

# **Sonic the Hedgehog**

Relazione per il progetto di *Programmazione  
ad Oggetti*  
A.A. 2024/25

Samuele Bizzocchi

*samuele.bizzocchi3@studio.unibo.it*

Francesco Giancaterino

*frances.giancaterin2@studio.unibo.it*

Filippo Merlini

*filippo.merlini2@studio.unibo.it*

Nicolas Mancini

*nicolas.mancini@studio.unibo.it*

31 agosto 2025

# Indice

<b>1 Analisi</b>	<b>2</b>
1.1 Descrizione e requisiti . . . . .	2
1.2 Modello del Dominio . . . . .	3
<b>2 Design</b>	<b>5</b>
2.1 Architettura . . . . .	5
2.2 Design dettagliato . . . . .	9
2.2.1 Sezione Bizzocchi: Fisica e movimento . . . . .	9
2.2.2 Sezione Giancaterino: Rendering e Core del Game . . .	12
2.2.3 Sezione Merlini: Game Engine ed eventi . . . . .	21
2.2.4 Sezione Mancini: Animazioni . . . . .	24
<b>3 Sviluppo</b>	<b>27</b>
3.1 Testing automatizzato . . . . .	27
3.2 Note di sviluppo . . . . .	28
3.2.1 Samuele Bizzocchi . . . . .	28
3.2.2 Francesco Giancaterino . . . . .	30
3.2.3 Filippo Merlini . . . . .	31
3.2.4 Nicolas Mancini . . . . .	32
<b>4 Commenti finali</b>	<b>36</b>
4.1 Autovalutazione e lavori futuri . . . . .	36
4.1.1 Samuele Bizzocchi . . . . .	36
4.1.2 Francesco Giancaterino . . . . .	37
4.1.3 Filippo Merlini . . . . .	37
4.1.4 Nicolas Mancini . . . . .	38
4.2 Difficoltà incontrate e commenti per i docenti . . . . .	38
<b>A Guida utente</b>	<b>40</b>

# Capitolo 1

## Analisi

### 1.1 Descrizione e requisiti

Il software consiste in una versione semplificata del gioco *Sonic The Hedgehog* prodotto da Sega e rilasciato nel 1991. Il gioco prevede quindi un personaggio giocante che si sposta in una mappa 2D, con l'obiettivo di raccogliere più anelli possibili e raggiungere il traguardo nel minor tempo possibile. La semplificazione sta nel minor numero di tipologie di nemici, mancanza di molti tratti caratteristici degli ambienti, mancanza dell'iconico spindash di Sonic, e nel minor realismo dei movimenti di Sonic. Inoltre, a differenza del gioco originale, questa versione avrà un singolo livello giocabile. Comunque saranno presenti sia ostacoli fissi sia nemici che si muoveranno e tenteranno di colpire il giocatore, per fargli perdere gli anelli recuperati fino a quel punto. La partita avrà termine una volta raggiunto il traguardo oppure al game over, causato dall'essere colpiti senza possedere nemmeno un anello.

#### Requisiti funzionali

- Il giocatore dovrà essere in grado di regolare la sua velocità di movimento in base al tempo di pressione dei comandi di movimento.
- Il giocatore dovrà poter raccogliere degli Anelli nel corso del livello, ovvero una valuta che lo protegge da un colpo, salvandolo dal gameOver se ne possiede almeno uno.
- Il giocatore dovrà essere in grado di saltare e collidere correttamente con tutti i blocchi fisici.
- Ci dovranno essere nemici e ostacoli per il giocatore da evitare. In caso di collisione il giocatore dovrà perdere tutti gli anelli posseduti.

- Un inizio e una fine del livello ben definiti.
- Possibilità di mettere in pausa.

## Requisiti non funzionali

- Menu iniziale.

## 1.2 Modello del Dominio

Il gioco si baserà sulla presenza di una mappa (un livello) nel quale il giocatore potrà muoversi liberamente, correndo e saltando a piacere, ma mantenendo sempre un singolo obiettivo: raggiungere il traguardo all'estrema destra della mappa. Tuttavia, sulla sua strada ci saranno delle entità ostili che cercheranno di eliminarlo. Sonic potrà distruggerle saltandoci sopra, ma toccarle in altro modo gli causerà il game over, a meno che non possegga degli anelli. Ecco perché l'obiettivo secondario del giocatore sarà raccogliere anelli mentre si muove verso il traguardo, così da essere protetto nel caso venga colpito. Le Entità ostili, detti semplicemente “nemici”, dovranno essere capaci di muoversi per inseguire il giocatore, e di infliggergli danno al contatto. Gli anelli dovranno dapprima essere fissi e fluttuanti all'inizio della partita, ma poi dovranno fuoriuscire dal corpo di Sonic quando viene colpito, come se gli cadessero di dosso, e allontanarsi da lui tutti assieme, fino ad una certa distanza. In tutto questo, porremo grande attenzione all'aspetto grafico di tutto il gioco.

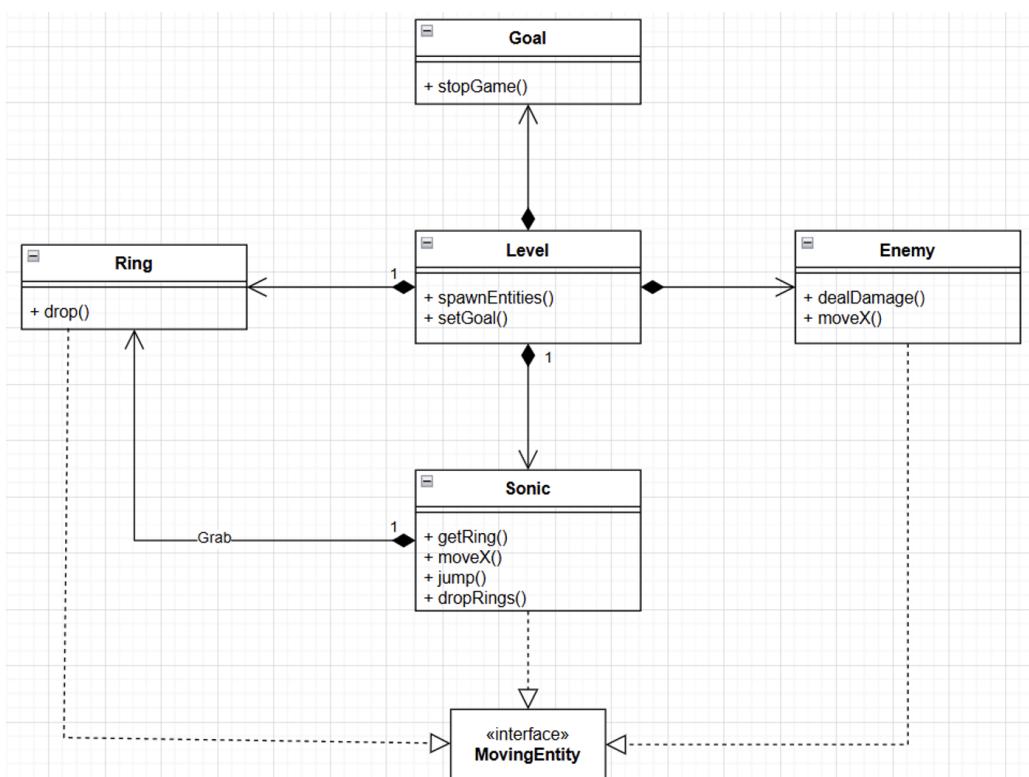


Figura 1.1: Diagramma UML delle entità della partita

# Capitolo 2

## Design

### 2.1 Architettura

L’architettura si basa sul Component pattern orientato ad eventi. Ogni Entità, ovvero ciascun game object che nel gioco svolge o subisce azioni/modifiche, ha una lista di elementi, detti Componenti, che vengono aggiornati ad ogni iterazione del game loop e/o conservano dati e comportamenti aggiuntivi. Questo pattern è stato proposto da Merlini, per via della sua forte scalabilità, e perché avrebbe semplificato la gestione delle entità, aspetto che ci stava mettendo in difficoltà sin dalla fase di progettazione. Infatti, ad ogni entità del gioco si possono aggiungere solo i componenti di cui necessita, e nulla di più, senza bisogno di creare una quantità esagerata di sottoclassi.

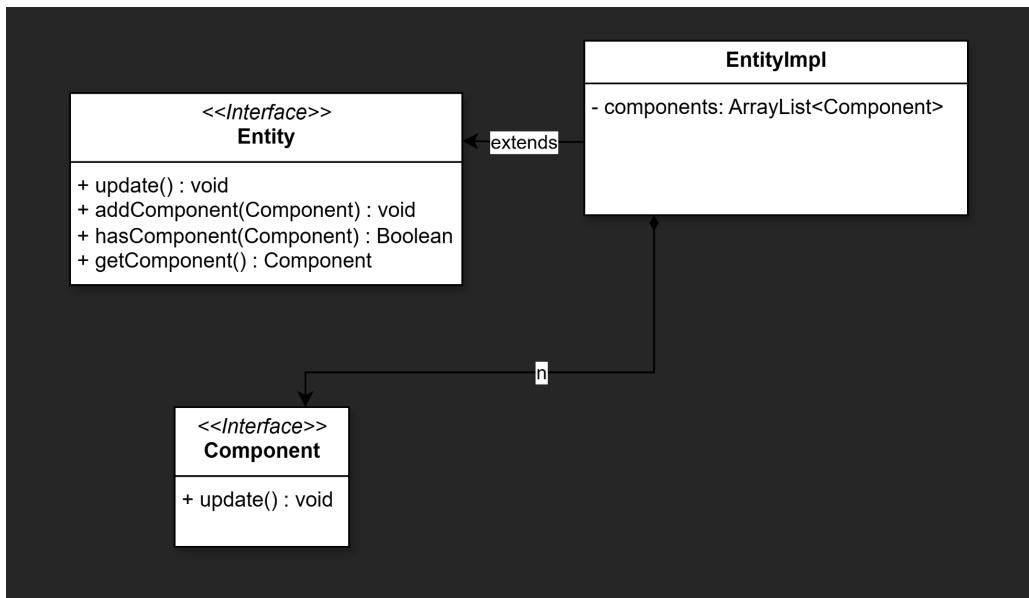


Figura 2.1: Diagramma UML dell'architettura principale

Gestendo le entità con un **Entity Manager**, aggiornare le entità significa aggiornare i suoi componenti, uno ad uno.

