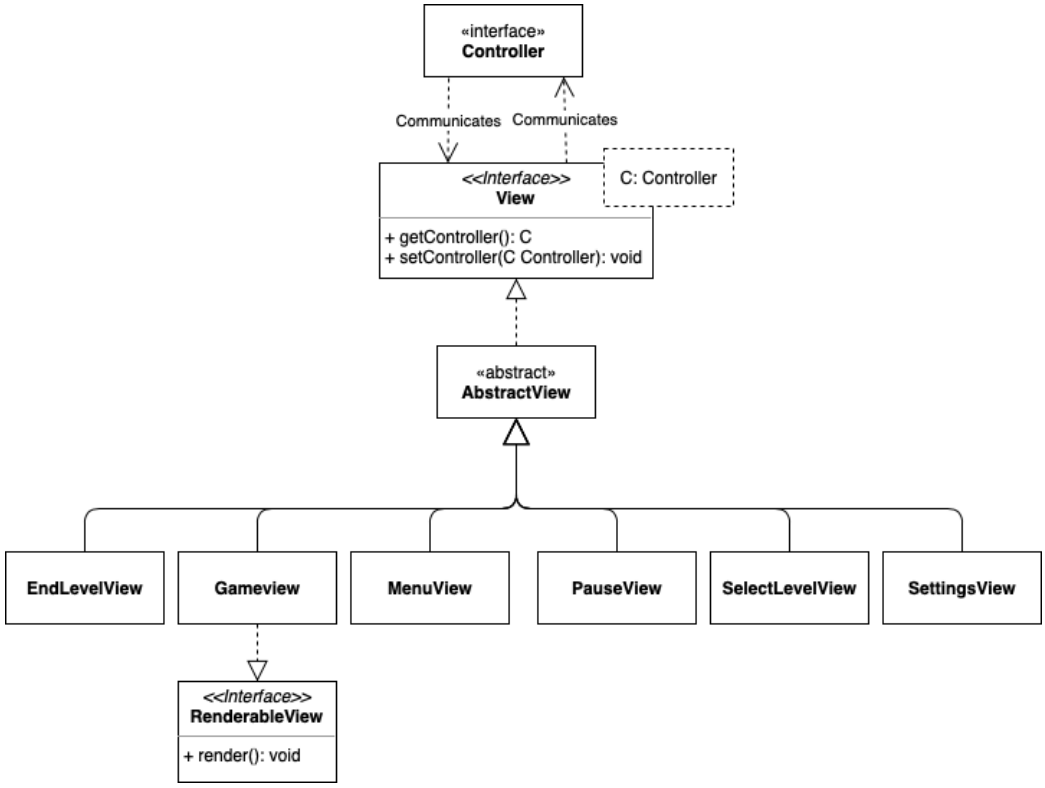

Figura 2.2: View dell'applicazione

2.1.3 Controller

La figura del `Controller` rappresenta la parte centrale del nostro applicativo, in quanto svolge la funzione di comunicazione tra le parti model e view. Al fine di avere un controllo più rigido nella gestione delle dipendenze e più flessibile per ciò che concerne eventuali modifiche, abbiamo centralizzato le comunicazioni tra le parti in un'unica interfaccia `Linker`.

2.2 Design dettagliato

2.2.1 Giovanni Antonioni

Nel corso dello sviluppo del progetto mi sono occupato principalmente degli aspetti riguardanti la struttura architetturale del Model e del GameLoop. L'obbiettivo che mi sono posto fin da subito è stato quello di realizzare un qualcosa di estendibile cercando di rispettare nella maniera più fedele possibile i principi DRY, KISS e SRP.

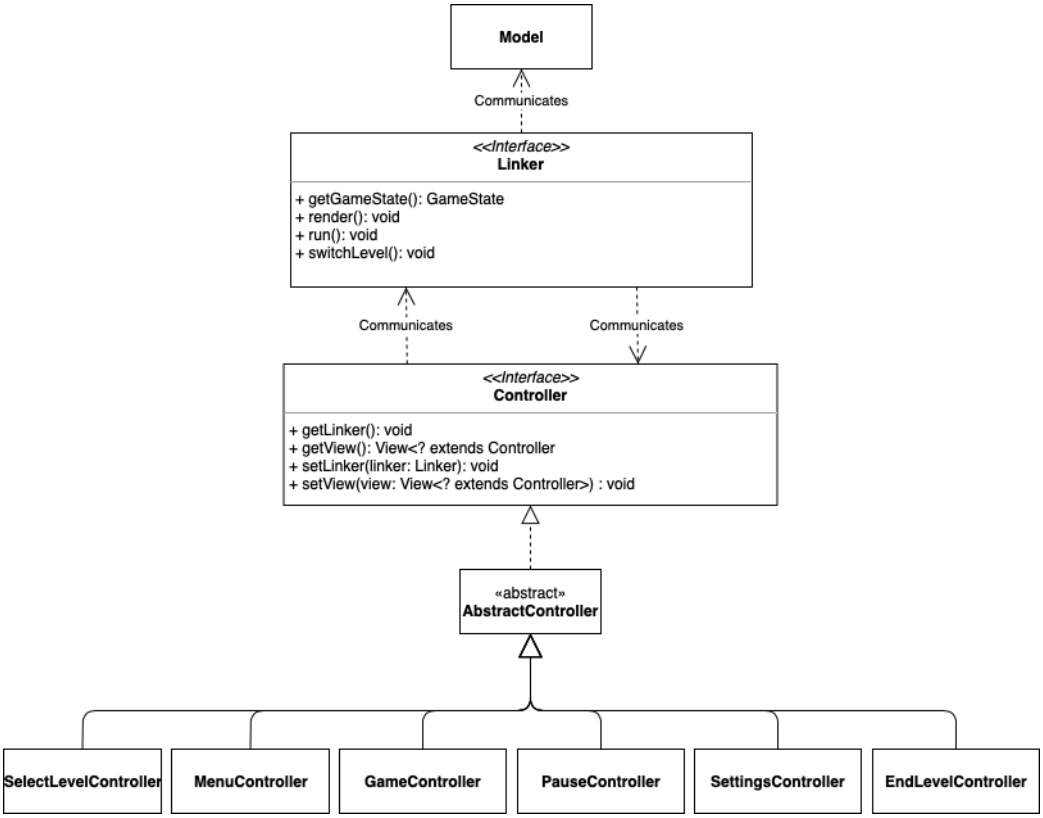

Figura 2.3: Controller dell'applicazione

ECS

In fase di analisi e progettazione dell'applicazione è emerso come questa possieda più elementi costitutivi che lavorano su domini applicativi differenti, questo aspetto comporta la costruzione di oggetti con una grande concentrazione di codice tendenti ad essere eccessivamente complessi, difficili da mantenere e modificare.
Nel cercare di prevenire questo scenario ho voluto adoperare il pattern **Entity-Component System** che permette di definire più moduli denominati *Components* i quali possono essere collegati (attached) o scollegati (detached) dinamicamente a delle classi container chiamate *Entities*.
In Figura 2.4 è mostrato lo schema architetturale che descrive i *Components*.
Ho realizzato l'interfaccia `Component` contenente i metodi necessari per collegare e scollegare un componente da un'entità ed un metodo per ottenere il riferimento a quest'ultima.
Ogni componente specializza l' `Entity` a cui è collegato fornendo dei metodi atti a gestirne un determinato comportamento:

- `UpdateComponent`: Gestisce gli aspetti riguardanti l'aggiornamento dello stato dell'Entità a cui è collegato.
- `SpriteComponent`: Gestisce la rappresentazione grafica tramite sprite dell'entità a cui è collegato. Adopera il pattern **Template** attraverso il metodo `getFileName()`.
- `EventComponent`: Implementa la gestione degli eventi dell'entità a cui è collegato. L'interfaccia non aggiunge nuovi metodi ma viene adoperata per l'identificazione della tipologia di componente che rappresenta.

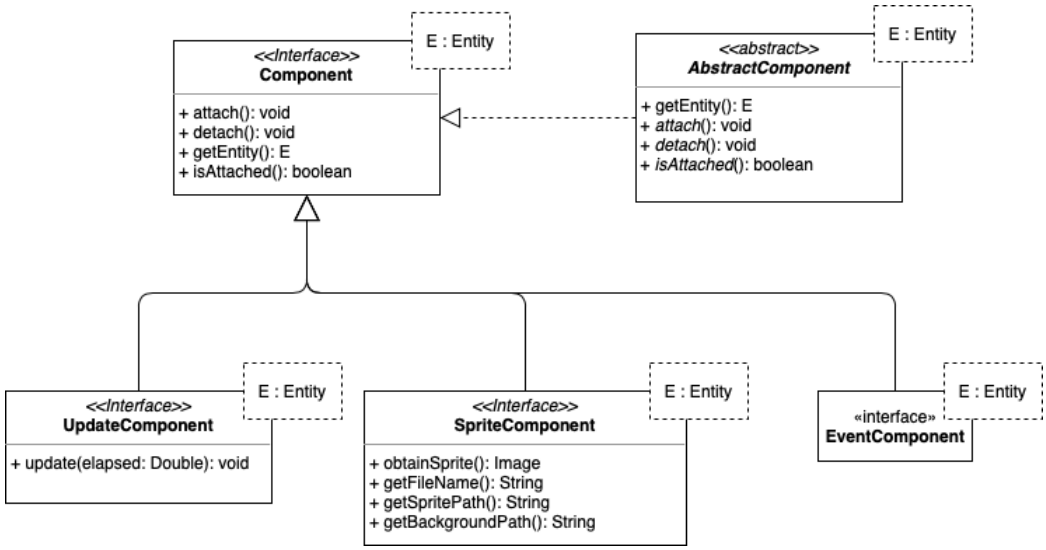

Figura 2.4: Schema UML architetturale `Component`

A ciascun `Component`, al momento della creazione, viene passato il riferimento dell'`Entity` che lo conterrà; risulterà, quindi, che il metodo `getEntity()` avrà logiche comuni a qualsiasi sua estensione mentre i metodi `attach()`, `detach()` e `isAttached()` varieranno in base alla tipologia di `Component` interessata. In Figura 2.5 la classe `AbstractComponent` rappresenta l'estensione di `Component` più astratta ed implementa il solo metodo `getEntity()` comune a ciascun componente. Le estensioni `AbstractUpdateComponent`, `AbstractSpriteComponent` e `AbstractEventComponent` rappresentano le astrazioni relative ad ogni specifica tipologia di `Component` ed in ciascuna di queste vengono implementati i metodi `attach()`, `detach()` e `isAttached()`.

In Figura 2.6, vengono descritte, infine, le *Entities*. L'interfaccia `Entity` contiene i metodi per registrare, rimuovere, recuperare e verificare l'esistenza

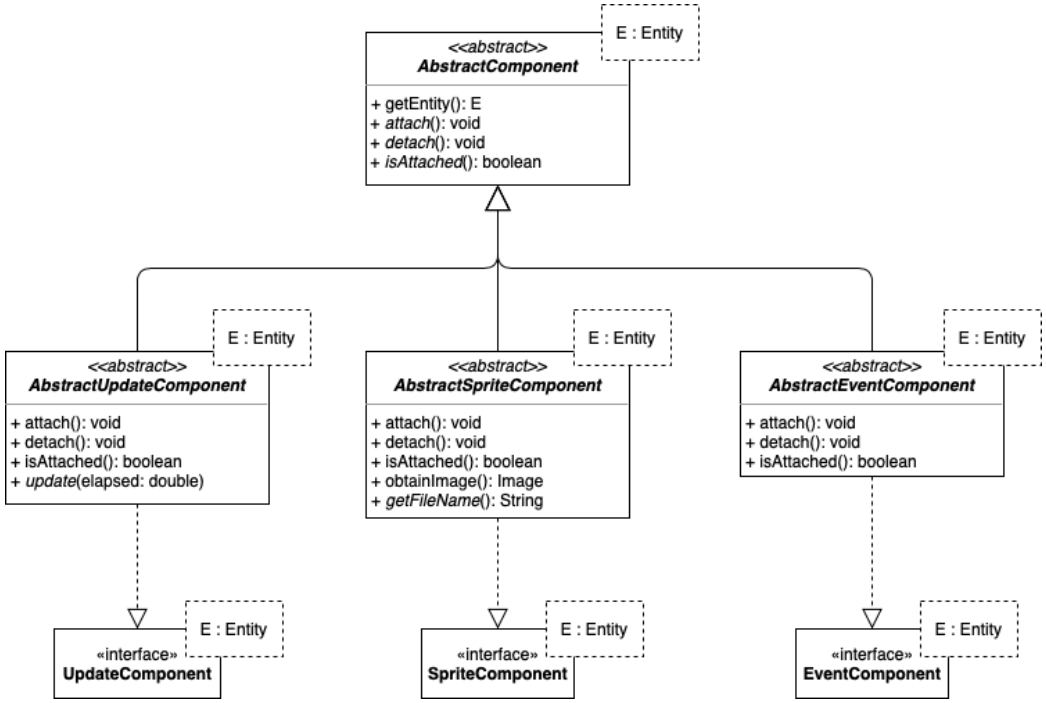

Figura 2.5: Astrazione di `Component`

di un componente all'interno di un'entità. Un'`Entity` può avere zero o più componenti registrati al suo interno, tutti di tipologia diversa e non ancora collegati.