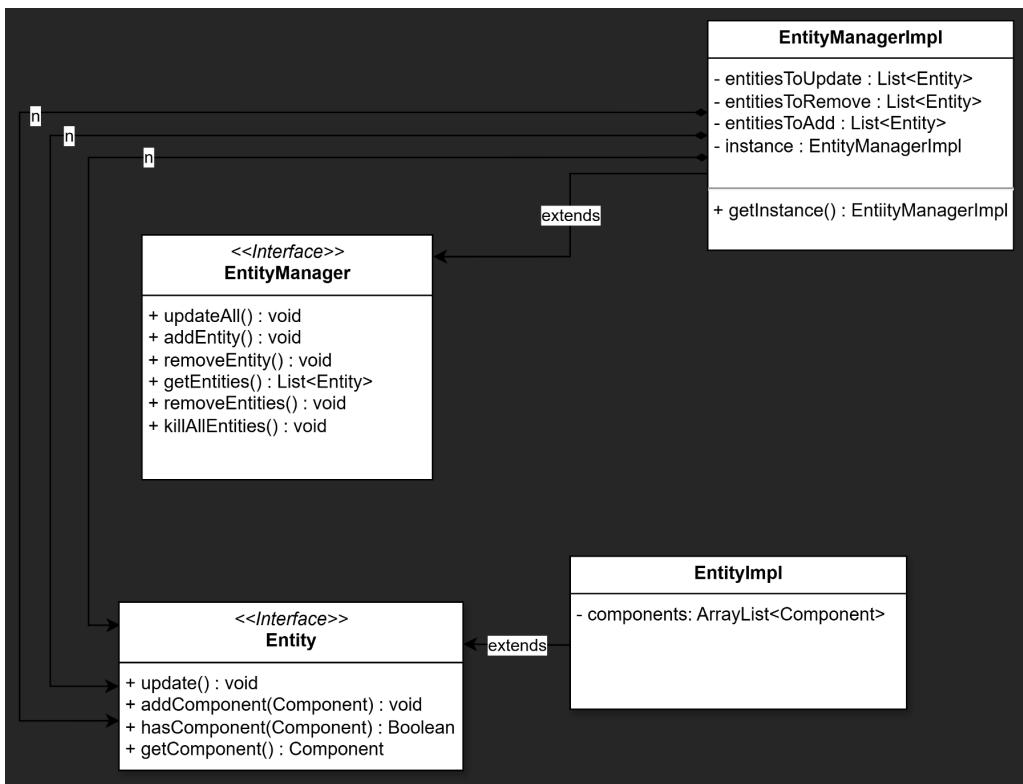


Figura 2.2: Diagramma UML dell'architettura del Entity Manager

Gli eventi sono gestiti dal pattern Subjects/Observers, gli oggetti che provocano modifiche o compiono azioni sono i Subject che notificano una serie di Observer. Gli Observer, a loro volta, si mettono in ascolto dei primi nel caso vengano notificati con eventi contenente le modifiche. Si è adoperato questo pattern per disaccoppiare gli oggetti tra di loro, facendo sì che possano comunicare senza essere in conoscenza dell'uno con l'altro. Ci sarà una sezione di classi dedicata al Rendering di tutto ciò che dev'essere visibile al giocatore. Esse interrogheranno altri componenti per interpretare ciò che sta accadendo a schermo, così da decidere quali sprite utilizzare. Infine, un'altra porzione di classi farà parte del core, gestendo il Gameloop.

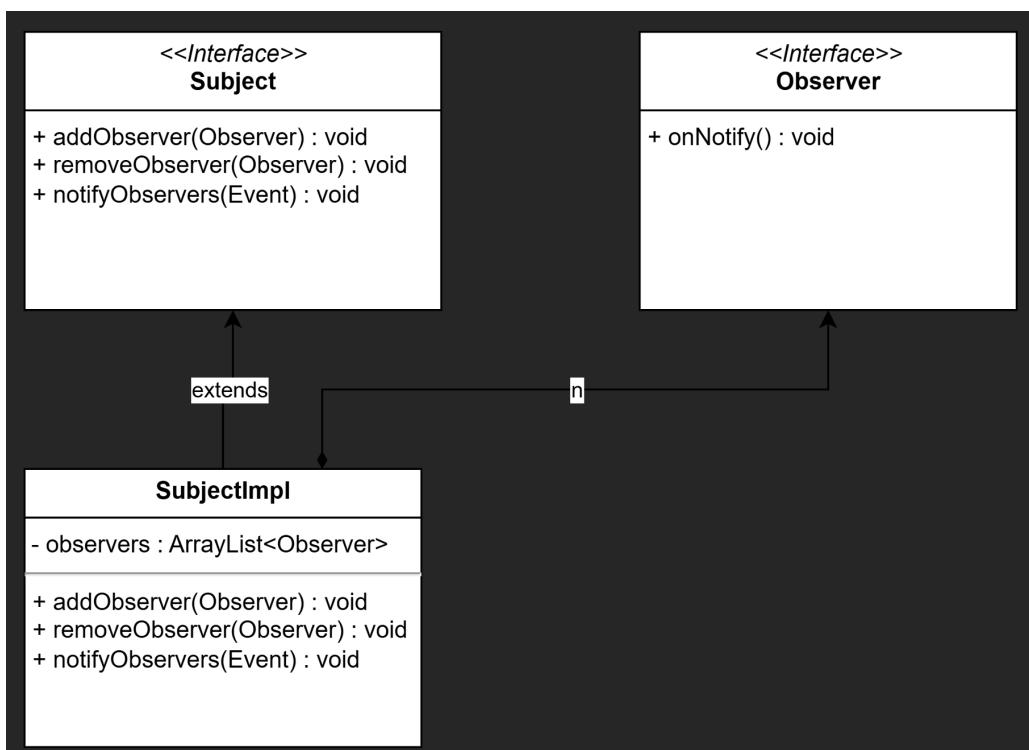


Figura 2.3: Diagramma UML dell’architettura degli eventi

## 2.2 Design dettagliato

### Nota generale

È da notare che non tutto nel gioco è un'entità. I blocchi che compongono il livello, le Tiles, non hanno bisogno di essere aggiornate, ergo non gli serve alcun componente. Bizzocchi, Mancini e Merlini hanno tutti creato almeno un'implementazione dei componenti. Siccome sapevamo che avremmo utilizzato ognuno i componenti in maniera fortemente differente, ci siamo ispirati al Template Method, lasciando il metodo per l'aggiornamento del Component astratto, per avere la massima possibile versatilità dell'interfaccia. La classe più importante è sicuramente Game.java, che contiene istanze di GamePanel e GameState, altre due classi di fondamentale importanza. Il primo gestisce il rendering e la parte grafica, il secondo invece permette di gestire i vari stati del gioco (menu iniziale, pausa, attivo, giocando).

#### 2.2.1 Sezione Bizzocchi: Fisica e movimento

##### Collisioni con l'ambiente

**Problema:** Le Tiles che compongono il livello devono poter fermare o comunque influenzare il movimento delle entità come dei veri oggetti solidi.

**Soluzione:** Nella superclasse per la fisica ho definito il metodo necessario a rilevare le collisioni con le Tiles in base ad un possibile movimento. Invece le collisioni fra Entity sono definite nelle sottoclassi di Physics. L'effetto di tale collisione viene stabilito da ciascuna sottoclasse. Questo mi ha permesso di creare una risposta alle collisioni più variegata e realistica. Ad esempio, Sonic e nemici si fermano quando vanno a sbattere contro un muro, invece gli anelli rimbalzano.

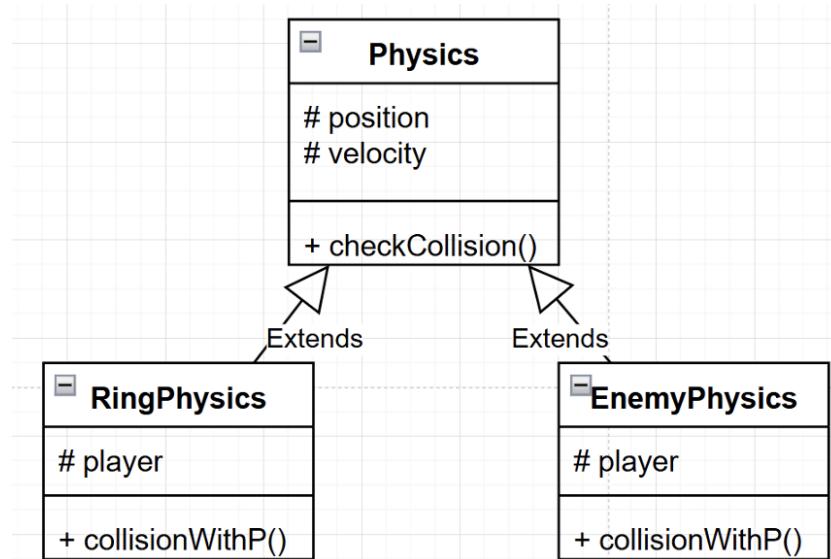


Figura 2.4: Rappresentazione UML dell’organizzazione della Fisica degli elementi di gioco

Pattern: Nessuno.

### Classe PlayerPhysics

**Problema:** La classe PlayerPhysics è fin troppo carica di metodi e attributi, in quanto deve essere capace di interagire con tutti gli elementi del gioco, dal primo all’ultimo.

**Soluzione:** Migliorare l’ereditarietà della mia porzione di progetto, definendo più metodi possibili nella superclasse per la fisica: `die()`, `canGoThere()`, `checkIntersections()` e perfino `landing()`, anche se non è utilizzato dagli anelli. In più delegare la gestione degli anelli ad un nuovo componente (`WalletComponent`). Questa soluzione mi ha permesso di sfruttare appieno l’Entity-Component-System. Ho ricevuto anche la collaborazione di Merlini per completare questa classe.

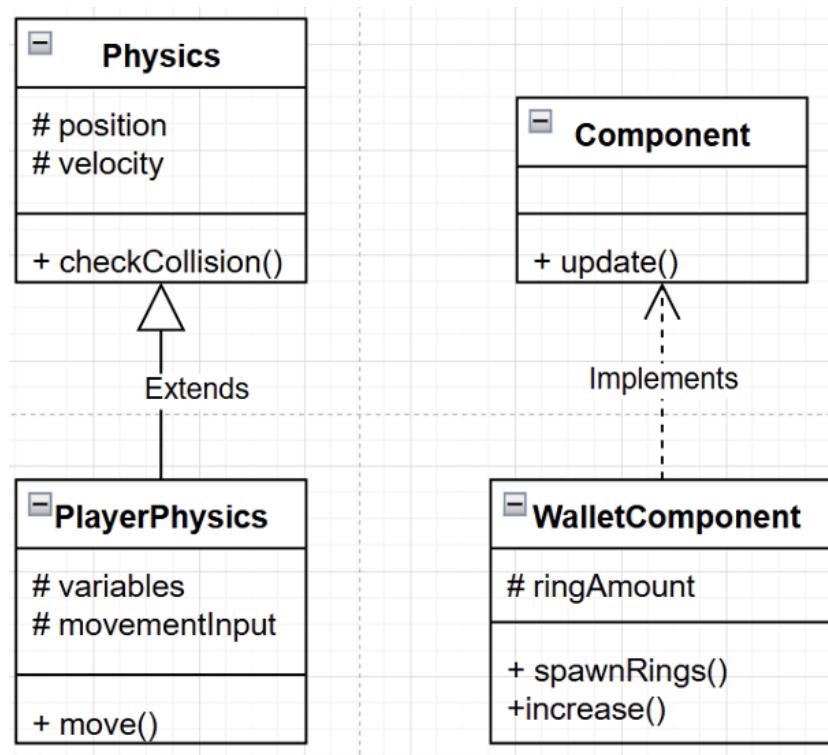


Figura 2.5: Rappresentazione UML dell'organizzazione del WalletComponent

Pattern: Entity-Component-System.