Gestione Eventi

E' stato necessario fin da subito scegliere una strategia efficiente per la corretta gestione degli eventi all'interno dell'applicazione. Questa infatti, pur presentando delle meccaniche di gioco elementari, è composta da svariati elementi che necessitano di uno scambio di informazioni continuo al seguito dell'avvenimento di una determinata azione. Un evento descrive una qualsiasi azione che si verifica all'interno dell'applicazione e associa ad essa dei dati che la caratterizzano.
Per la soluzione del problema sono stati inizialmente presi in considerazione i pattern **Observer** e **Mediator** i quali, tuttavia, presentano alcuni difetti se inseriti nel contesto del seguente progetto.
Nel primo, ogni componente che emette degli eventi (detto anche *Subject*), deve poter permettere la registrazione di osservatori (*Observers*) interessati alla ricezione di quest'ultimi.
Questa soluzione risulta poco scalabile in quanto ogni *Observer* che necessita

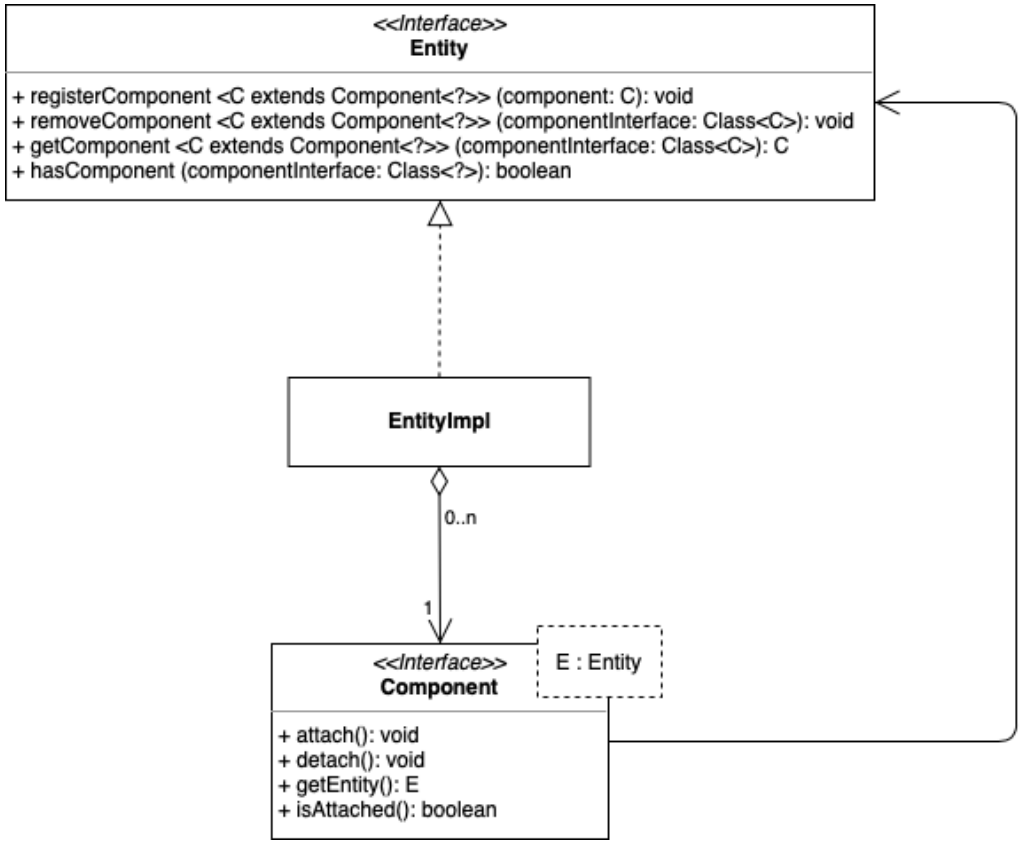

Figura 2.6: Schema UML architetturale di `Entity`

di ricevere degli eventi deve registrarsi ad ogni *Subject* a cui è interessato: tanti più sono i *Subjects* a cui registrarsi, tanto più risulta difficile mantenere il codice relativo a questo sistema.
Una soluzione alternativa potrebbe essere quella di utilizzare **Mediator** e centralizzare la gestione degli eventi. Tramite questo approccio potremmo ricadere tuttavia in un problema accennato precedentemente, ovvero, lo sviluppo di un oggetto con un'eccessiva concentrazione di codice e, di conseguenza, difficilmente mantenibile e poco scalabile.
Date le seguenti premesse ho deciso di adoperare il pattern **Publish-Subscribe**. Viene introdotta la classe `EventBus` la quale permette la sottoscrizione di oggetti `Subscriber` alla ricezione di eventi specifici. All'interno dell'applicazione può esistere una sola ed unica istanza di `EventBus` richiamabile globalmente, per tale motivo viene adoperato il pattern **Singleton**; l'istanza è recuperata attraverso il metodo `getDefault()`.Un `Subscriber` si registra all'interno di `EventBus` tramite il metodo `register()` e si disiscrive tramite `unregister()`.

Nel momento in cui un'entità (`Publisher`) deve notificare un evento ri-

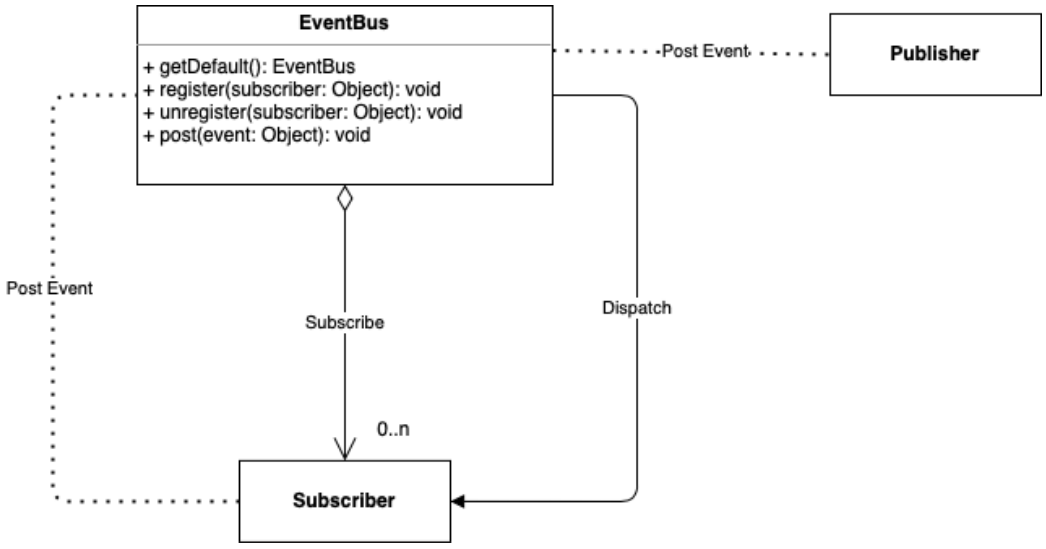

Figura 2.7: Funzionamento `EventBus`

chiama il metodo `post()` di `EventBus` passando come argomento un oggetto contenente informazioni riguardo il tipo di evento che si è verificato e i dati che questo trasporta.
L'`EventBus` prende in carico l'evento ricevuto e trasporta le sue informazioni a tutti i `Subscriber` interessati a riceverlo.
Tale organizzazione presenta dei vantaggi notevoli, in quanto viene separata la gestione degli eventi, demandata agli stessi *Subscribers*, dalla gestione del dispatching di un evento, la quale rimane centralizzata e controllata dall'unica entità `EventBus`.
All'interno del Model ogni `Entity` interessata alla ricezione di eventi possiederà uno specifico `EventComponent` che agirà scome `Subscriber`.
Per quanto riguarda la modellazione degli oggetti rappresentanti un evento viene messa a disposizione un'interfaccia `Event`, la quale sarà estesa in base al tipo di evento che si vuole creare.
Per l'agevolazione della creazione di `Event` ho introdotto una **factory** statica chiamata `Events`.

Power-up

Iniziamo a trattare la gestione dei power-up partendo dalla loro rappresentazione interna all'arena di gioco (Figura 2.10). Un oggetto rappresentante un power-up implementa l'interfaccia `Powerup`, estensione di `Element`. Alla

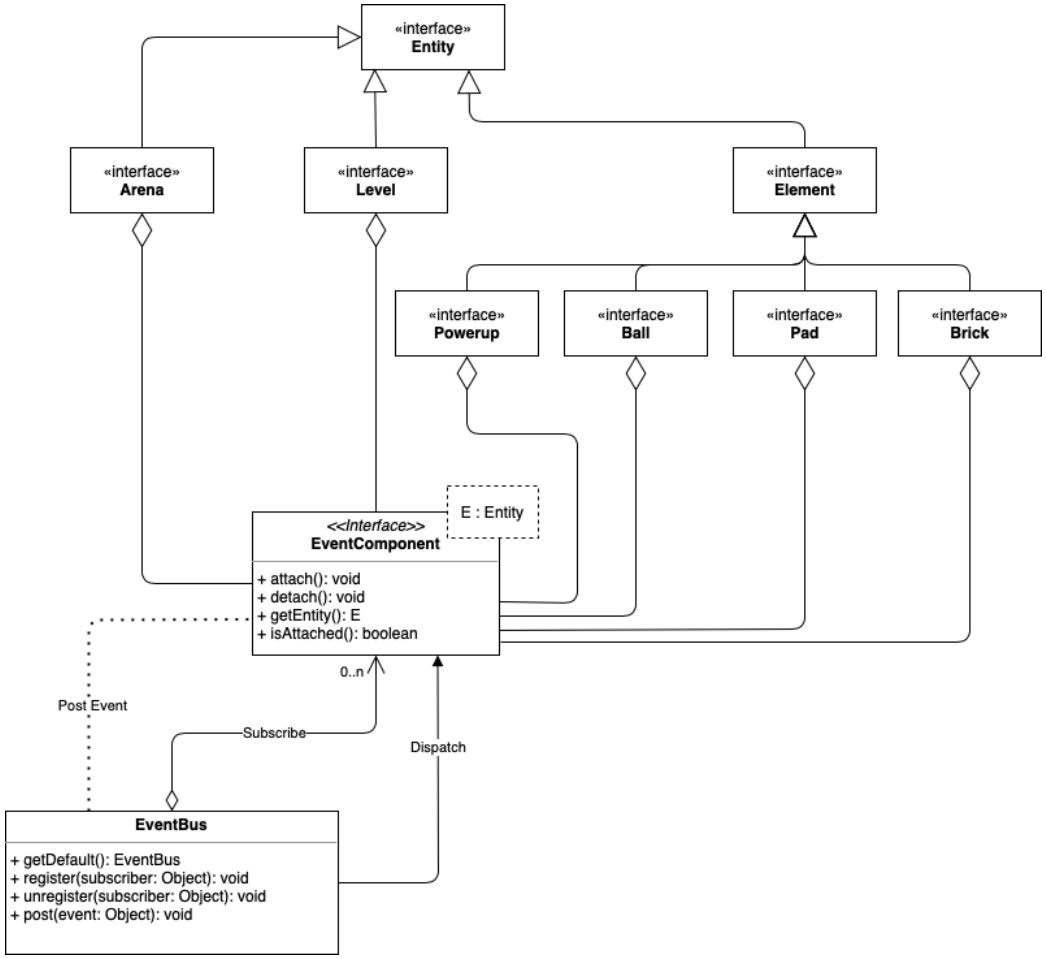

Figura 2.8: Gestione degli eventi tramite `EventComponent` i quali agiscono da `Subscriber`

creazione di una nuova istanza viene attribuita a questa un valore dell'enumeratore `PowerupType` il quale verrà impiegato per adoperare il pattern **Strategy**. Un power-up, infatti, applica un determinato effetto e viene visualizzato nell'arena in maniera differente in base al tipo di `PowerupType` assegnatogli.
La procedura di generazione avviene dopo la distruzione di un mattoncino, quando un'istanza di `Brick` invia all'arena di gioco (`Arena`) l'evento `BrickDestructionEvent` indicante la sua distruzione.
Se quest'ultima ha portato alla generazione di una nuova istanza di `Powerup` il brick distrutto viene rimosso ed il power-up sarà posizionato alle coordinate passate tramite evento.
La fisica dell'elemento è gestita attraverso il componente `PowerupUpdateComponent`,

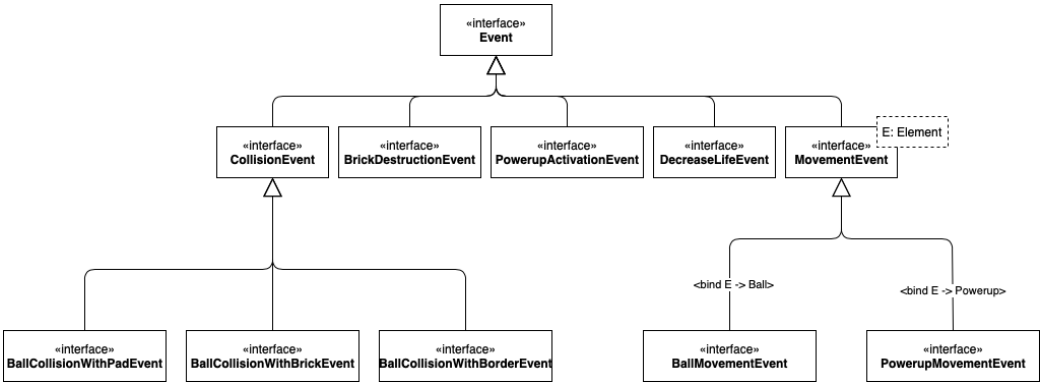

Figura 2.9: Gerarchia degli eventi presenti all'interno dell'applicazione

il quale sarà collegato a `Powerup` al momento della sua creazione e permetterà, ad ogni suo aggiornamento, di farlo scendere verso il fondo dell'arena facendogli mantenere una velocità costante.
Ad ogni aggiornamento della sua posizione, il `Powerup` genera un nuovo evento `PowerupMovementEvent`, il quale è intercettato dall'istanza di `Pad`.
Al verificarsi di tale evento il `Pad` controlla se stia collidendo con un oggetto `Powerup` e, in caso affermativo, avvia la procedura dell'applicazione dell'effetto, inviando l'evento `PowerupActivationEvent` il quale verrà gestito nuovamente da `Arena`.

Power-up Handler

All'interno dell'applicazione ciascun effetto di power-up attivo è rappresentato da un thread temporizzato che ha il compito di applicarne e rimuoverne gli effetti. Ciascuno di questi deve avere la possibilità di essere messo in pausa o essere immediatamente interrotto in base a determinate condizioni (Figura 2.11).
L'interfaccia `PowerupEffect` mette a disposizione i metodi per applicare e rimuovere gli effetti di un power-up, identificarne il tipo e determinarne la loro durata in secondi.
La creazione e gestione dei threads degli effetti è affidata al componente `PowerupHandler`, la cui interfaccia presenta i metodi necessari per aggiungere un nuovo effetto e impostare tutti i threads in stato di pausa o di interromperli.
`PowerupHandler` adopera il **Thread Pool** pattern, gestito in una classe interna chiamata `InternalExecutor` che estende `PausablePool`, per le logiche di pausa dei threads, e implementa i metodi di `PowerupPool` per l'immissione di un thread all'interno del pool.

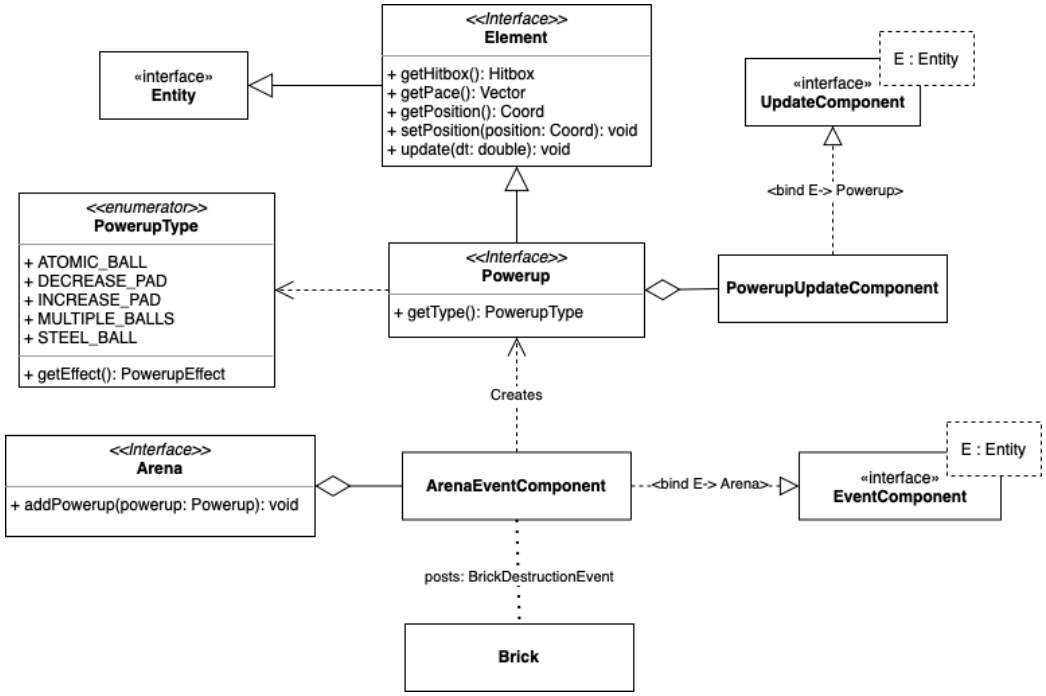

Figura 2.10: Schema UML architetturale di `Powerup`

Level

Per la gestione dei livelli ho ideato l'interfaccia `Level`, il cui compito è fornire i metodi necessari atti a memorizzare un punteggio, gestire le vite totali messe a disposizione del giocatore e controllare un'istanza di `Arena` (Figura 2.12).
Ho voluto utilizzare uno **State** pattern per indicare in che stato si trovi il livello durante lo svolgimento della partita assegnando a questo uno dei valori dell'enumeratore `LevelStatus`.
`Level` implementa l'interfaccia `Entity` ed è in grado di registrare al suo interno dei *Components*, in particolare, sono assegnati a lui in fase di creazione un `UpdateComponent` ed un `EventComponent`.
L'interfaccia `LevelLoader` dispone di un unico metodo nominato `fromFile()`, che permette di creare un nuovo livello di gioco tramite un file di configurazione.

GameState

Per il controllo del flusso della partita ho realizzato un componente `GameState` che, in maniera analoga a `Level`, adopera uno `State` pattern (Figura 2.13).

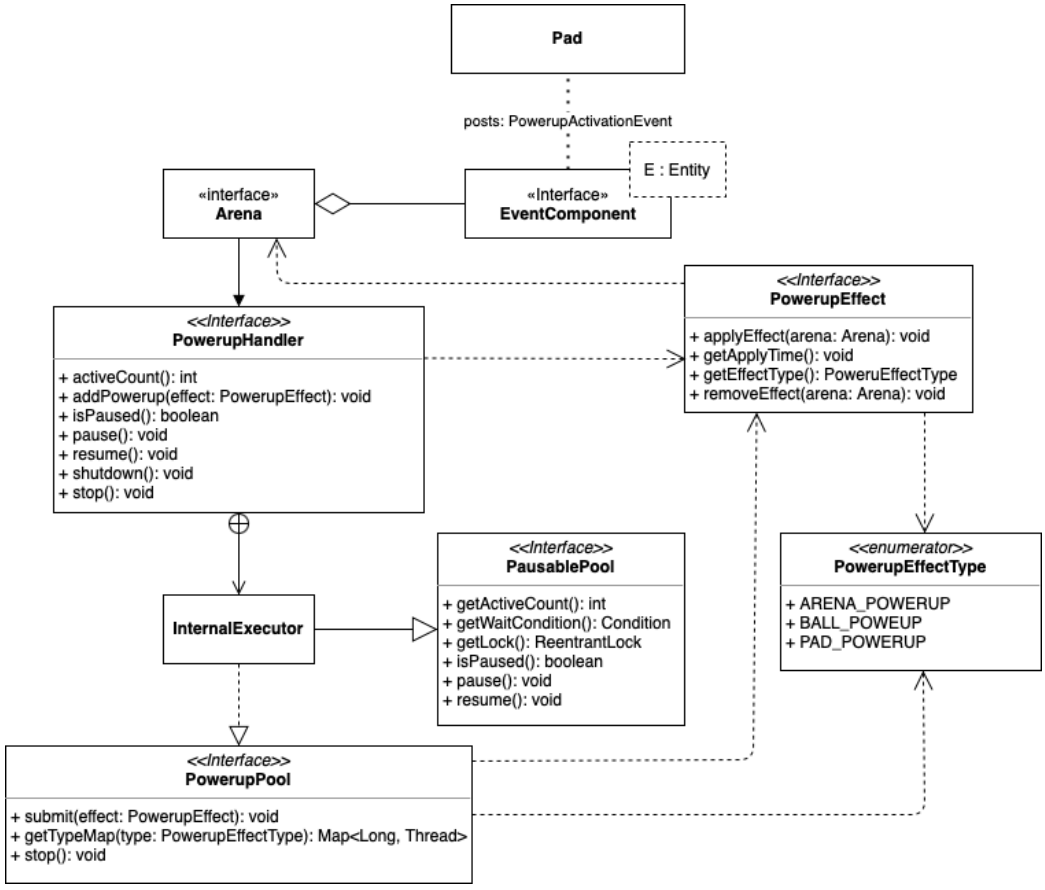

Figura 2.11: Gestione degli effetti dei power-up tramite `PowerupHandler`

Ciascun stato del gioco è rappresentato da un valore dell'enumeratore `StateEnum` ed influisce sul comportamento globale dell'applicazione. `GameState` modifica il suo stato in base alla rilevazione di:

- **Input esterni**: derivanti da mosse del giocatore o al cambio di una vista del gioco.
- **Input interni**: l'aggiornamento del dominio del gioco può determinare un cambio di stato.

Tramite `GameState` è possibile inoltre cambiare o selezionare uno specifico livello da giocare, ottenere il punteggio totale della partita e aggiornare il dominio di gioco. Questo rappresenta di fatto l'unico componente del model in grado comunicare con la parte controller dell'applicazione.

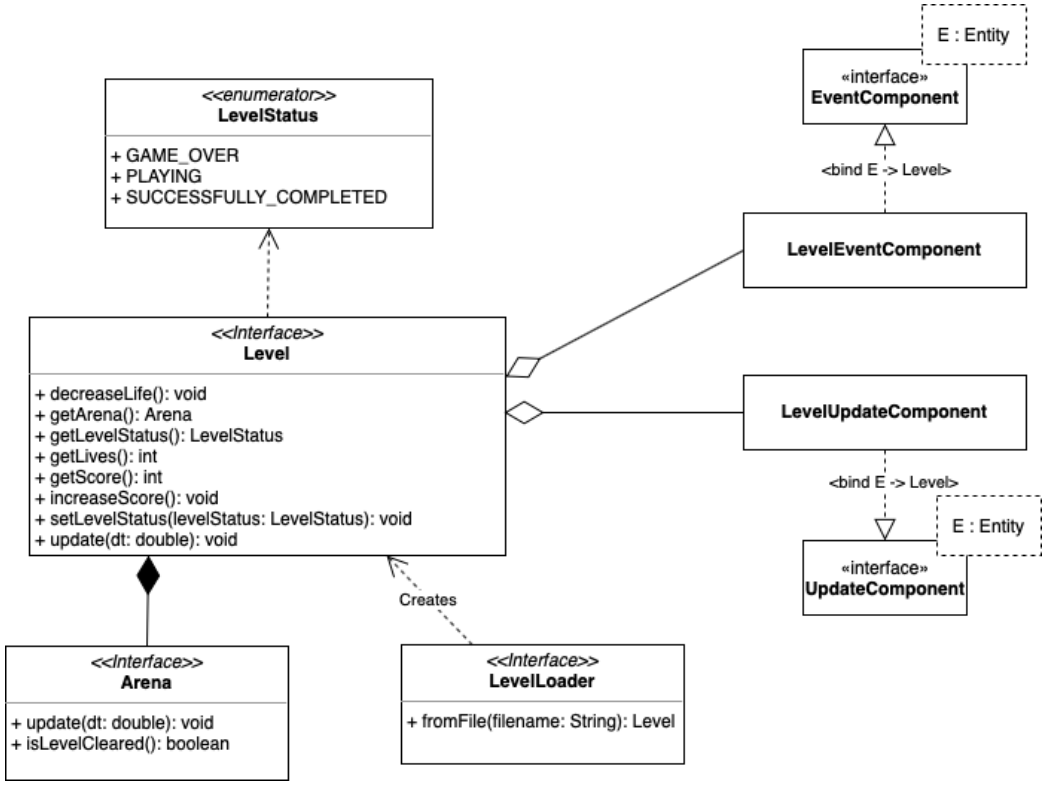

Figura 2.12: Schema strutturale UML di `Level`

Game Loop

Ho realizzato, infine, il motore di gioco `GameLoop` (Figura 2.14). Questo non è altro che un thread che viene attivato all'avviarsi dell'applicazione, il cui compito è quello di entrare in un ciclo continuo nel quale si occupa di aggiornare il model richiamando il metodo `update()` del `GameState` e di farne visualizzare gli elementi all'interno di una vista di gioco.
Per eseguire queste operazioni `GameLoop` è collegato a `Linker` da cui ottiene il riferimento al `GameState` e alla `View` corrente. Una `View` può essere aggiornata da `GameLoop` solamente se estende l'interfaccia `RenderableView`. `GameLoop` si occupa di processare l'input di un utente passato tramite `Linker`, per farlo sfrutta il **command** pattern attraverso la classe `Command` contenente le informazioni circa le istruzioni da eseguire.