## 2.2.2 Sezione Giancaterino: Rendering e Core del Game Game Loop

**Problema:** Gestire correttamente il tempo e sincronizzare la logica di gioco (update) e il rendering (repaint) per garantire un’esperienza fluida e costante, indipendentemente dalla velocità del computer. Inoltre, andavano gestiti gli input e la corretta chiusura del thread di gioco.

**Soluzione:** Inizialmente avevo pensato un game loop senza separare la logica di aggiornamento del gioco e il rendering ma presto mi sono reso conto che questa soluzione non era sostenibile in quanto non rendeva il gioco fluido e scalabile. Quindi sono passato ad implementare un Fixed Timestep Game Loop nel’ omonima classe; in questo modo ho separato la logica di aggiornamento (120 UPS) e rendering (60 FPS) utilizzando accumulatori di tempo. Questo consente di mantenere la coerenza del gameplay, recuperare eventuali frame persi eseguendo più aggiornamenti consecutivi se necessario ed assicurare prestazioni stabili. L’implementazione di Runnable consente al GameLoop di essere eseguito su un thread separato, avviato mediante startLoop(). In questo modo il ciclo di gioco non interferisce con l’interfaccia grafica, assicurando reattività e continuità di aggiornamento.

Le responsabilità principali sono distribuite tra:

- GameStateManager: gestisce la logica e le transizioni tra stati di gioco.
- GamePanel: rappresenta la superficie principale di disegno, delegando la gestione degli input a classi specializzate.<sup>1</sup>
- Game: punto di ingresso dell’applicazione; gestisce l’avvio e l’arresto del gioco, utilizzando un reference method (`this::stop`) come callback.

Questo approccio garantisce una chiara separazione delle responsabilità, migliora la leggibilità del codice e assicura modularità ed estendibilità del sistema.

---

<sup>1</sup>La gestione degli input avviene come segue:

- Al GameStateManager: per azioni che cambiano lo stato del gioco (come passare al menu di pausa o uscire dal gioco).
- All’InputManager: per azioni relative al gameplay (come muovere il personaggio).

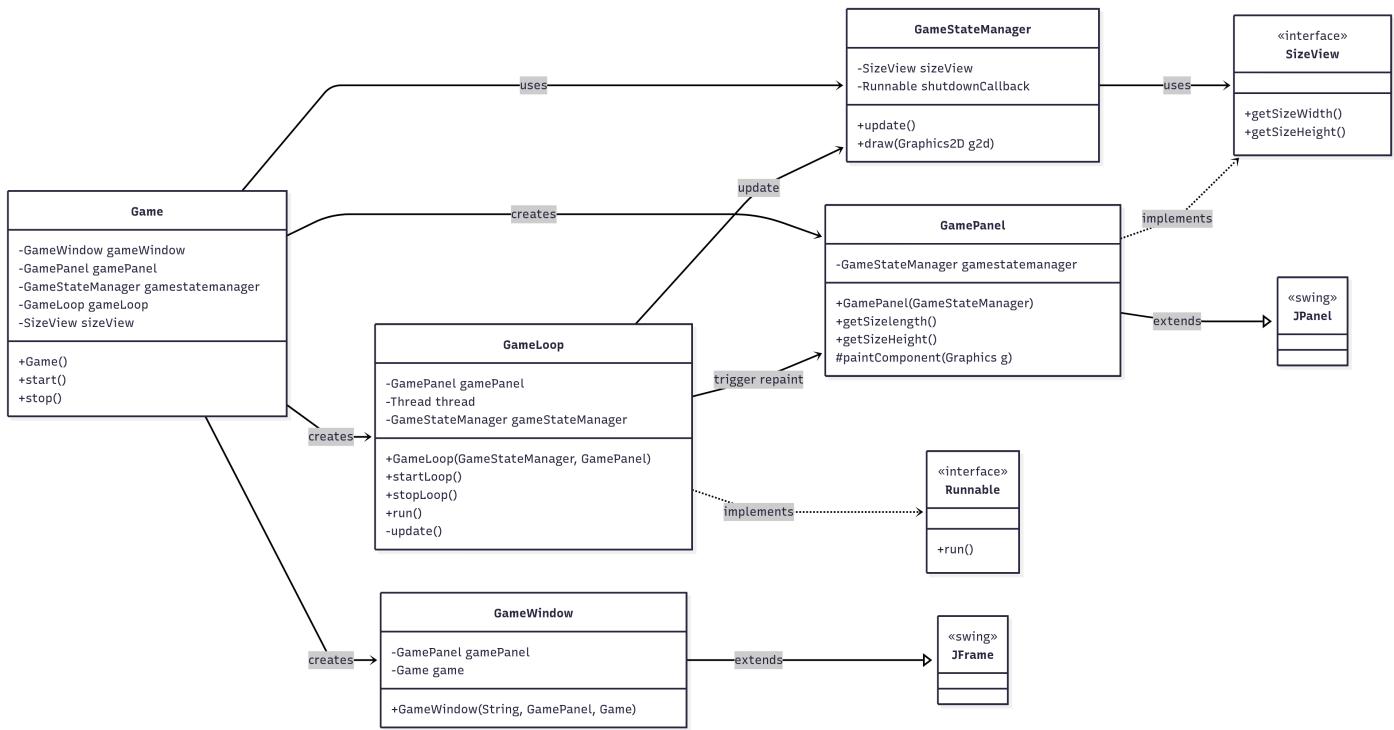


Figura 2.6: Rappresentazione UML dell’organizzazione semplificata del core del gioco

Pattern: Nessuno.

### Gestione degli stati del gioco

**Problema:** Il gioco deve gestire stati distinti (Menu, Playing, Paused) ciascuno con proprie logiche di input, aggiornamento e rendering. Era necessario un meccanismo che permettesse transizioni fluide senza creare dipendenze cicliche.

**Soluzione:** Ho deciso di adottare lo State Pattern come segue:

- Interfaccia astratta `GameState` con i metodi comuni `update()`, `draw()` e gestione input.
- Ogni stato è una classe concreta che implementa l’interfaccia.

- Il `GameStateManager` funge da *context*, imposta lo stato corrente e delega a esso l'aggiornamento, il rendering e input.

Il `GameStateManager` è implementato come **Singleton**, garantendo un'unica istanza. Nello specifico si occupa di:

- Delegare `update()` e `draw()` allo stato attualmente attivo.
- Gestire le transizioni da uno stato all'altro sia attraverso il trigger degli eventi sia attraverso gli input.
- Ricevere gli input dal `GamePanel` e inoltrarli allo stato corrente.

Questo design disaccoppia la logica di ogni stato, rendendo l'aggiunta di nuovi stati un'operazione semplice, senza dover modificare il resto del codice.

Per la gestione del flusso degli eventi è stato adottato l'**Observer Pattern**. Ogni `GameState` estende `SubjectImpl`, acquisendo così la capacità di notificare eventi ai propri osservatori. In questo modo, ad esempio, il `PlayingState` può generare un evento di *game over*, a cui il `GameStateManager` è registrato come observer: il manager riceve la notifica e cambia lo stato del gioco in modo reattivo e disaccoppiato.

L'integrazione delle classi `GS` e `GSM` con `SubjectImpl` e `Subject` è stata sviluppata in collaborazione con Merlini, che ha contribuito a rendere il `GSM` pienamente funzionale.

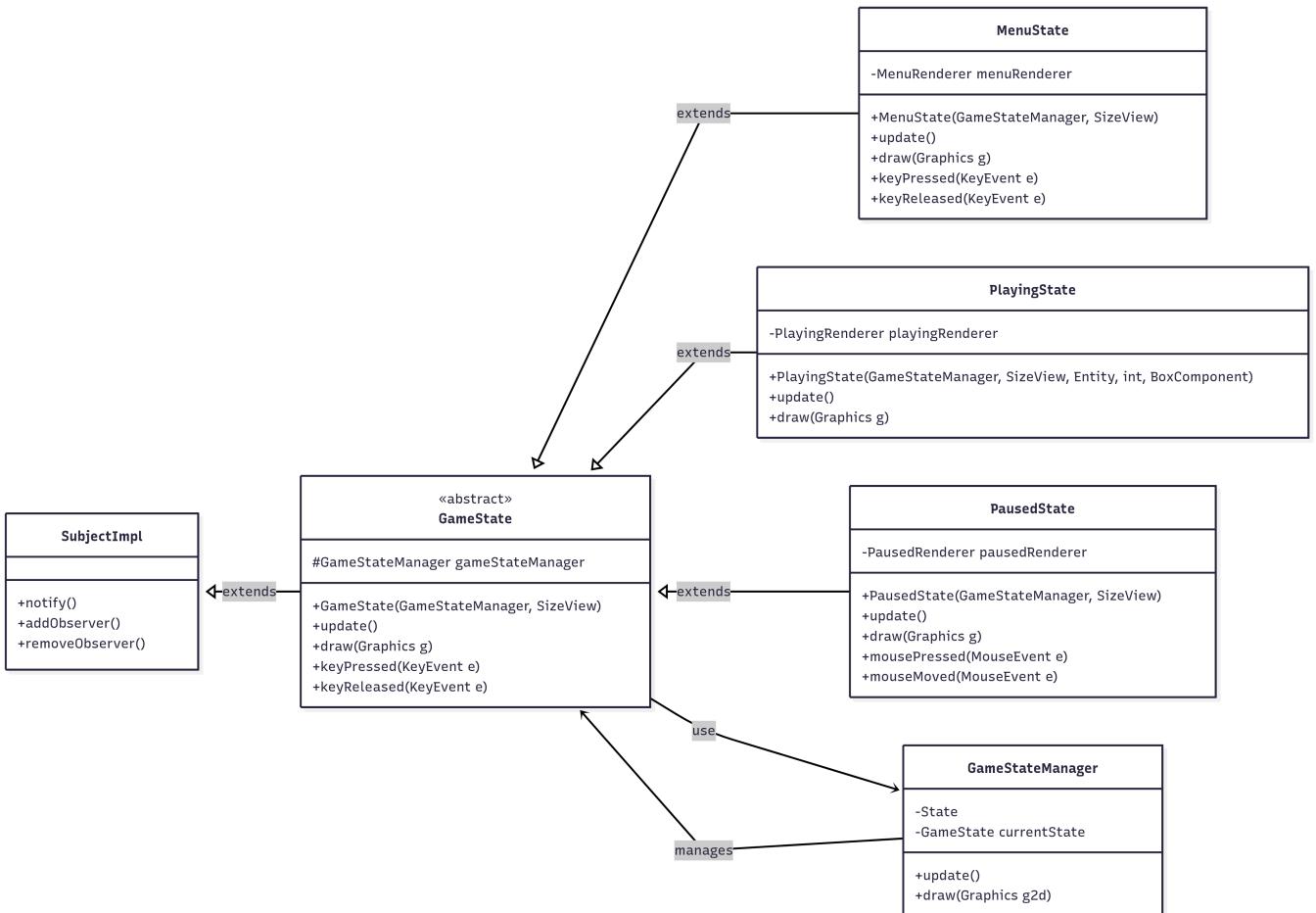


Figura 2.7: Rappresentazione UML dell’organizzazione del GameState e GameStatemanager<sup>2</sup>

Pattern: State Pattern.

---

<sup>2</sup>L’etichetta **manages** è utilizzata per indicare che il **GameStateManager** ha il compito di controllare il ciclo di vita e le transizioni degli stati del gioco. Essa chiarisce la relazione tra le due classi secondo il design pattern **State**, dove il manager agisce come orchestratore.

## Rendering degli stati

**Problema:** Ogni stato del gioco (Menu, Playing, Paused) richiede logiche di rendering completamente diverse quindi era necessario un sistema flessibile che permettesse di cambiare il comportamento del rendering dinamicamente.

**Soluzione:** Ho adottato lo **Strategy Pattern** attraverso l’interfaccia **Renderer** (*Strategy Interface*) e le sue implementazioni nei vari stati (*Concrete Strategy*). Ogni renderer (*Context*) incapsula la logica specifica del proprio dominio:

- **MenuRenderer:** gestisce animazioni, gradienti, effetti animati del logo;
- **PlayingRenderer:** renderizza il mondo di gioco, entità, camera, HUD;
- **PausedRenderer:** mostra una schermata con pulsante interattivo.

Attraverso questa architettura si possono aggiungere nuovi renderer senza modificare il codice esistente, rendendo quindi il gioco scalabile. Lo sviluppo di alcuni metodi relativi alle animazioni e al rendering delle entità di gioco nel **PlayingRenderer** è stato realizzato in collaborazione con Mancini, che ha contribuito a rendere questo componente pienamente funzionale.

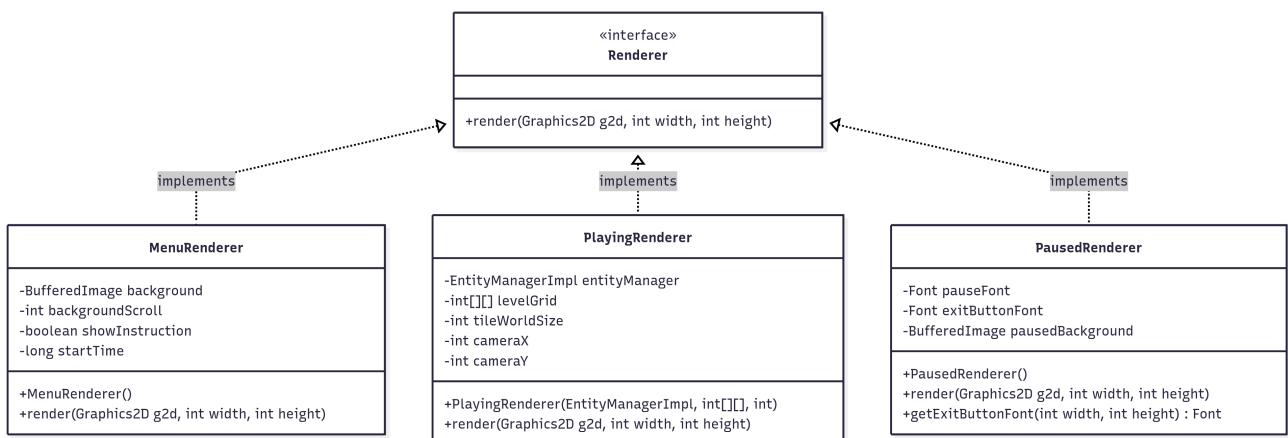


Figura 2.8: Rappresentazione UML dell’organizzazione del Renderer

Pattern: Strategy Pattern.

## Design Responsive

**Problema:** Il sistema di rendering doveva adattarsi dinamicamente a diverse risoluzioni e dimensioni della finestra, mantenendo proporzioni corrette per font e per gli sfondi. Era necessario un sistema che fosse indipendente dalle dimensioni specifiche del pannello.

**Soluzione:** La soluzione riguarda gli stati di gioco `Menu` e `Paused`, in quanto per lo stato di `Playing` il design responsivo non è stato applicato. Nello specifico, le sprite utilizzate per rappresentare gli elementi grafici non sono state implementate con logiche di scalabilità dinamica.<sup>3</sup>

Passando all'implementazione della soluzione, ho creato un'interfaccia `SizeView` che astrae le informazioni dimensionali del pannello di gioco. I renderer utilizzano coefficienti di scala relativi (percentuali della dimensione dello schermo) per calcolare dinamicamente:

- Dimensioni degli sfondi e dei font proporzionali;
- Corretto posizionamento delle scritte informative;
- Scale factors per le animazioni.

Questo approccio segue il principio di **Interface Segregation**, fornendo solo le informazioni essenziali ai renderer senza esporre l'intera implementazione del `GamePanel`.

---

<sup>3</sup>Ciò significa che, durante l'esecuzione del gioco, le dimensioni degli elementi rimangono fisse, indipendentemente dalla risoluzione dello schermo o dalla dimensione della finestra di gioco.

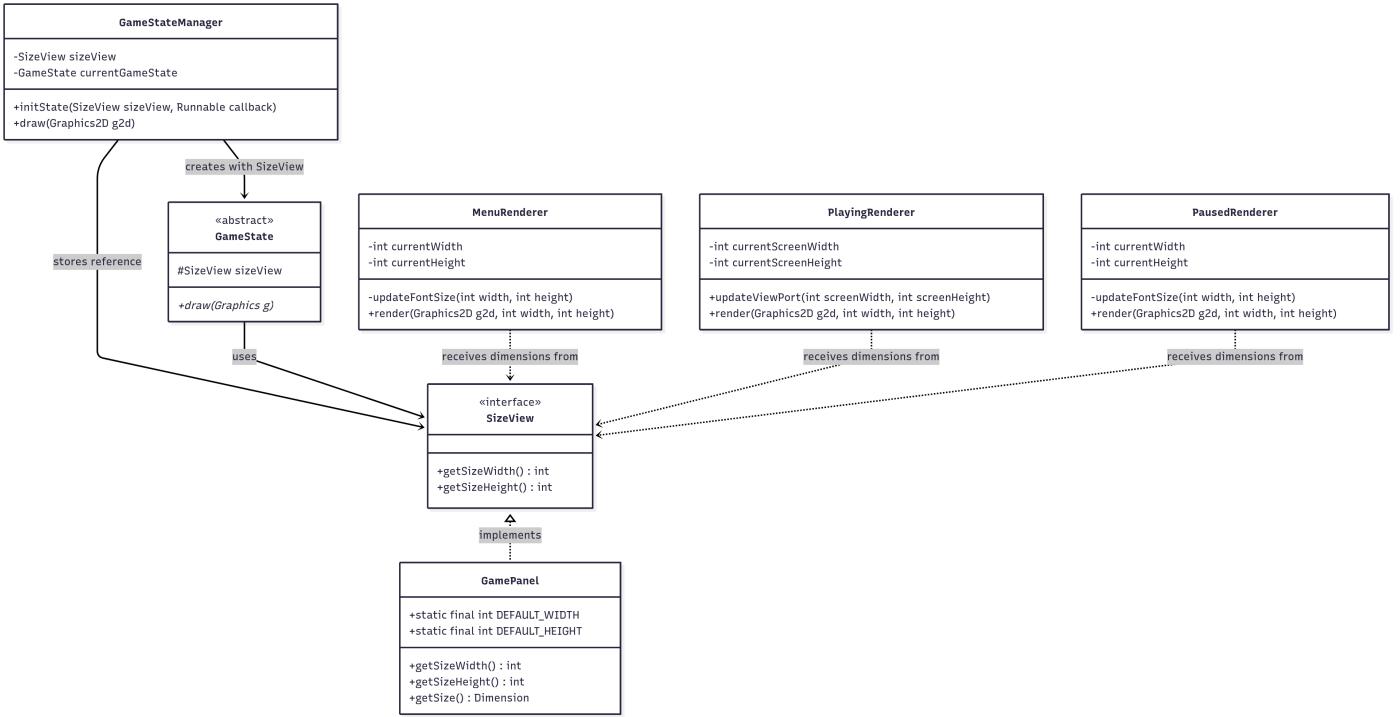


Figura 2.9: Rappresentazione UML semplificata dell'organizzazione relativa all'interfaccia `SizeView`

Pattern: Nessuno.

### Generazione e caricamento livello

**Problema:** Sviluppare un sistema per generare dinamicamente il livello e successivamente di istanziare correttamente tutte le entità del gioco mantenendo il codice scalabile ed estendibile.

**Soluzione:** Il livello viene inizialmente rappresentato dal `LevelGenerator` come una griglia bidimensionale di interi, dove ciascun valore identifica un tipo di elemento di gioco. Questo approccio consente di descrivere il layout in maniera semplice e modificabile. Il caricamento del livello è affidato al `LevelManager`, che interpreta la griglia e utilizza diverse fabbriche specializzate (`PlayerFactory`, `EnemyFactory` e `RingFactory`) per creare le entità corrispondenti. Tale approccio applica il **Factory Method Pattern** con l'obiettivo di centralizzare la creazione degli oggetti nelle factory specializza-

te, rendendo il sistema più flessibile e facile da estendere. Nello specifico del pattern:

- **Product**: l'interfaccia `Entity` implementata dall'omonima `EntityImp`.
- **Creator**: il `LevelManager` è la classe che agisce da creatore. Nonostante non abbia un metodo factory in senso stretto, il suo metodo `loadLevel()` utilizza le fabbriche per ottenere gli oggetti di cui ha bisogno.
- **ConcreteCreator**: le classi `PlayerFactory`, `EnemyFactory` e `RingFactory` sono le fabbriche concrete.

Nel design del sistema di gestione dei livelli, la mia responsabilità ha riguardato lo sviluppo delle classi `LevelManager` e `LevelGenerator`, mentre le factory specializzate e l'interfaccia `Entity` con relativa implementazione sono state sviluppate da Merlini. La scelta di integrare la spiegazione del design nella mia parte di relazione risiede nel fatto che il `LevelManager` rappresenta la parte conclusiva del pattern.