2.2.2 Luca Rubboli

Come contributo allo svolgimento del progetto mi sono occupato principalmente dell'organizzazione strutturale del caricamento delle View e del design

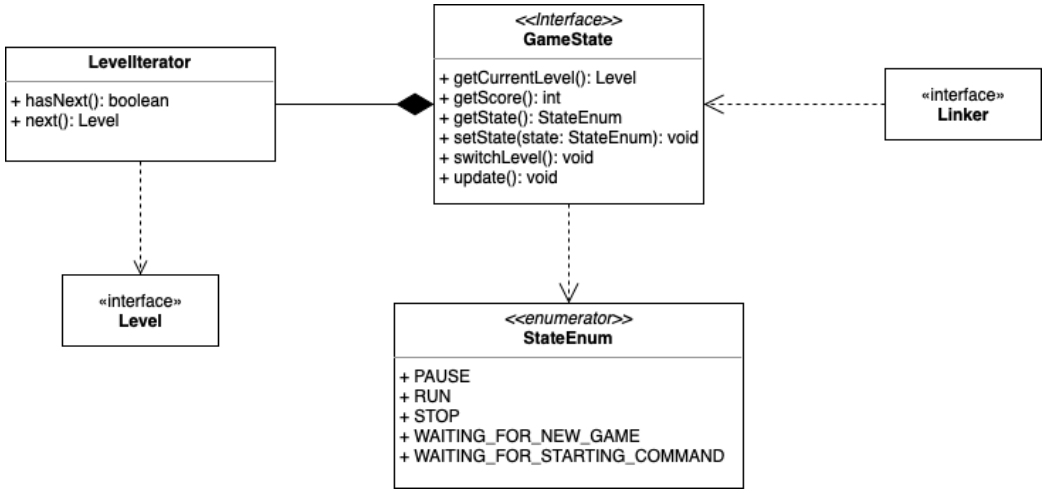

Figura 2.13: Schema strutturale UML di `GameState`

di alcune di esse, dell'entità centrale Linker, del design delle sprites e dell'implementazione di alcune parti del Model. L'obbiettivo cardine a cui ho aderito è quello di lavorare sull'estendibilità e sul cercare di predisporre nella maniera più agevole possibile spazio per future modifiche, oltre che prediligere un codice leggibile, rispettando i principi DRY, KISS e SRP.

Caricamento View

Per quanto concerne l'organizzazione del caricamento delle differenti View a seconda delle relative fasi di gioco, ho deciso di sviluppare due principali componenti (Figura 2.15). Il `Loader` ha il compito di caricare e, successivamente, legare correttamente, una tipologia di scena e il relativo `Controller`. Lo `SceneHandler`, invece, si occupa di richiedere la nuova scena al `Loader`, mostrarla graficamente e legare il nuovo `Controller` all'istanza di `Linker`. Inizialmente era stato valutato l'utilizzo di un pattern **Singleton** per poter accedere allo `SceneHandler`, ma con l'introduzione dell'entità `Linker` la riduzione delle dipendenze legate ad esso e la sua centralizzazione sono risultate un'alternativa altrettanto valida.

Linker

L'entità `Linker` è stata ideata per far fronte ad un problema di dipendenze multiple tra Model e View.
Alla sua creazione, istanzia e avvia il `Gameloop`, e gli conferisce il `GameState`

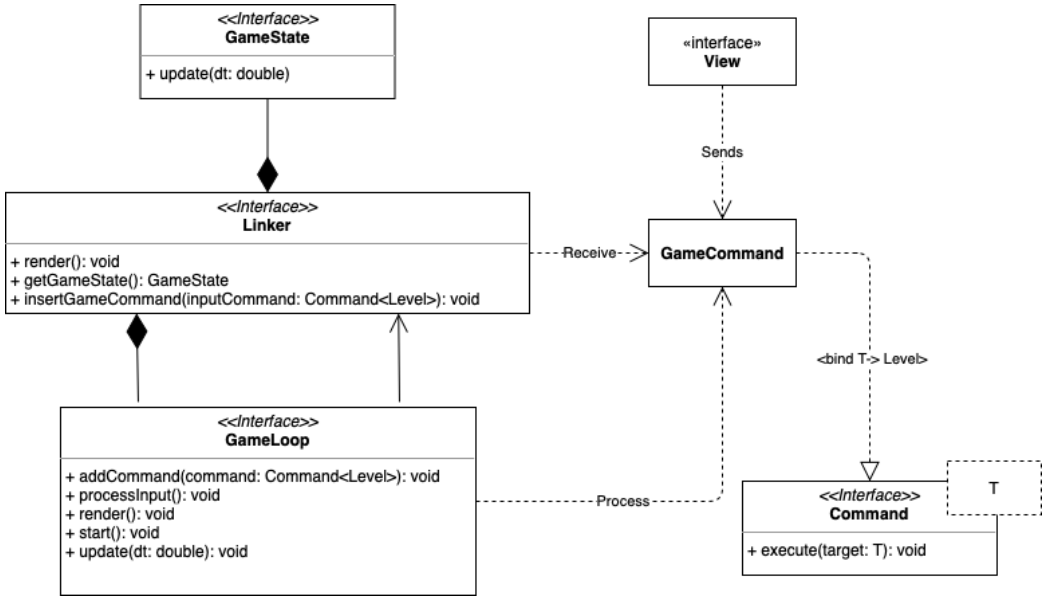

Figura 2.14: Schema strutturale UML di `GameLoop`

iniziale.
Oltre a ridurre le dipendenze, il `Linker` svolge diverse funzionalità:

- *Input utente*: gestisce i comandi in input dell'utente, eseguendoli subito in caso di comandi di stato, oppure accodandoli in caso di comandi di gioco. Come mostrato in Figura 2.16, per svolgere questa funzionalità si appoggia all'entità `InputHandler`, che è incaricata di stilare la lista, con annessa i relativi effetti, di comandi che l'applicativo deve gestire.
- *Centralizzazione*: offre un punto di accesso al Model a tutti i `Controller` delle View (Figura 2.17), che ne possiedono un'istanza, e richiama anche eventuali render grafici.
- *Cambi scena*: gestisce tutti i cambi scena, demandando allo `SceneHandler` (Figura 2.18) tutta la gestione del caricamento scena, e occupandosi della concretezza dei vari `GameState`.

Element

L'interfaccia `Element` è preposta ad incapsulare tutte le caratteristiche dei vari elementi che compongono l'arena di gioco. Le loro peculiarità fondamentali sono individuabili in:

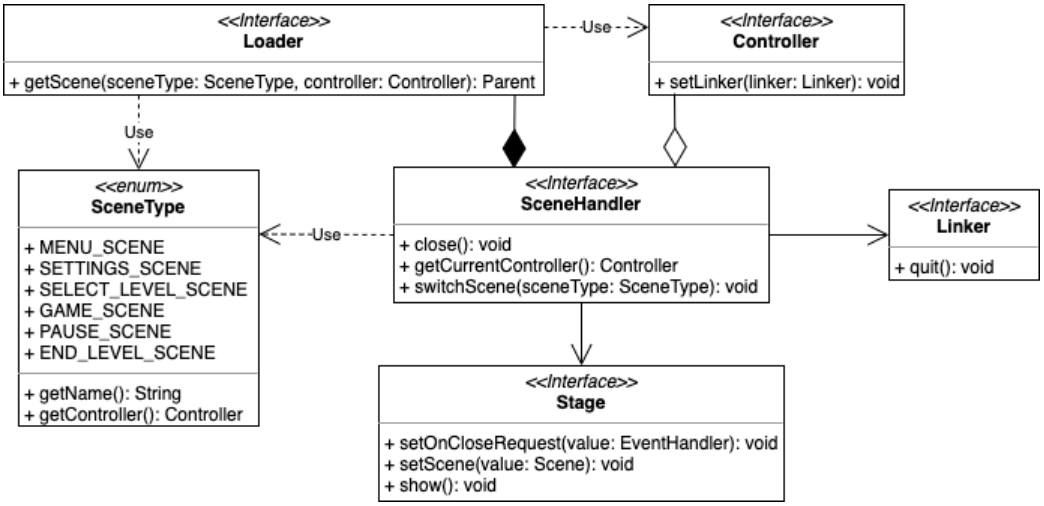

Figura 2.15: Schema strutturale UML di `SceneHandler`

- `Dimension`: Incapsula i concetti di lunghezza e larghezza di ogni elemento.
- `Position`: Rappresenta le coordinate dell'elemento all'interno dell'arena di gioco.
- `Pace`: Incapsula il concetto di vettore velocità, che permette, ad ogni update, il ricalcolo della nuova posizione, in caso si tratti di un elemento di movimento.
- `Hitbox`: Incapsula il concetto di area occupata dall'oggetto nell'arena di gioco.

La classe astratta `AbstractElement` (Figura 2.19) invece è stata organizzata per implementare tutti i metodi comuni agli `Element`. I metodi implementati vertono sulle caratteristiche principali degli `Element` stessi, ad esclusione della velocità, per cui vi è distinzione tra `Element` di movimento e fissi.

Ball

Come altra sezione di Model ho implementato l'elemento `Ball` (Figura 2.20), che, appunto, estende `Element`. In prima battuta ho deciso di assegnare alla `Ball` uno stato, rispettando uno **Strategy** pattern, che ne incapsula diverse peculiarità. I differenti stati sono raccolti in `BallType`, ognuno di essi presenta 3 importanti peculiarità della `Ball`.

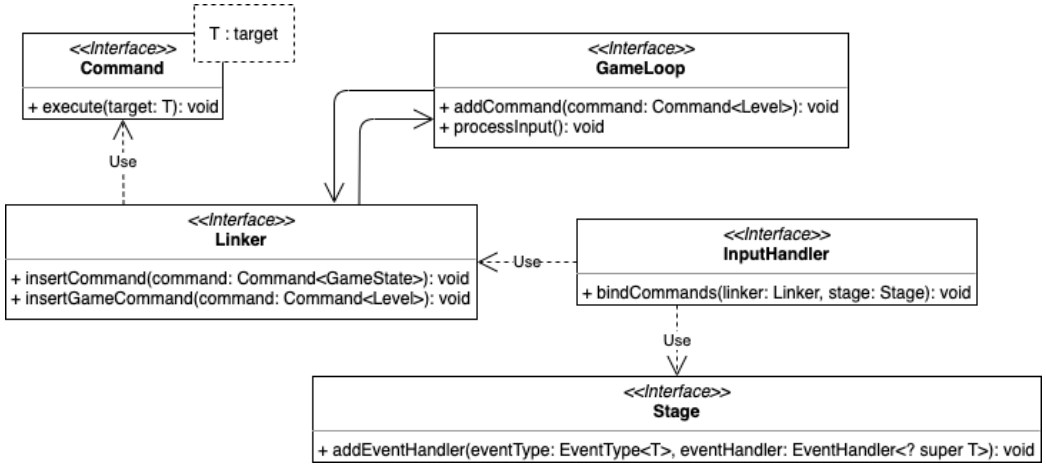

Figura 2.16: Schema strutturale UML di `Linker` per la gestione dell'input

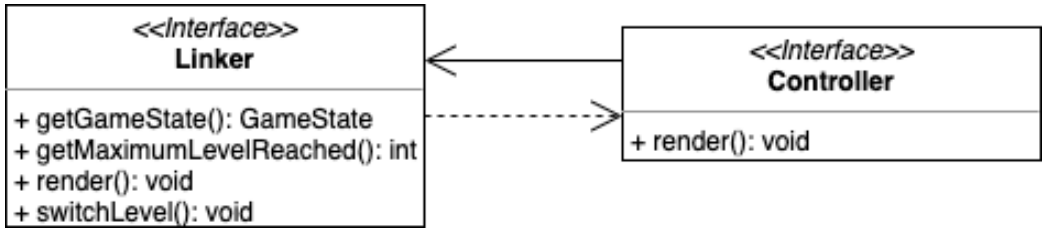

Figura 2.17: Schema strutturale UML di `Linker` applicato alla centralizzazione del Model

- *Danno*: Indica il danno che la `Ball`, in caso di collisione con un `Brick` distruttibile, arreca a quest'ultimo. Per poter rappresentare un danno infinito, abbiamo optato per l'utilizzo di Opzionali, assegnandoli vuoti per indicare questa peculiarità, oppure con un valore definito per indicare il danno effettivo.

- *Moltiplicatore di velocità*: Indica il fattore moltiplicativo del vettore velocità di `Ball`.

- *Rimbalzo*: Indica se la `Ball` potrà, o meno, rimbalzare come risposta ad una collisione con un `Brick` distruttibile. Il rimbalzo su tutte le altre superfici rimarrà invece invariato.

Essendo la `Ball` un'estensione di `Entity`, è in grado di possedere dei *Components*:

- `UpdateComponent`: Gestisce tutte le peculiarità relative a movimenti e rimbalzi della `Ball` stessa.

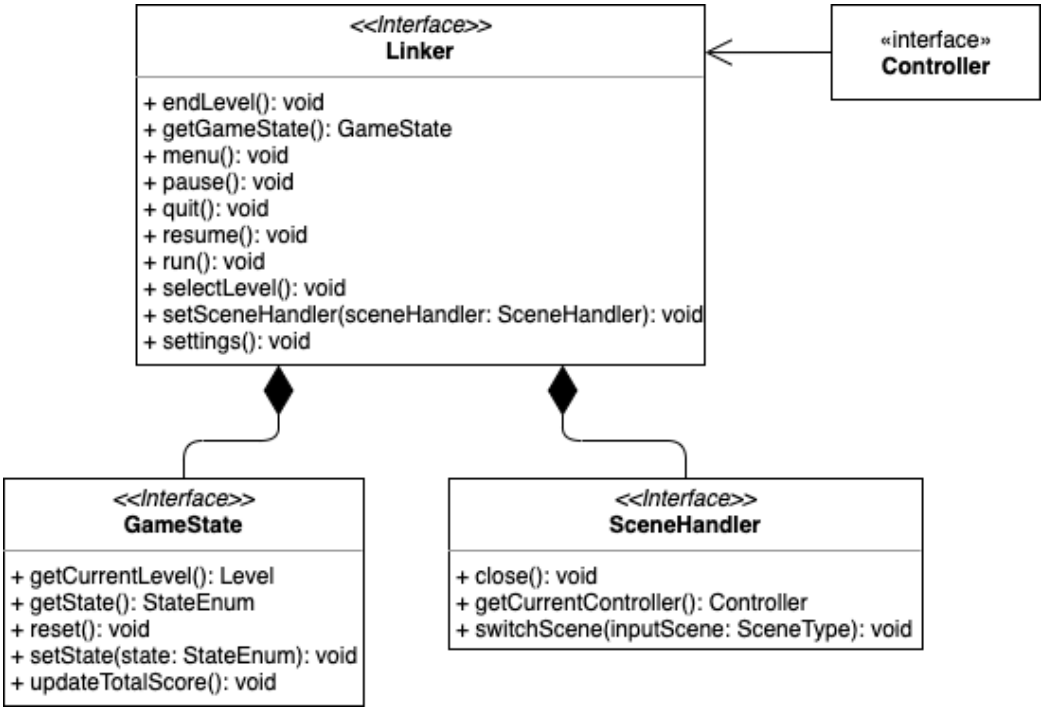

Figura 2.18: Schema strutturale UML di `Linker` per la gestione cambi scena

- `EventComponent`: Gestisce tutta la sezione di eventi che riguardano la `Ball`.

Per quanto riguarda invece la costruzione della `Ball` ho deciso di utilizzare il pattern **Builder**, la scelta è stata fatta per agevolarne la semplicità di costruzione, agendo comunque su tutte le sue peculiarità, e per garantire la concretezza dell'elemento prima di poterlo effettivamente istanziare.

2.2.3 Riccardo Traini

Nel corso dello sviluppo del progetto mi sono principalmente occupato della fisica e dell'Arena di gioco del Model, della stampa dell'Arena di gioco con tutti i suoi elementi nella View e degli effetti sonori e musica di sottofondo. Fin da subito mi sono imposto di rispettare i principi DRY, KISS e SRP.

Fisica di gioco

La fisica di gioco è definita dai movimenti degli elementi di gioco e dalle loro collisioni. Una collisione tra elementi di gioco è un fenomeno che si

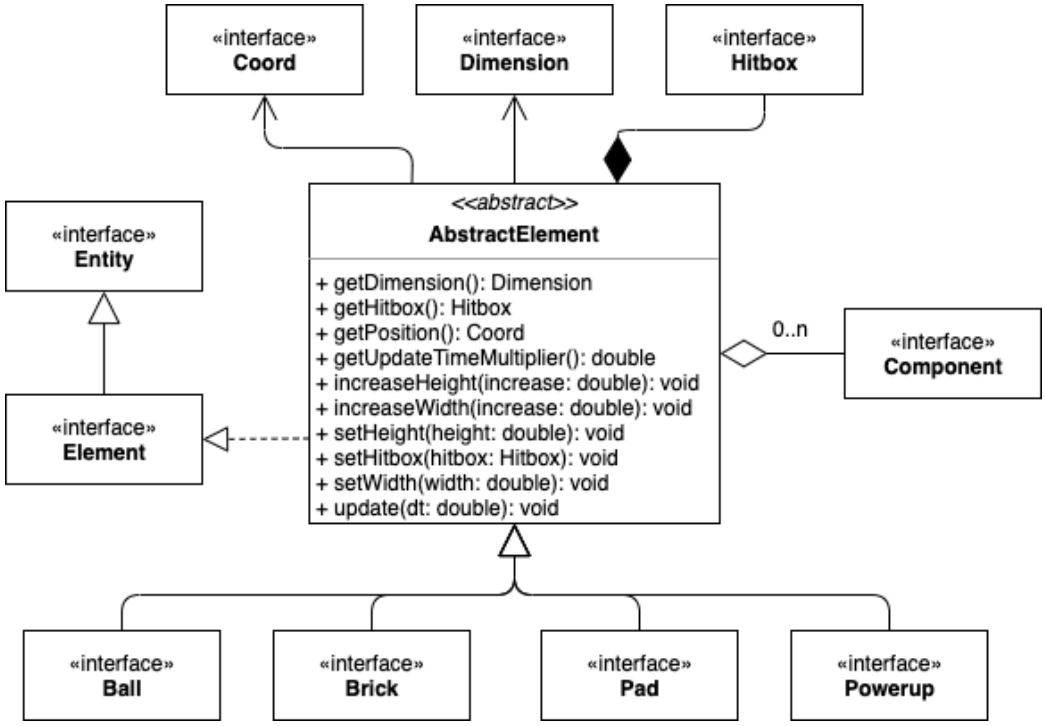

Figura 2.19: Schema strutturale UML di `AbstractElement`

verifica quando uno di questi entra a contatto con gli altri. Ad ogni elemento viene assegnata una sua `Hitbox` (Figura 2.21), che può essere circolare (`BallHitbox`) o rettangolare (`RectHitbox`).

- **Controllo delle collisioni**: Le `Hitbox` stesse si occupano dei calcoli per il controllo di un'eventuale collisione tra due di esse. Le informazioni di ogni collisione vengono gestite dalla `CollisionInformation`, che gestisce sia il lato dell'elemento dove è avvenuta la collisione, sia l'offset (**??**) che deve essere applicato alla posizione della `Ball` solo nel caso la collisione si verifichi tra `BallHitbox` e `RectHitbox`; l'offset viene utilizzato per spostare la `Ball` nella posizione più vicina per cui non si verifichi più una collisione.
- **Applicazione delle Collisioni**: Le collisioni vengono per lo più gestite tramite eventi. Ogni `Ball` immagazzina tutte le `CollisionInformation` ricevute e le applica ad ogni update del `Gameloop`.

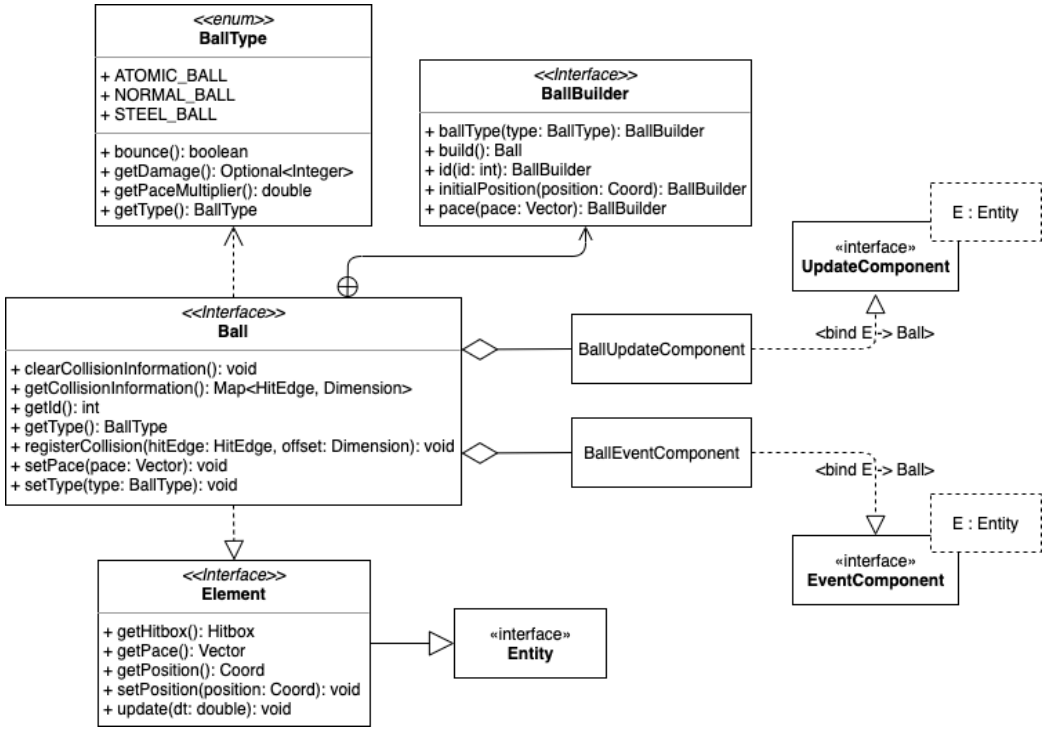

Figura 2.20: Schema strutturale UML di `Ball`

Arena di gioco

L'`Arena` (Figura 2.22) è la classe del Model che identifica l'area di gioco effettiva. Essa contiene tutti gli elementi di gioco, e ne gestisce aggiunta e rimozione, insieme al movimento del `Pad` e ai check delle collisioni con i bordi dell'`Arena` stessa.