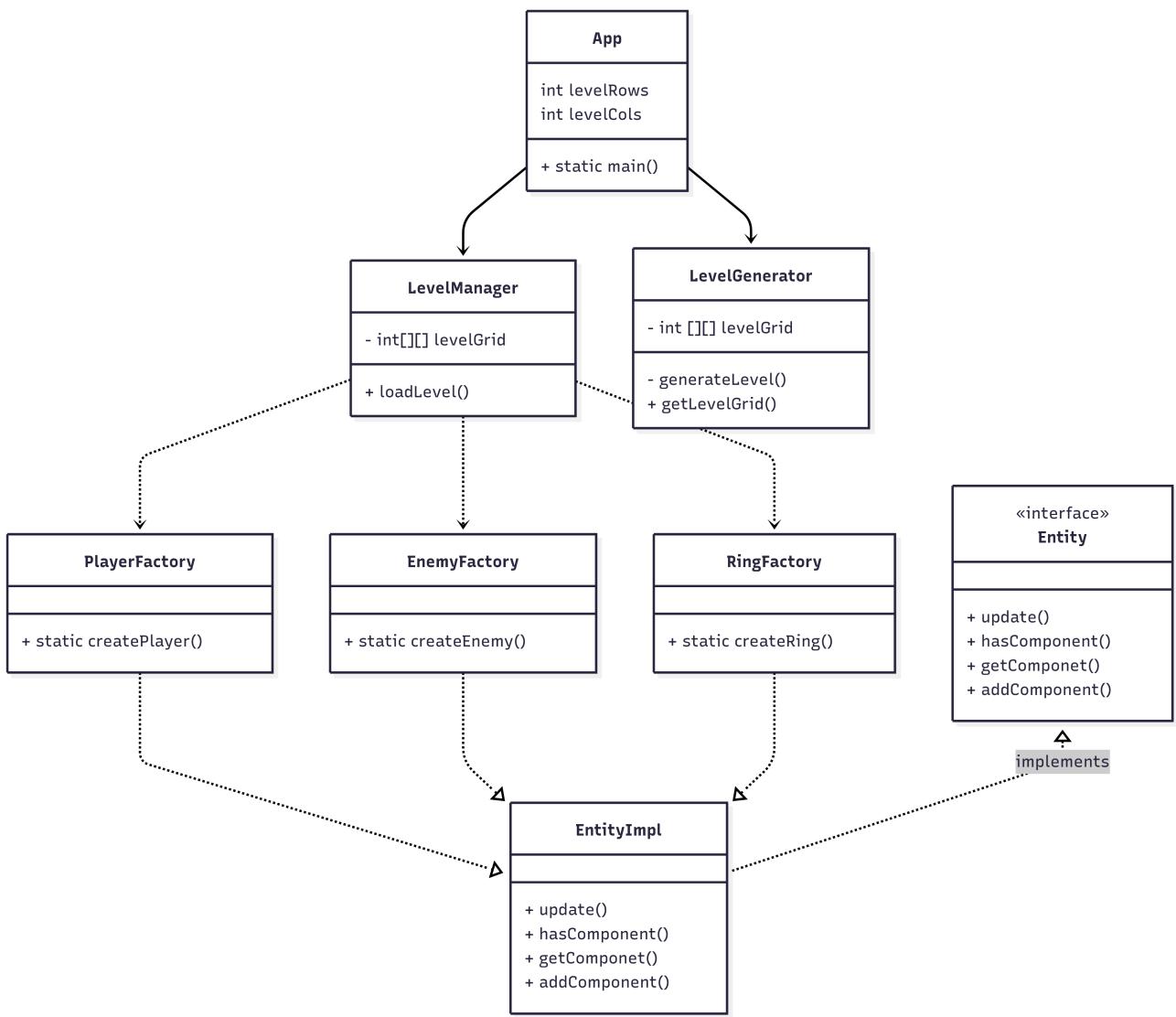


Figura 2.10: Rappresentazione UML semplificata dell'organizzazione della generazione e caricamento del livello

Pattern: Factory Method Pattern.

## 2.2.3 Sezione Merlini: Game Engine ed eventi

### Ciclo vitale entità

**Problema:** Particolare attenzione al ciclo vitale delle Entità, quando aggiornarle, rimuoverle, aggiungerle.

**Soluzione:** Definito il ciclo vitale delle entità: Aggiunta → Aggiornata → Rimossa.

Gestisco la concorrenza con diverse liste per le entità che vengono aggiornate in quelle sezioni critiche del codice, ovvero quando aggiorno un'entità faccio sì che quella vada aggiunta nella rispettiva lista. Quando va rimossa o aggiunta una certa entità, mi assicuro che mentre itero non venga aggiornata quell'entità, evitando eccezioni dovute ad una mancata gestione dell'occorrenza.

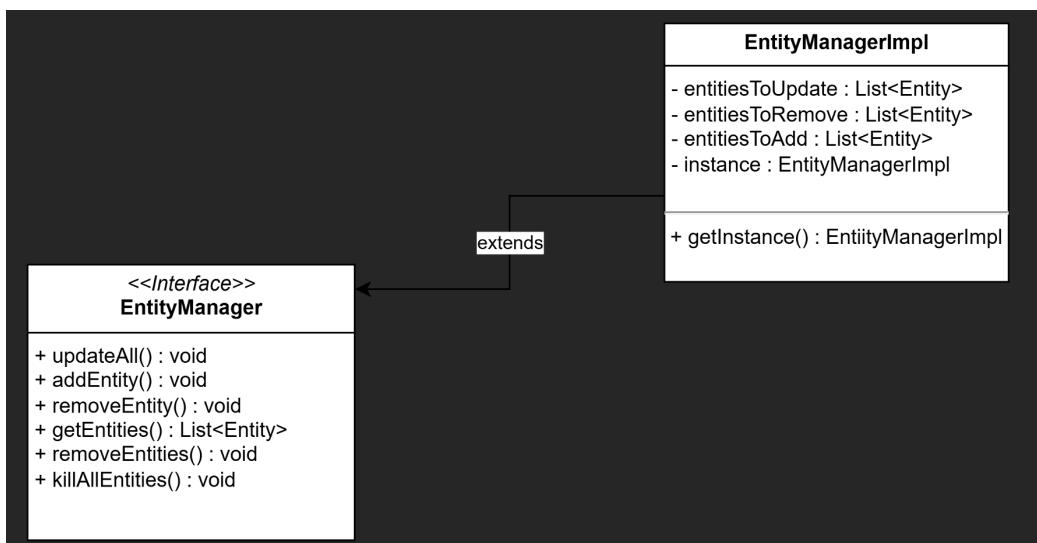


Figura 2.11: Rappresentazione UML dell'organizzazione dell'Entity Manager

Pattern: Nessuno.

## Decoupling eventi

**Problema:** Capire quali classi devono assumere il ruolo di **Subject** e/o **Observer**, e quando notificare gli eventi agli **Observer**, nel caso dei **Subject**.

**Soluzione:** Ragiona su quale classe compie azioni o modifiche, il **Subject**, che si ripercuotono su altre, l'**Observer**. Quando, ad esempio, dei **Subject** come **Physics** e le classi figlie devono notificare chi le ascolta, devo capire quali possono essere le classi interessate. Quando viene distrutto un'entità per via di una collisione, naturalmente il **EntityManager** deve saperlo, e quindi sarà l'**Observer** di quel **Subject** e ne gestirà l'evento.

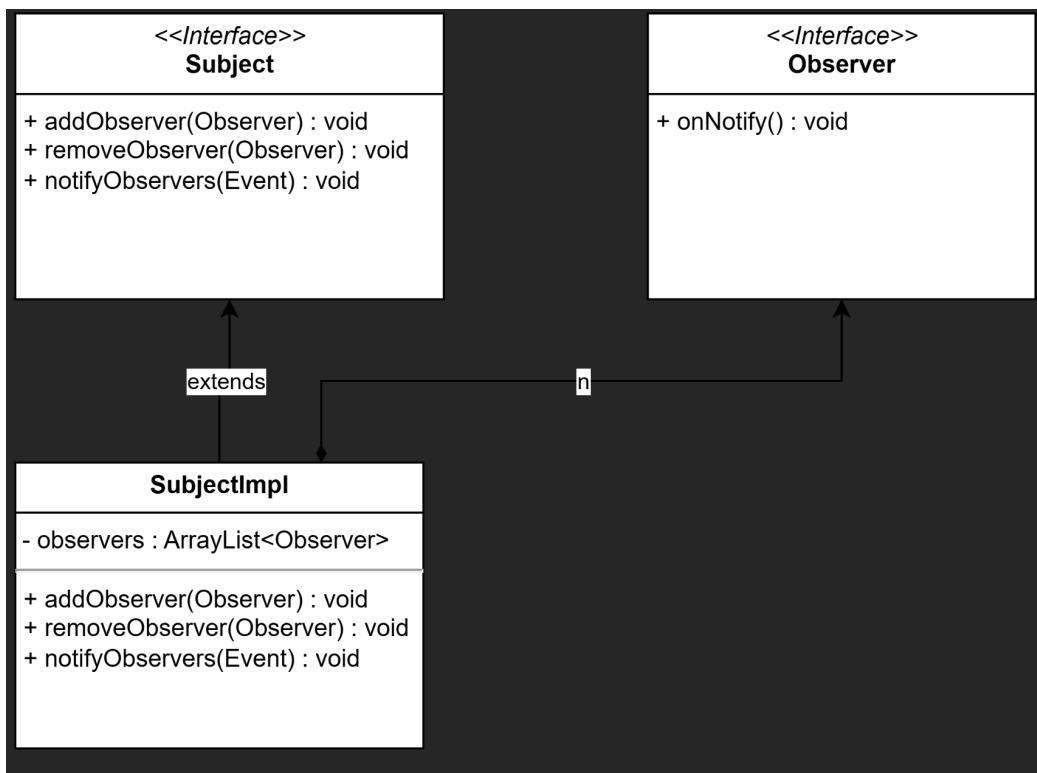


Figura 2.12: Rappresentazione UML Subject-Observer

Pattern: Subject-Observer.

## Gestione input

**Problema:** Gestire gli input tramite una mappatura delle azioni ai rispettivi comandi da tastiera.

**Soluzione:** Utilizzo un **InputManager** che legge gli input da tastiera attraverso dei **KeyListener** e li mappa alle relative azioni, insieme a un componente **InputComponent**, che gestisce gli eventi **KeyListener** tramite il **Subject-Observer pattern** per le azioni istantanee, come la pausa e il salto, mentre per azioni continue, come la corsa, si aggiorna costantemente con il polling dell'input salvato dall' **InputManager**.