GameView

La `GameView` (Figura 2.23) è il controller di JavaFX che si occupa di stampare a video l'`Arena` stessa con i suoi elementi. Al suo interno è possibile individuare più componenti che assolvono le varie funzionalità al fine di ottenere una stampa corretta dell'`Arena`:

- `Drawer`: Si occupa del disegno sul Canvas di JavaFX della `GameView` delle immagini rappresentanti gli elementi dell'`Arena` (Figura 2.24).
- `Binder`: Per mantenere il "ratio" del Canvas e ridimensionare la grandezza del font del testo delle Label, indipendentemente dalla dimensione della finestra dell'applicazione, ho utilizzato dei `Binder` (Figu-

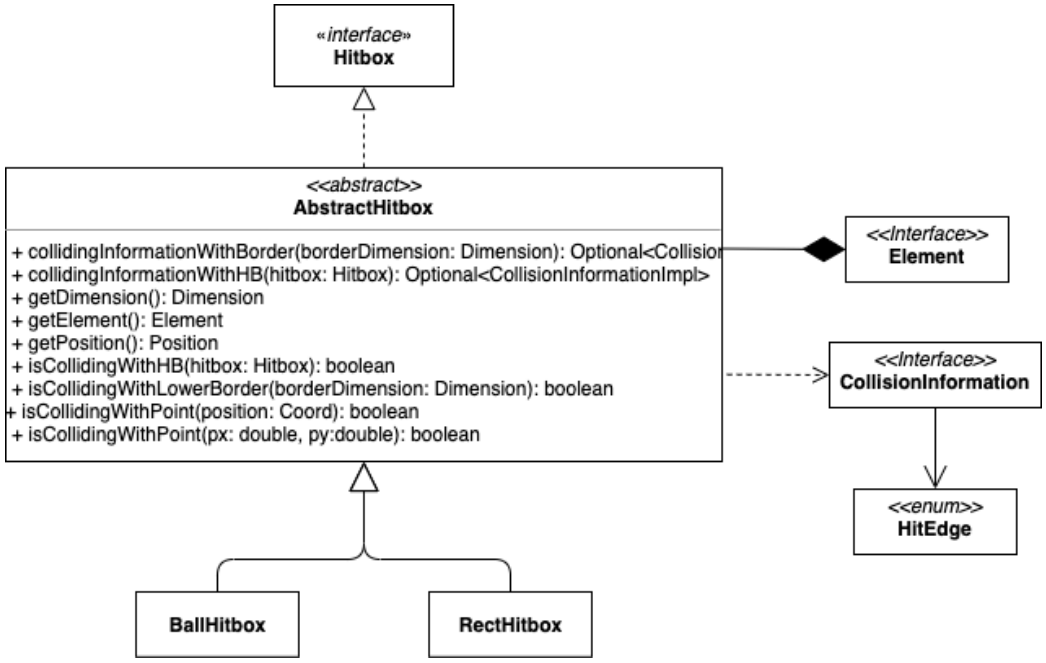

Figura 2.21: Schema strutturale UML di `Hitbox`

ra 2.25) che collegano delle precise proprietà del Canvas (CanvasRatioBinder) e delle Label (LabelSizeBinder) al Pane principale (Main Pane) della `GameView`.

Il `CanvasRatioBinder` si occupa di decidere quale delle due dimensioni del Main Pane è quella restrittiva, e fare il bind delle WidthProperty e HeightProperty di conseguenza, sempre mantenendo il "ratio" iniziale. Invece il `LabelSizeBinder` si occupa di ingrandire o rimpicciolire la grandezza del font del testo in base alla larghezza del Main Pane.

SoundPlayer

Per la gestione degli effetti sonori e della musica di sottofondo ho utilizzato una classe (Figura 2.26) con metodi statici, che per la riproduzione dei file in formato ".wav" utilizza la classe `MediaPlayer` di JavaFX. Per sincronizzare gli effetti sonori direttamente con gli eventi che avvengono nel model viene istanziato un `SoundEffectEventHandler` che viene iscritto all'`EventBus`.