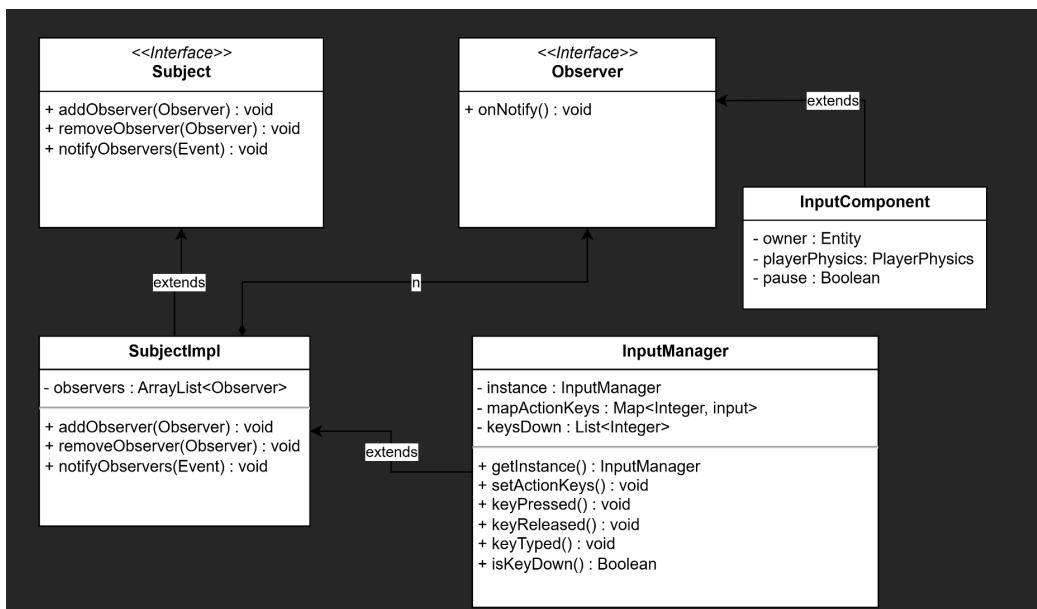


Figura 2.13: Rappresentazione UML dell'organizzazione dell'InputManager

Pattern: Subject-Observer.

## 2.2.4 Sezione Mancini: Animazioni

### Sistema animazioni

**Problema:** Creare un sistema di animazioni flessibile che possa gestire diversi tipi di entità (Sonic, nemici, anelli) con stati di animazione differenti, garantendo facilità di estensione

**Soluzione:** Ho implementato un sistema basato su generics con `GenericAnimator<T>` che utilizza lo **Strategy Pattern** per gestire due tipologie di animazioni. La classe è astratta e parametrizzata sul tipo di stato `T`, permettendo a ogni entità di definire i propri stati tramite `enum`. Il sistema utilizza un `record` interno `AnimationData` per incapsulare `frames`, `delay` e strategia di animazione. Le strategie sono implementate come `BiConsumer<GenericAnimator<T>, AnimationData<T>`, permettendo comportamenti diversi (looping per stati come “idle” e “spinning”, progressivo per azioni come “walking” e “running”). Ho utilizzato `Optional<T>` per gestire i valori che potrebbero essere `null`. Nel metodo `update()` uso:

```
Optional.ofNullable(animations.get(currentState)).ifPresent()
```

per eseguire la strategia solo se l'animazione esiste, evitando controlli `null` esplicativi.

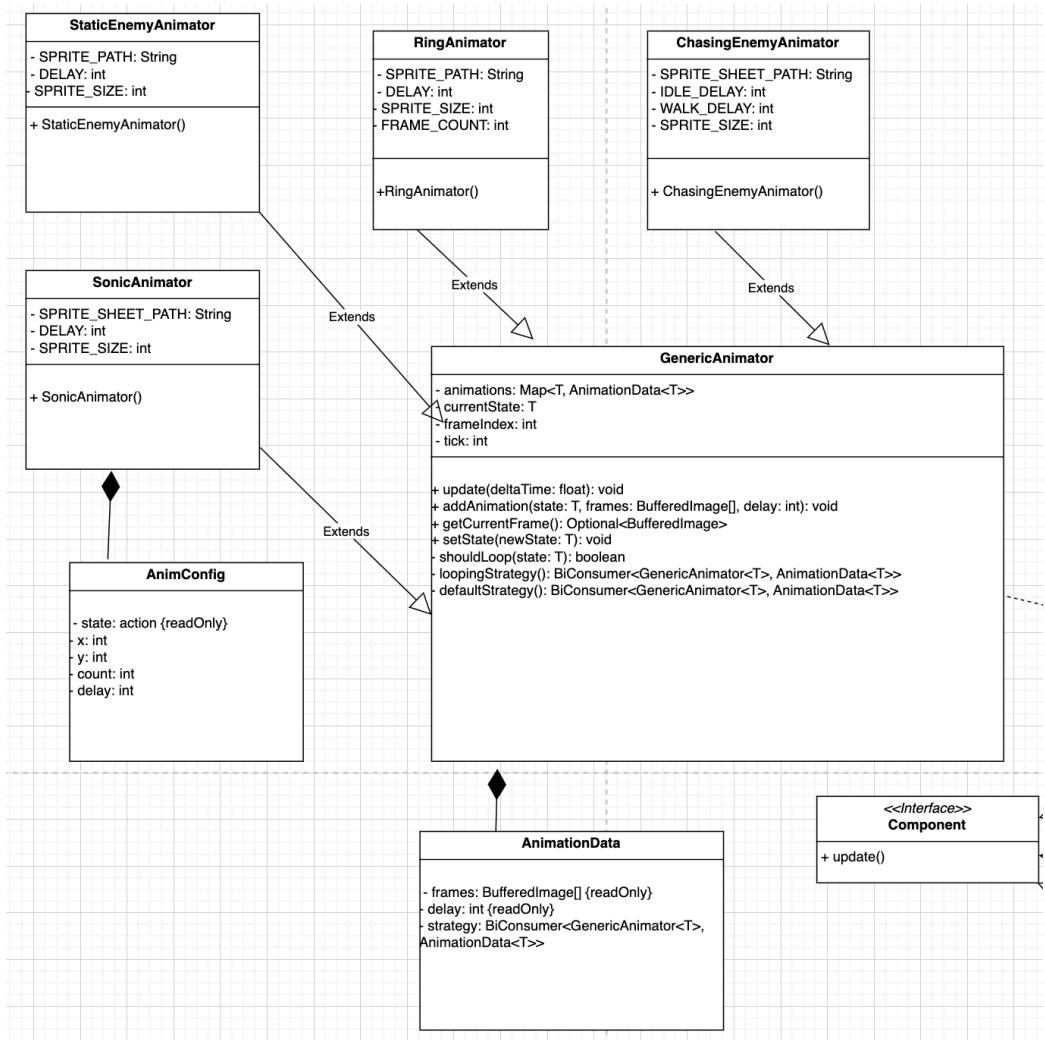


Figura 2.14: Rappresentazione UML dell'organizzazione del GenericAnimator

Pattern: Strategy + Template Method.

## Caricamento sprite

**Problema:** Gestire il caricamento e la suddivisione efficiente degli sprite sheet in frame individuali, evitando operazioni costose ripetute e garantendo flessibilità nel posizionamento dei frame.

**Soluzione:** Ho creato la classe `SpriteLoader` che utilizza `Stream` con `IntStream.range()` per generare i frame in modo funzionale. Il metodo `getFramesByPixels()` utilizza `mapToObj()` per trasformare gli indici in sotto-immagini tramite `getSubimage()`, e `toArray()` per salvare il risultato. La configurazione delle animazioni in `SonicAnimator` utilizza una `List` di `record AnimConfig`, processata con `forEach()` e *method reference*.

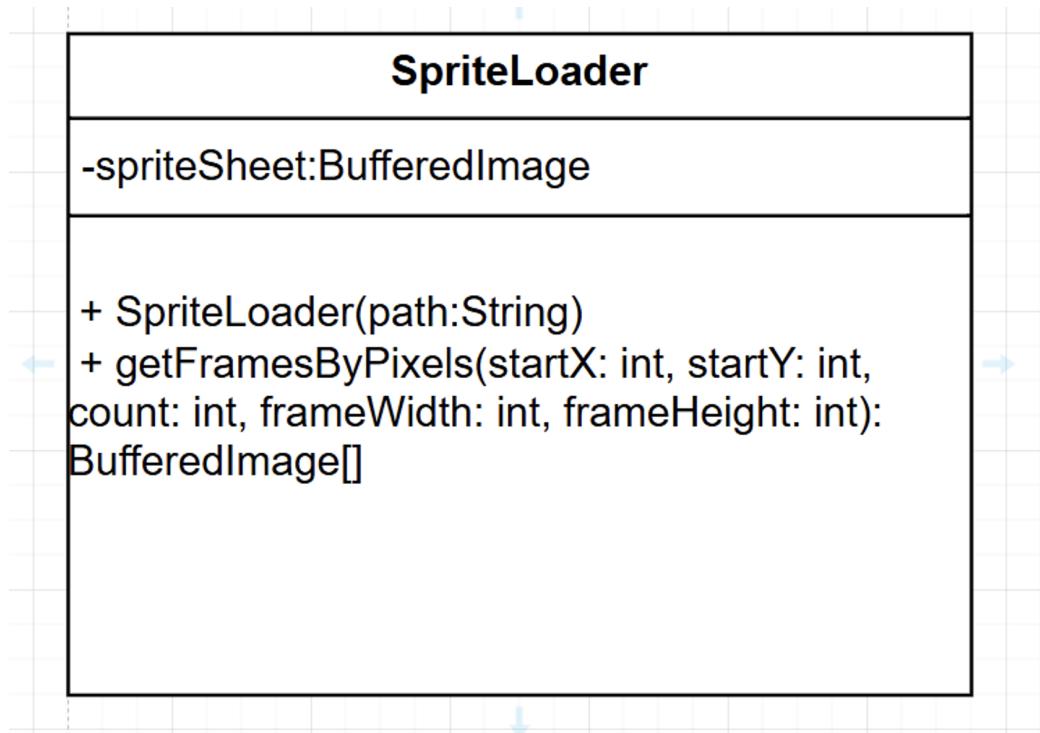


Figura 2.15: Rappresentazione UML della classe `SpriteLoader`

Pattern: Nessuno.

# Capitolo 3

## Sviluppo

### 3.1 Testing automatizzato

Sottoponiamo ad un test tutti i componenti relativi alla fisica, e l'implementazione dell'interfaccia per le Entità, e lo facciamo utilizzando JUnit.

Test sulla Fisica:

- Verifica che Sonic non possa compenetrarsi con le Tiles.
- Verifica che Sonic possa saltare e cadere dopo il salto.
- Verifica che Sonic cada anche senza aver prima saltato (c'era un bug per cui Sonic non cadeva senza aver prima saltato).
- Verifica il game over.
- Verifica che i nemici inseguano Sonic e non si compenetrino con le Tiles
- Verifica che gli anelli vengano raccolti da Sonic.