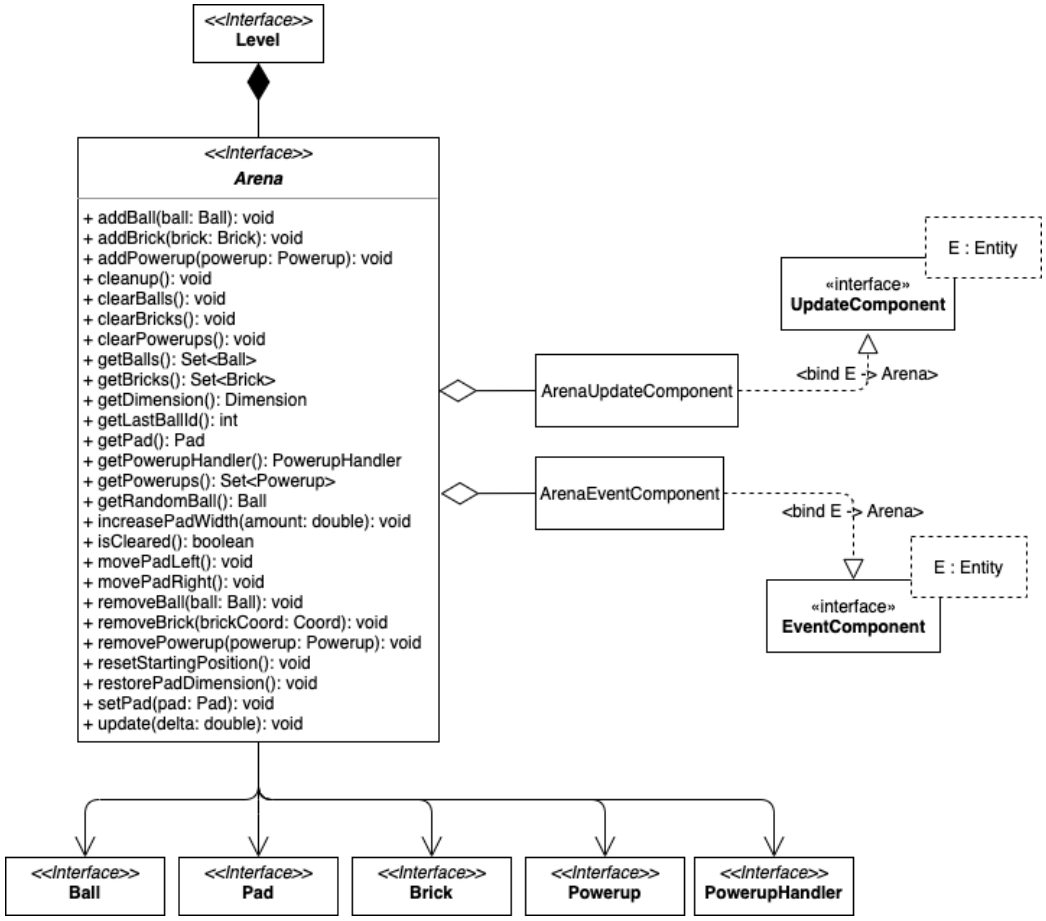

Figura 2.22: Schema strutturale UML di `Arena`

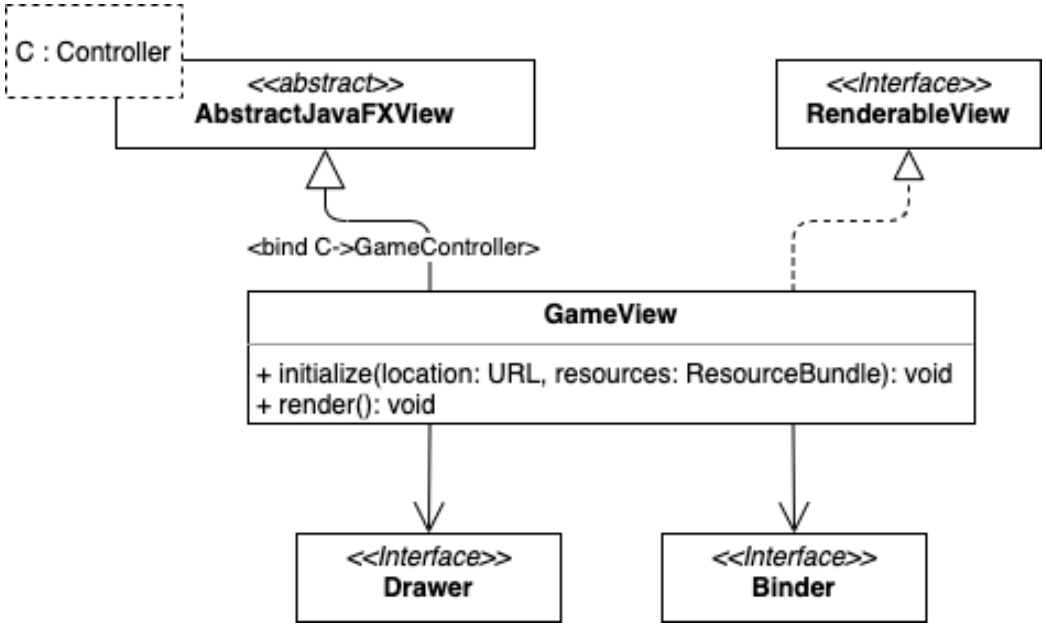

Figura 2.23: Schema strutturale UML di `GameView`

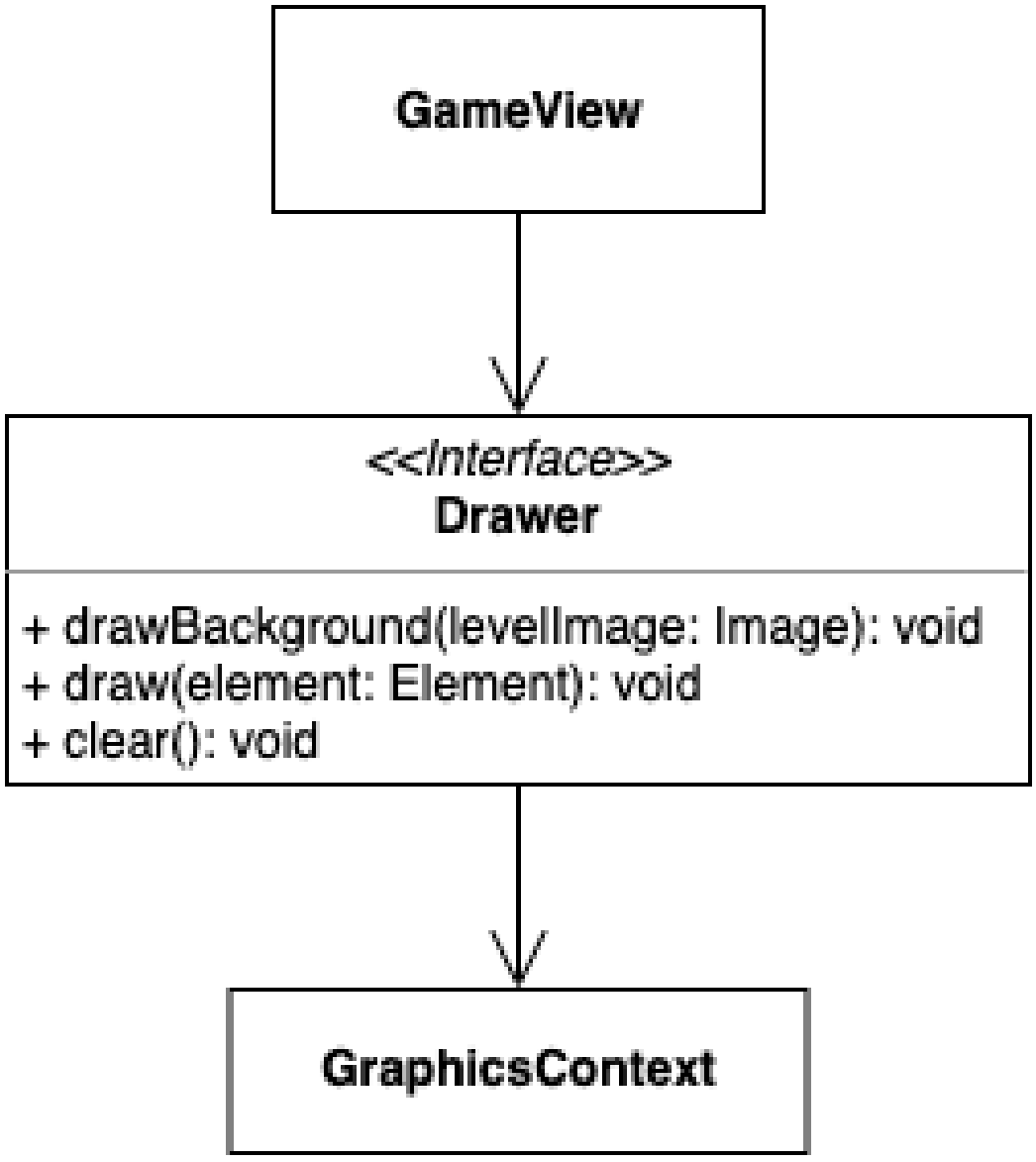

Figura 2.24: Schema strutturale UML di `Drawer`

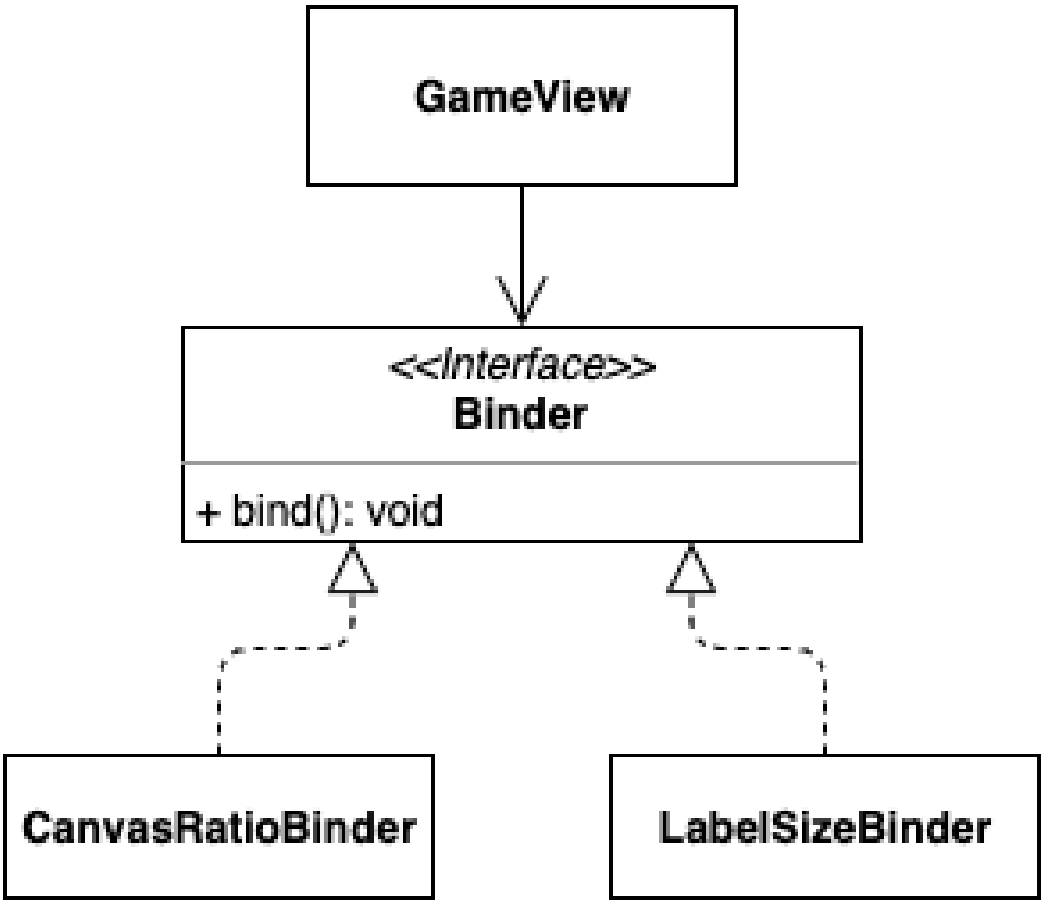

Figura 2.25: Schema strutturale UML di `Binder`

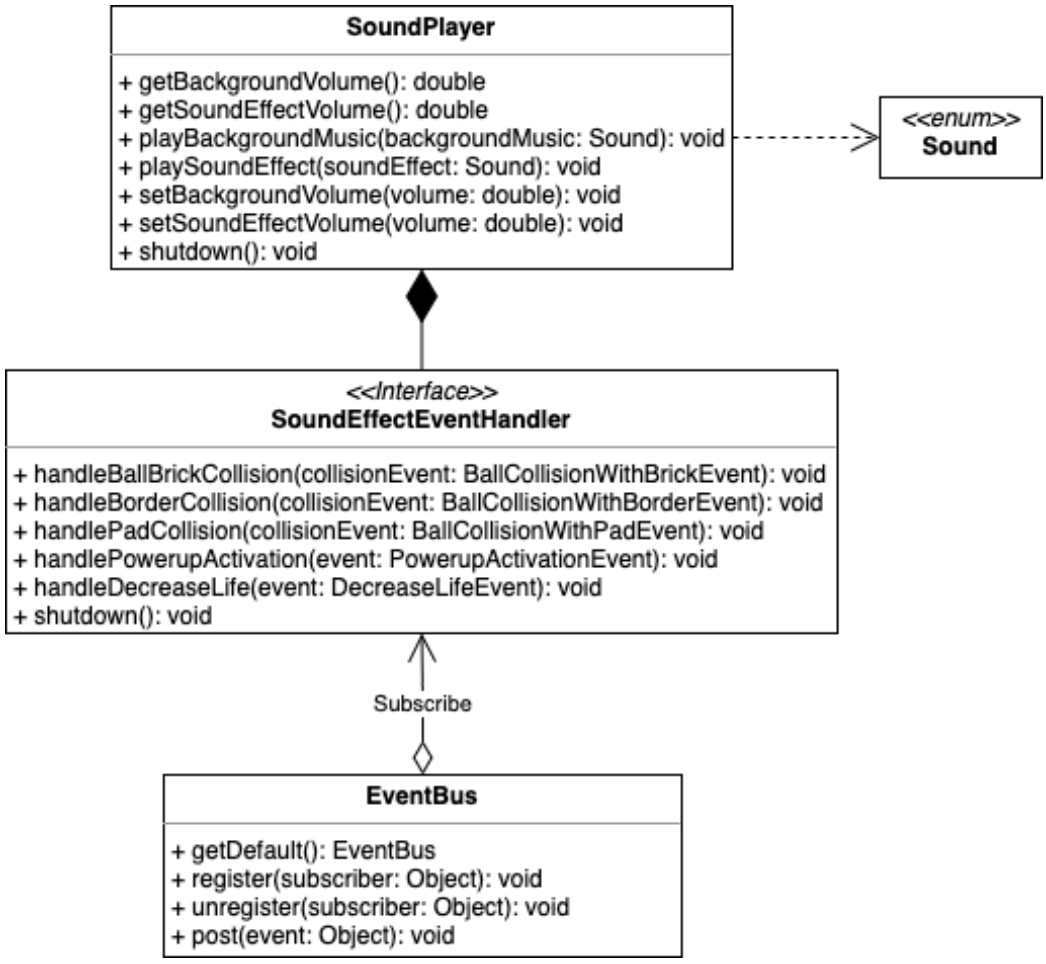

Figura 2.26: Schema strutturale UML di `SoundPlayer`

Capitolo 3

Sviluppo

3.1 Testing Automatizzato

A supporto dello sviluppo, nel corso del progetto, abbiamo adoperato la libreria JUnit per effettuare testing automatizzati. Lo scopo è stato quello di creare dei moduli di testing che non solo siano in grado di testare delle feature basilari, ma anche di controllare l'esito di alcune operazioni forzandone alcuni aspetti.

3.1.1 Giovanni Antonioni

Occupandomi principalmente della parte strutturale del dominio, la fase di testing ha coperto un ruolo fondamentale nel lavoro svolto. Ho creato le classi di testing del sistema ECS testando le funzionalità delle `Entity` e il meccanismo di attacco/stacco con `Components` di varia tipologia.
Dovendo poi interagire in maniera molto marcata con l'arena di gioco ho realizzato, assieme al supporto di Riccardo Traini, i vari test per il componente `Arena`. Infine, sono stati realizzati test basilari sia per i livelli che per il game state.

Moduli realizzati:

- `EntityTest`
- `ArenaTest`
- `ElementFactoryTest`
- `LevelTest`

- `LevelIteratorTest`
- `GameStateTest`

3.1.2 Luca Rubboli

La fase di testing, nonostante il mio ruolo in ambito Model sia più marginale rispetto ad altri componenti del gruppo, è risultata molto utile; l'implementazione di test efficaci infatti mi ha permesso di far emergere alcune lacune e comportamenti indesiderati soprattutto a livello di update e movimenti della `Ball`.

Moduli realizzati:

- `BallTest`
- `ElementTest`

3.1.3 Riccardo Traini

Per lo sviluppo della fisica di gioco ho ritenuto necessaria una fase di testing, che si è rilevata fondamentale. Ho creato la classe di testing per le Hitbox che mi ha permesso di rilevare tutti i comportamenti indesiderati, soprattutto dei casi limite delle collisioni tra BallHitbox e RectHitbox.

Moduli realizzati:

- `HitboxTest`
- `SoundPlayerVolumeTest`

3.2 Metodologia di lavoro

Come approccio di lavoro abbiamo sempre prediletto la formazione di schemi UML primordiali prima di passare all'implementazione del vero e proprio codice, analizzando criticamente eventuali problematiche che potevano scaturire dalla scelta di una determinata soluzione piuttosto che un'altra.
Questa metodologia di lavoro ci ha permesso di confrontarci spesso tra di noi, facendo emergere i punti più critici delle varie situazioni e instradando fin da subito una strategia risolutiva che ne prevedesse una effettiva soluzione, andando poi a raffinarla successivamente per far fronte ad eventuali problematiche di entità inferiore.

Fondamentale, al fine di seguire questo approccio, è stata una corretta suddivisione dei lavori, in quanto ciascun componente ha lavorato in modo che le proprie modifiche intaccassero il meno possibile quelle degli altri e, a tal proposito, un corretto utilizzo di Git è stato altrettanto centrale. Per quest'ultimo abbiamo adoperato una metodologia basata su Git Flow in cui il branch principale è "master". Ad ogni occorrenza di sviluppo di una nuova feature o di modifica di una parte di programma già esistente, abbiamo aperto un nuovo branch, al fine di apportare tali cambiamenti di software in maniera del tutto isolata e controllata. Una volta sicuri che tale implementazione non intaccasse la stabilità dell'applicazione si procedeva alla fase di merge in master.
Talvolta sono occorse situazioni in cui, al fine di delineare una soluzione efficace e ben strutturata, è stato necessario un confronto di gruppo ed una ricerca approfondita dell'argomento tramite la rete.

3.2.1 Giovanni Antonioni

Realizzazione sistema Entity Component System per la gestione degli elementi del dominio e sistema di intercomunicazione basato su eventi. Gestione dei livelli del gioco e composizione di questi attraverso files di configurazione, per facilitare lo switch tra `Levels` ho realizzato anche un iteratore per questi. Realizzazione del GameState e del motore di gioco (`GameLoop`).

3.2.2 Luca Rubboli

Implementazione `AbstractElement` e `Ball`, per cui ho predisposto un Builder. Design delle sprites e di alcune View del gioco. Sviluppo di `Linker` con annessa gestione dell'input utente e gestione del caricamento delle varie View.

3.2.3 Riccardo Traini

Implementazione di `Arena` e di `Hitbox`, collegate direttamente agli Element. Design dei livelli, i loro sfondi e di alcune View del gioco. Realizzazione sistema `GameView-Drawer-Binder` per il mantenimento dell'aspect ratio durante la stampa dell'`Arena`. Implementazione del `SoundPlayer` e della regolazione del volume nella SettingsView.

3.3 Note di sviluppo

3.3.1 Giovanni Antonioni

Feature avanzate del linguaggio e librerie che ho adoperato:

- **Utilizzo di generici**
- **Lambda expressions**
- **Stream**
- **Programmazione funzionale**
- **Optional**
- **Utilizzo di ThreadPoolExecutor**: E' stato adoperato, ed esteso, per la realizzazione del sistema di handling dei thread relativi ai power-up.
- **Libreria esterna "Snake YAML"** : Permette di leggere dei files di configurazione in formato yaml ed estrarne il loro contenuto. Utilizzata per la costruzione di livelli da files.
- **Libreria esterna "EventBus"** : Utilizzata per la gestione degli eventi. Essendo questa una feature chiave della nostra applicazione mi sono voluto basare su una libreria affidabile e stabile.
- **Build system Gradle**: Adoperato per la fase di compilazione dell'applicazione e per la generazione del file `.jar`.

Crediti

Nella realizzazione del motore grafico ho preso spunto dal codice mostrato dal professor **Ricci** adoperandolo come base.

3.3.2 Luca Rubboli

Feature avanzate del linguaggio e librerie che ho adoperato:

- **Utilizzo di generici**
- **Lambda expressions**
- **Programmazione funzionale**
- **Optional**
- **Libreria grafica JavaFX**

Crediti

Per la realizzazione delle differenti viste ho preso vari spunti da ricerche su Internet e da piccoli corsi che mi hanno permesso di consolidare un'infarinatura generale nel mondo di JavaFX.