## 3.2 Note di sviluppo

### 3.2.1 Samuele Bizzocchi

#### Esempi di codice significativo

Metodi per la gestione del movimento e delle collisioni

```
public boolean canGoThere(direction dir, float distance){  
    /*This method is to be used for both movement of X  
     axis and Y axis. It simply determines if the  
     entity that owns  
     * this physics instance can move in a certain way  
     without colliding with any tile.  
    */  
    Rectangle2D.Float wannaBeThere = new Rectangle2D.  
        Float();  
    if (dir == direction.right || dir == direction.left){  
        wannaBeThere.setRect(hitbox.getX() + distance,  
            hitbox.getY(), hitbox.getWidth(), hitbox.  
            getHeight());  
    } else if (dir == direction.up || dir == direction.  
        down){  
        wannaBeThere.setRect(hitbox.getX(), hitbox.getY()  
            + distance, hitbox.getWidth(), hitbox.  
            getHeight());  
    }  
    /*check if moving as the entity wants to would cause  
     it to compenetrates in any tile*/  
    for (Rectangle2D.Float tile : tiles) {  
        if (wannaBeThere.intersects(tile)){  
            return false;  
        }  
    }  
    return true;  
}  
  
public void moveY(float deltaTime){  
    /*JUMPING*/  
    if(jumpFrames > 0){  
        if(canGoThere(direction.up, ySpeed)){  
            jumpFrames--;  
        } else {  
            /*hitting the ceiling causes him to start  
             falling */  
        }  
    }  
}
```

```

                jumpFrames = 0;
                ySpeed = 0;
            }
        }
        /*FALLING. Sonic starts to fall only one update
         after he ran out of jumpingFrames*/
        else if (canGoThere(direction.down, Math.max(
            initFallSpeed, ySpeed))){
            if (ySpeed <= 0){
                ySpeed = initFallSpeed;
            } else if (ySpeed < maxFallSpeed){
                ySpeed += fallMod;
            }
        }
        /*LANDING.    if Sonic is't already on the ground, he
         lands*/
        else if (canGoThere(direction.down, Float.MIN_VALUE))
        {
            if(ySpeed > 0){
                landing();
            }
        }
        owner.getComponent(TransformComponent.class).moveY(
            ySpeed);
    }

    public void landing() {
        /*If the falling speed don't connect precisely the
         Entity and the ground under it, the Entity will
         just
         * keep floating and falling endlessly, or
         compenetrates in the ground.
         * This method come in handy for that problem,
         asserting the Tile that forms a floor, closest to
         * the Entity, and "fixing" the Entity to it.
         */
        float maxHeight = Float.MAX_VALUE;
        ArrayList<Rectangle2D.Float> candidates = new
            ArrayList<>();
        TransformComponent transform = owner.getComponent(
            TransformComponent.class);
        for (Rectangle2D.Float tile : tiles) {
            Rectangle2D.Float projection = new Rectangle2D.

```

```

        Float(transform.getX(), (float) tile.getY(),
        transform.getWidth(), transform.getHeight());
    if (projection.intersects(tile) && tile.getY() >=
        transform.getY() + transform.getHeight()) {
        candidates.add(projection);
    }
}
for (Rectangle2D.Float candidate : candidates) {
    if (candidate.getY() < maxHeight) {
        maxHeight = (float)candidate.getY();
    }
}
transform.setY(maxHeight - transform.getHeight());
ySpeed = 0;
}

```

### Librerie utilizzate

- `java.awt.geom.Rectangle2D`

Non ho utilizzato nessuna delle feature avanzate elencate nel template. Non ho trovato punti del codice dove convenisse utilizzare *lambda expressions*, perché ho riscontrato che causavano un fortissimo lag sin dai primi secondi di gioco.

Per il sistema delle collisioni mi sono ispirato a un sistema mostrato in questo video: link. All'inizio ho completamente deviato dall'impostazione del codice mostrata in questo tutorial, pensando che fosse fin troppo semplice per raggiungere le 70 ore obbligatorie. Tuttavia, giunto oltre la deadline, mi sono reso conto che la soluzione da me ideata non funzionava affatto.

### 3.2.2 Francesco Giancaterino

#### Utilizzo di Lambda Expressions

**Feature:** Method reference per callback di shutdown

**Link:** GitHub - Game.java Linea 45

#### Utilizzo di Java 2D

- **Feature:** Java 2D API - Font loading e TrueType parsing  
**Link:** GitHub - MenuRenderer.java

- **Feature:** `java.awt.Graphics2D` - API grafiche avanzate (Gradient-Paint, AffineTransform, ecc.)  
**Link:** GitHub - PlayingRenderer.java

Per il font delle scritte ho utilizzato : NiseSegaSonic.TTF

Per le immagine ho utilizzato rispettivamente : Menu e Pausa

## Game Loop

**Feature:** Fixed timestep game loop

Per quanto riguarda lo sviluppo del loop di gioco, mi sono trovato in difficoltà; ho utilizzato diversi tutorial per capire come gestire al meglio questa problematica. Il più importante è stato sicuramente questo: YouTube - Fixed Timestep Game Loop

### 3.2.3 Filippo Merlini

#### Uso di Stream e Lambda Expressions

Per la gestione degli eventi è stato fatto uso di **Java Stream API e Lambda Expressions**, al fine di rendere il codice più compatto e leggibile, riducendo la logica boilerplate tipica dei cicli esplicativi.

**Link:** GitHub - SubjectImpl.java (linea 28)

#### Librerie usate

- `java.awt.event.KeyEvent`
- `java.awt.event.KeyListener`

Ho consultato diverse guide per impostare l'architettura: Game Programming Patterns - Component Pattern e Game Programming Patterns - Observer Pattern

Avevo optato inizialmente per un approccio *Entity-Component-Systems*, ma ho poi adottato un più semplice *Component Pattern*, perché sviluppare il primo avrebbe occupato molte più ore di lavoro senza un reale ritorno in termini di prestazioni e funzionalità, a causa della presenza del Garbage Collection in Java.

### 3.2.4 Nicolas Mancini

#### Metodi più rilevanti

```
@Override
public void update(final float deltaTime) {
    Optional.ofNullable(animations.get(currentState))
        .ifPresent(anim -> anim.strategy().accept(
            this, anim));
}

/**
 * Strategy for looping animations
 * Frames repeat in a cycle
 */
private static <T> BiConsumer<GenericAnimator<T>,
    AnimationData<T>> loopingStrategy() {
    return (animator, anim) -> {
        animator.tick++;
        if (animator.tick >= anim.delay()) {
            animator.tick = 0;
            animator.frameIndex = (animator.frameIndex +
                1) % anim.frames().length;
        }
    };
}

/**
 * Strategy for non-looping animations
 * Plays frames once and stops at the last one
 */
private static <T> BiConsumer<GenericAnimator<T>,
    AnimationData<T>> defaultStrategy() {
    return (animator, anim) -> {
        animator.tick++;
        if (animator.frameIndex < anim.frames().length -
            1) {
            if (animator.tick >= anim.delay()) {
                animator.tick = 0;
                animator.frameIndex++;
            }
        }
    };
}
```

```

    /**
     * Determines whether the state should use a looping
     * strategy.
     *
     * @param state the animation state
     * @return true if the state should loop, false otherwise
     */
    private boolean shouldLoop(final T state) {
        final String name = state.toString().toLowerCase();
        return name.equals("idle") || name.equals("spinning")
    }

    /**
     * Adds a new animation for a given state.
     *
     * @param state the animation state
     * @param frames the frames of the animation
     * @param delay the delay between frame changes
     */
    public void addAnimation(final T state, final
        BufferedImage[] frames, final int delay) {
        final BiConsumer<GenericAnimator<T>, AnimationData<T
            >> strategy =
            shouldLoop(state) ? loopingStrategy() :
            defaultStrategy();
        animations.put(state, new AnimationData<>(frames,
            delay, strategy));

        if (currentState == null) {
            currentState = state;
        }
    }

    /**
     * Returns the current frame of the animation, if
     * available
     *
     * @return an Optional containing the current
     *         BufferedImage frame
     */
    public Optional<BufferedImage> getCurrentFrame() {

```

```

        return Optional.ofNullable(animations.get(
            currentState))
            .filter(anim -> anim.frames().length > 0)
            .map(anim -> anim.frames()[frameIndex]);
    }

    SpriteLoader

    /**
     * Extracts a series of frames from the loaded sprite
     * sheet
     *
     * @param startX the starting X pixel coordinate
     * @param startY the starting Y pixel coordinate
     * @param count the number of frames to extract
     * @param frameWidth the width of each frame in pixels
     * @param frameHeight the height of each frame in pixels
     * @return an array of images
     */
    public BufferedImage[] getFramesByPixels(int startX, int
        startY, int count, int frameWidth, int frameHeight) {
        return IntStream.range(0, count)
            .mapToObj(i -> spriteSheet.getSubimage(
                startX + i * frameWidth,
                startY,
                frameWidth,
                frameHeight
            ))
            .toArray(BufferedImage[]::new);
    }
}

```

## Utilizzo di Optional e Lambda Expression

**Feature:** Sistema di animazione estensibile mediante Generics

**Link:** GitHub - GenericAnimator.java

### Librerie usate

- java.util.Optional
- java.util.stream.IntStream
- java.util.function.BiConsumer

- `java.awt.image.BufferedImage`
- `javax.imageio.ImageIO`

Risorse consultate:

- The Spriters Resource per le sprite.
- YouTube per la logica di cambio animazione con tick e delay.
- StackOverflow per l'approccio iniziale al caricamento delle sprite.

Inizialmente avevo implementato un sistema di animazioni definite staticamente, con classi separate per ogni tipo di entità e logica di animazione duplicata. Ogni animator aveva il proprio set di metodi specifici e la gestione dei frame era ripetitiva e poco flessibile. A metà strada, mi sono reso conto che questo approccio non scalava bene e stava generando molto codice duplicato. Ho quindi deciso di **refactorare completamente il sistema**, creando la classe generica `GenericAnimator<T>` che utilizza lo *Strategy Pattern* per gestire diversi comportamenti di animazione e i vari animator che la estendono. La migrazione dal sistema hardcoded a quello generico ha richiesto di riscrivere una buona percentuale delle classi, ma il risultato finale è molto più pulito e manutenibile. Il caricamento delle immagini, dato che nel foglio originale non sono raggruppate sempre per stato, richiede comunque la presenza di qualche numero statico.