3.3.3 Riccardo Traini

Feature avanzate del linguaggio e librerie che ho adoperato:

- **Utilizzo di generici**
- **Lambda expressions**
- **Programmazione funzionale**
- **Optional**
- **Stream**
- **Libreria grafica JavaFX**

Crediti

Per la realizzazione delle View e del SoundPlayer ho fatto varie ricerche su Internet consultando diversi corsi, in modo da poter utilizzare senza troppi problemi la libreria JavaFX. Per le collisioni ho preso spunto dal codice mostrato dal professor Ricci e da http://www.jeffreythompson.org/collision-detection/circle-rect.php.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Giovanni Antonioni

Sicuramente la realizzazione di questo progetto è stata altamente stimolante per svariati motivi. Anzitutto mi ha permesso di affrontare in maniera approfondita le varie fasi di sviluppo di un applicativo complesso, invogliandomi a ricercare una soluzione ottimale ad un problema analizzandone le varie implicazioni. Un altro aspetto importante è stato poi il lavorare internamente ad un gruppo di persone che mi ha permesso di vedere quali tipologie di criticità emergano nel suddividere nella maniera più ottimale possibile i vari compiti e nel rispettare delle sotto scadenze.
Altro aspetto importante è stato quello di solidificare una conoscenza base di `Git`, la quale ci ha permesso agilmente di sovrapporre le nostre implementazioni e modifiche durante questi mesi di sviluppo.
Internamente al gruppo mi sono preso carico di quello che è il realizzare un aspetto architetturale efficiente sin dalle prime ore di analisi, cercando di delineare un ordine nell'organizzazione del codice e nella strutturazione del progetto stesso.

4.1.2 Luca Rubboli

Questo progetto ha dato la possibilità di mettermi in gioco totalmente, sia a livello di singolo, sia in concomitanza con altre persone, offrendo una visione, seppur limitata, a tuttotondo di ciò che potrà poi prevedere un eventuale futuro lavoro in questo ambito. Ho apprezzato la possibilità di poter approfondire diverse situazioni e criticità in maniera autonoma, e di arricchire la mia conoscenza anche con il contributo degli altri componenti del gruppo.

Questo percorso ha permesso inoltre di avere un'idea ben più concreta di strumenti molto potenti di cui inizialmente avevo una conoscenza superficiale, `Git` soprattutto. Di fondamentale importanza è stata una metodologia di lavoro efficace, basata sulla costruzione di schemi e discussioni prima della stesura effettiva del codice.

4.1.3 Riccardo Traini

La realizzazione di questo progetto è interessante sotto diversi punti di vista. Prima di tutto ho potuto provare l'esperienza di un progetto di gruppo, dove le discussioni e la realizzazione di schemi, così come la suddivisione dei compiti scanditi da date di scadenza, si sono rilevate fasi fondamentali. Ho potuto approfondire inoltre le varie fasi di sviluppo di un progetto, ricercando una soluzione ottimale ad ogni problema che emerge. Parte importantissima dello sviluppo del progetto è stato l'utilizzo di `Git`, uno strumento molto potente che ci ha consentito uno sviluppo fluido e una comunicazione efficace. Ho anche apprezzato la possibilità di arricchire le mie conoscenze con l'ausilio degli altri componenti del gruppo. Questo progetto mi ha quindi offerto una visione generale di come potrebbe essere un futuro lavoro in questo ambito.

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Giovanni Antonioni

Poco da aggiungere a quanto già scritto nella valutazione del corso effettuata lo scorso semestre. Questo corso è stato ottimo sotto molteplici punti di vista poichè oltre ad affrontare con una chiarezza totale degli argomenti complessi riguardo, non solo gli aspetti del linguaggio Java ma bensì della programmazione ad oggetti in generale, mi ha permesso di sperimentare tramite laboratorio queste conoscenze acquisite. Ritengo che tutto, dalle lezioni alle prove di valutazione, sia stato strutturato al meglio e che abbia contribuito a solidificare delle conoscenze certamente utili in ambito lavorativo.

4.2.2 Luca Rubboli

Personalmente ho trovato questo corso molto ben strutturato, in grado di gettare solide basi nell'ambito della programmazione ad oggetti. Ho apprezzato molto anche la parte di laboratorio, sebbene mi sia trovato un po' in

difficoltà nell'approcciare una grande vastità di argomenti. Come piccolo appunto evidenzierei la difficoltà che ho riscontrato nell'interfacciarmi le prime volte con gli Stream, che rappresentano un modo di concepire soluzioni di problemi differente rispetto a quelli adottati precedentemente. Per arrivare alla perfezione, a cui questo corso sicuramente ambisce, credo che debba essere introdotto materiale leggermente più dettagliato sugli Stream appunto, per il resto ritengo che gli argomenti siano stati esplicati in maniera ineccepibile, mantenendo sempre un approccio orientato anche al mondo del lavoro. Nel corso dello sviluppo del progetto è stato stimolante, seppur con qualche leggera difficoltà, approfondire l'ambito grafico.

4.2.3 Riccardo Traini

Il corso è stato eccellente, gli argomenti sono stati trattati in modo esemplare e con grande chiarezza, ma soprattutto in modo scorrevole, nonostante la quantità di argomenti trattati e approfonditi sulla programmazione ad oggetti. Ho trovato anche molto interessanti gli approfondimenti sulle interfacce grafiche e le lezioni del Professor Ricci. Anche le lezioni di laboratorio sono state ben eseguite, però sono diventate abbastanza complicate con l'imposizione della didattica a distanza, soprattutto per la mancanza di registrazioni. Pecca a parte, il corso in generale è stato gestito ottimamente, e ha sempre mantenuto un approccio rivolto anche al mondo del lavoro.

Appendice A

Guida utente

A.1 Avvio dell'applicativo

Per avviare l'applicazione occorre aver installato all'interno della propria macchina una versione di Java 11 o più recente, che comprenda anche il modulo della libreria JavaFX.

- Posizionarsi all'interno della cartella contenente il jar relativo all'applicazione (solitamente chiamato `Pyxis-all.jar`).
- Eseguirlo tramite il comando `java -jar JarPackage.jar`

A.2 Menù di gioco

All'avvio dell'applicazione, la GUI presenterà il menù di avvio, nella quale il giocatore potrà scegliere, cliccando sull'apposito bottone, se

- `New Game`: Avviare una nuova partita dal primo livello.
- `Settings`: Mostrare il menù di impostazioni dell'applicazione.
- `Levels`: Mostrare il menù di selezione dei livelli da giocare.
- `Quit`: Chiudere l'applicazione.

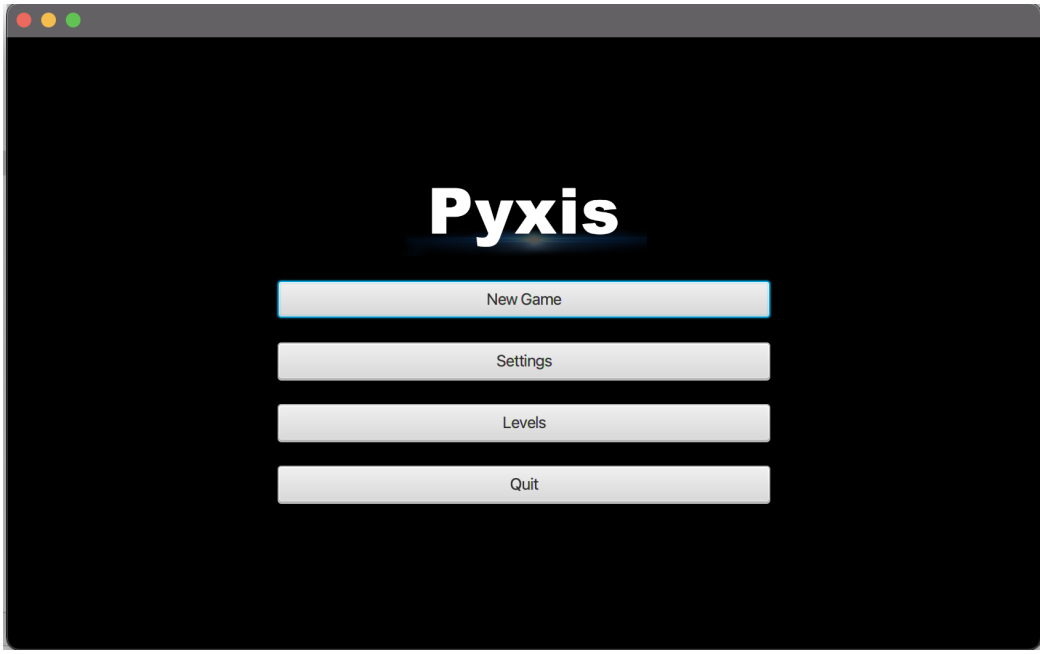

Figura A.1: Menù dell'applicazione

A.3 Schermata di gioco

All'avvio di una nuova partita si potra notare il livello caricato con al suo interno i mattoncini da distruggere, il pad e la palla. Per avviare la partita premere `SPACE`, la palla inizierà a muoversi verso la parte superiore dell'arena. Tramite i tasti `A` e `D` è possibile controllare il pad muovendolo verso sinistra e destra. In qualsiasi momento è possibile bloccare la partita in uno stato di pausa cliccando il tasto `esc`, l'applicativo mostrerà il menù di pausa da cui si potrà riprendere la partita o modificare alcune impostazioni, oppure terminare la partita. Qualora si decida di riprendere la partita, sarà necessario notificare nuovamente la ripresa con la pressione del tasto `SPACE`.

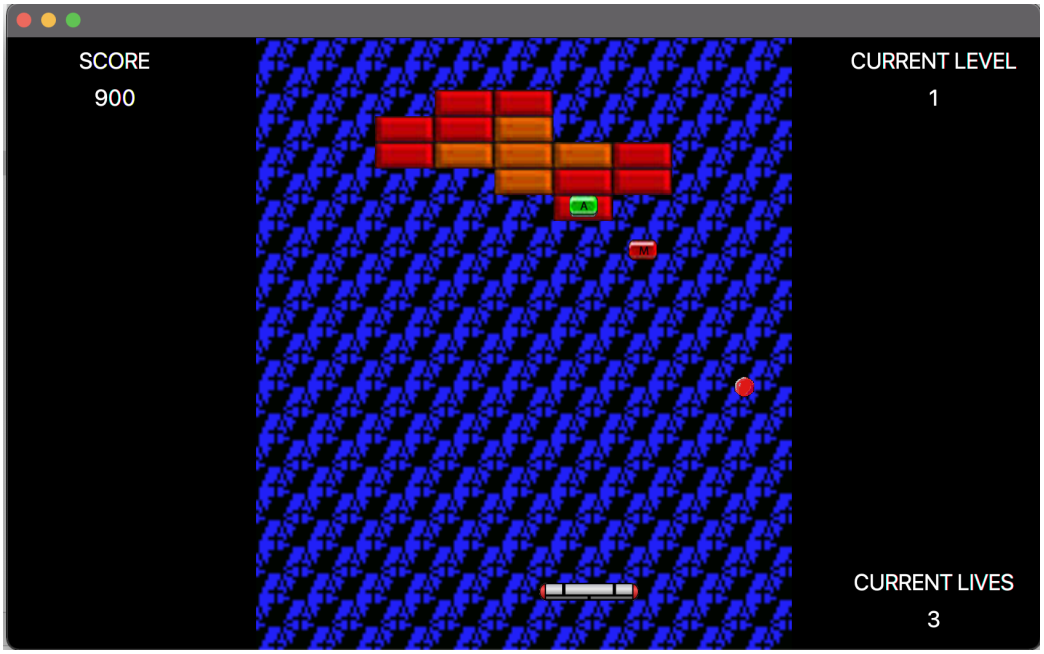

Figura A.2: Schermata di gioco dell'applicazione

Appendice B

Esercitazioni di laboratorio

B.1 Giovanni Antonioni

- Laboratorio 04: https://virtuale.unibo.it/mod/forum/discuss.php?d=62685#p127690
- Laboratorio 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=62684#p101487
- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=62579#p100967
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=62582#p101467
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=63865#p103271
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=64639#p104010
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=66753#p111944
- Laboratorio 11: https://virtuale.unibo.it/mod/forum/discuss.php?d=66463#p106888

B.2 Luca Rubboli

- Laboratorio 04: Non svolto.

- Laboratorio 05: Non svolto.
- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=62579#p101503
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=62582#p101060
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=63865#p104233
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=64639#p104235
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=66753#p106691
- Laboratorio 11: Non svolto.

B.3 Riccardo Traini

- Laboratorio 04: Non svolto.
- Laboratorio 05: Non svolto.
- Laboratorio 06: Non svolto.
- Laboratorio 07: Non svolto.
- Laboratorio 08: Non svolto.
- Laboratorio 09: Non svolto.
- Laboratorio 10: Non svolto.
- Laboratorio 11: Non svolto.