# Capitolo 4

## Commenti finali

### 4.1 Autovalutazione e lavori futuri

#### 4.1.1 Samuele Bizzocchi

Francamente sono stato molto deluso dalle mie personali capacità organizzative e progettistiche. La fase di analisi e design è stata un completo disastro, indi per cui anche la fase di sviluppo ha richiesto un'enormità di tempo rispetto a quello che ci aspettavamo. Abbiamo iniziato a scrivere ognuno il proprio codice senza guardare oltre la punta del nostro naso, senza pensare all'architettura generale prima di tutto; questa specie di reverse engineering ci è costata cara, costringendoci a reinventare l'architettura un paio di volte. Dal punto di vista personale ritengo che la mia parte di lavoro fosse fin troppo semplice, tanto che mi sono complicato a dismisura il lavoro proprio per paura che il risultato finale non fosse soddisfacente ai fini della valutazione, e forse ho finito anche per penalizzare il mio voto. Tuttavia ho fatto ammenda per questa diseguaglianza nel carico di lavoro occupandomi di tutta la parte del Testing Automatico perlopiù da solo, dirigendo il lavoro di altri membri del gruppo (di cui non farò né nome né numero) che non avevano le idee ben chiare sui loro compiti, ed essendo in generale la persona che spronava il resto del gruppo a pensare e rimediare agli errori iniziali di progettazione. Sul lato positivo dell'esperienza, ho imparato veramente ma veramente tanto lavorando a questo progetto, e so che la prossima volta farò molto meglio. Penso anche che abbiamo fatto un ottimo lavoro per la scalabilità del progetto (anche se sicuramente non perfetto). La suddivisione dei compiti poteva essere migliore, ma ci siamo aiutati a vicenda ed abbiamo coperto l'uno le mancanze e le lacune dell'altro, quindi il teamwork c'è stato e si è sentito.

#### 4.1.2 Francesco Giancaterino

Lavorare per la prima volta in team a un progetto di questa portata è stata un’esperienza estremamente formativa, poiché mi ha posto di fronte a problematiche nuove che hanno richiesto tempo, impegno e dedizione per essere affrontate e risolte. Il mio ruolo principale è stato lo sviluppo del core del gioco e del sistema di rendering. Quest’ultimo si è rivelato più complesso di quanto mi aspettassi, in quanto è stato necessario integrarlo e coordinarlo con il sistema di animazioni e con la gestione della fisica del personaggio. Questo aspetto ha favorito la collaborazione con i colleghi, con i quali ho avuto modo di discutere e confrontarmi sulle possibili soluzioni alle diverse difficoltà emerse durante lo sviluppo. In generale sono soddisfatto del lavoro svolto, anche se avrei voluto gestire con maggiore attenzione le tempistiche di sviluppo. Dal punto di vista personale, questa esperienza mi ha permesso di sviluppare ulteriormente le mie competenze tecniche e di confrontarmi per la prima volta con lo sviluppo di un software in maniera organizzata e collaborativa. L’uso di Git per lo sviluppo del progetto e la gestione di problematiche nuove e complesse si è rivelato particolarmente stimolante, contribuendo alla mia crescita professionale e alla comprensione delle dinamiche di lavoro in team. In prospettiva, mi sarebbe piaciuto curare maggiormente il design complessivo del gioco e approfondire la parte relativa alla grafica, magari tramite l’impiego di librerie più avanzate. Tuttavia, per ragioni legate alle tempistiche, non è stato possibile dedicare a questi aspetti l’attenzione desiderata.

#### 4.1.3 Filippo Merlini

Fondamentalmente il mio ruolo nel progetto è consistito nella **progettazione dell’applicazione**, in particolare nella gestione degli oggetti di gioco, del loro ciclo vitale e del modo in cui questi rispondono agli eventi, ricevuti sotto forma di notifica dalle altre parti del progetto svolte dai miei colleghi. Ho dovuto costantemente mettere mano a tali parti per mantenerle in “contatto” nel miglior modo possibile. Un’altra parte che mi è costata diverse ore di lavoro è stata la **rifinitura del codice** affinché rispettasse principi come *SRP* (Single Responsibility Principle) e *OCP* (Open/Closed Principle), cercando di mantenerlo il più semplice, estendibile e qualitativo possibile. La scelta e l’impegno di seguire i pattern come *Observer*, *Component* e anche il *Singleton* ha richiesto un’attenzione ulteriore alla coerenza del codice. L’unione di questi approcci ha portato anche a seguire una forma di *MVC*, sebbene non in modo “strict”. Un esempio può essere visto nell’*HUDComponent*, una parte considerata *model* che però svolge attività da

*view: GitHub - HUDComponent.java* Tutto sommato, il risultato finale non mi dispiace. Rimpiango soltanto di non essere riuscito a implementare una funzionalità aggiuntiva come il **sonoro**, a causa della mancanza di tempo per il testing. Infine, **lavorare in gruppo** è stato molto interessante dal mio punto di vista, specialmente nel seguire il lavoro di Bizzocchi per garantire che le sue implementazioni avessero la giusta conseguenza in run-time.

#### 4.1.4 Nicolas Mancini

La mia parte del progetto si è concentrata sul sistema di animazioni, quindi caricamento delle sprite da file e gestione delle transizioni tra i vari stati delle entità. È stato bello lavorare in team, mi sono interfacciato principalmente con Bizzocchi per far funzionare insieme fisica e animazioni, e con Giancatrino per il coordinamento generale. Ci siamo trovati spesso in presenza, cosa che si è rivelata fondamentale per risolvere i problemi di integrazione. Il carattere del gruppo è stato buono, tutti si sono messi a disposizione con buona volontà e impegno. Ho avuto alcuni rallentamenti dovuti anche a impegni extrascolastici che si sono sovrapposti. Inoltre, ho perso troppo tempo su cose probabilmente inutili - perfezionamenti e dettagli che alla fine non erano così importanti. Avendo inoltre cambiato architettura e anche suddivisione di certi compiti a lavori in corso, altro tempo è stato impiegato lì. Come primo progetto di questa portata ci sta aver fatto errori di priorità. Sono contento del risultato finale, credo che il sistema sia scalabile e utilizzi bene le feature avanzate di Java. L'esperienza di gruppo è stata molto formativa, anche se a volte la coordinazione richiedeva più tempo del previsto. Come sviluppi futuri sicuramente l'aggiunta di nuovi nemici, oggetti e power-up comporterà nuove animazioni, eventualmente con nuove strategie. Per decidere le strategie si potrebbero usare enum come proprietà e non stringhe esplicite che rischiano di essere fragili.

## 4.2 Difficoltà incontrate e commenti per i docenti

Non posso ripeterlo abbastanza, la fase di progettazione è stata una totale catastrofe, e la colpa è 100% nostra. Non abbiamo considerato innumerevoli fattori, nodi fra i nostri progetti, e limitazioni di java. Non ci siamo informati abbastanza sugli strumenti che stavamo utilizzando e questo ci ha causato (più a me che agli altri, io specialmente sono colpevole di questa mancanza) di scrivere molto più codice del necessario, solo per cancellarlo in seguito. Abbiamo anche riscontrato tutti e quattro INNUMEREVOLI problematiche

legate a Gradle. A lezione non avevamo mai riscontrato problemi simili. In tanto ogni volta che eseguivamo il programma con gradle github riconosceva le innumerevoli cartelle e file creati, creando così un gran scompiglio nello stash. Inoltre in più istanze abbiamo trovato che per alcuni di noi il codice di una branch funzionava perfettamente lanciandolo con ./gradlew run, invece per altri questo non partiva affatto, o ancora peggio, eseguiva frammenti di codice passati, già rimossi dall'intera repository. Ritengo che questo abbia a che fare con impostazioni di sistema dei nostri computer, ma non siamo mai riusciti a risolvere questa problematica e forse dovrebbe essere discussa a lezione. Per finire, anche se non ritengo sia colpa del corso se ci siamo dimostrati così impreparati alla fase di Analisi e Design, ritengo che sia necessario mettere alla prova gli studenti su questo aspetto anche prima della fine delle lezioni. In altre parole ritengo necessario assegnare agli studenti anche esercizi più lunghi, più simili a progetti a tutto tondo (quindi come questo che abbiamo appena svolto, ma più in piccolo); lo scopo di ciò è far sì che gli ignoranti come me si rendano conto per tempo di quanto sia importante avere un'architettura ben definita PRIMA di iniziare a scrivere codice.

# Appendice A

## Guida utente

### Breve spiegazione dei comandi e delle dinamiche di gioco

- All'avvio l'utente si trova direttamente nel menu principale (Figura A.1), dal quale è possibile avviare la partita premendo il tasto ENTER.

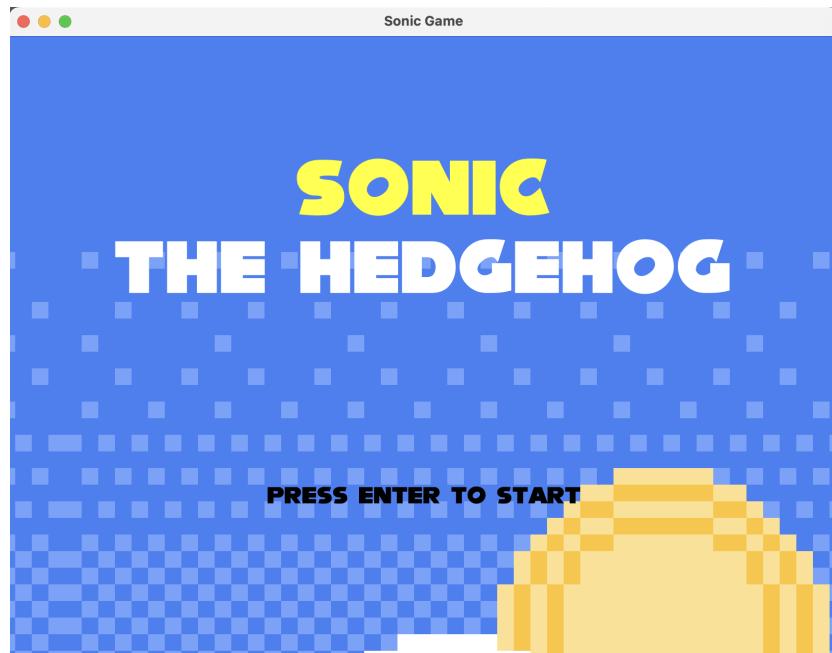


Figura A.1: Menu del gioco

- Il livello (Figura A.2) si sviluppa orizzontalmente, l’obiettivo è raggiungere il traguardo (rappresentato dalla bandiera blu) superando i nemici presenti; una volta completato il livello, il gioco rimanda nuovamente alla schermata di menu.
- In caso di caduta o di morte del personaggio (collisione con un nemico senza anelli), il giocatore viene riportato al menu iniziale e ha la possibilità di ricominciare la partita.
- I nemici possono essere sconfitti saltando direttamente sopra di essi.
- In alto a sinistra è presente il conteggio degli anelli raccolti che si aggiorna dinamicamente.

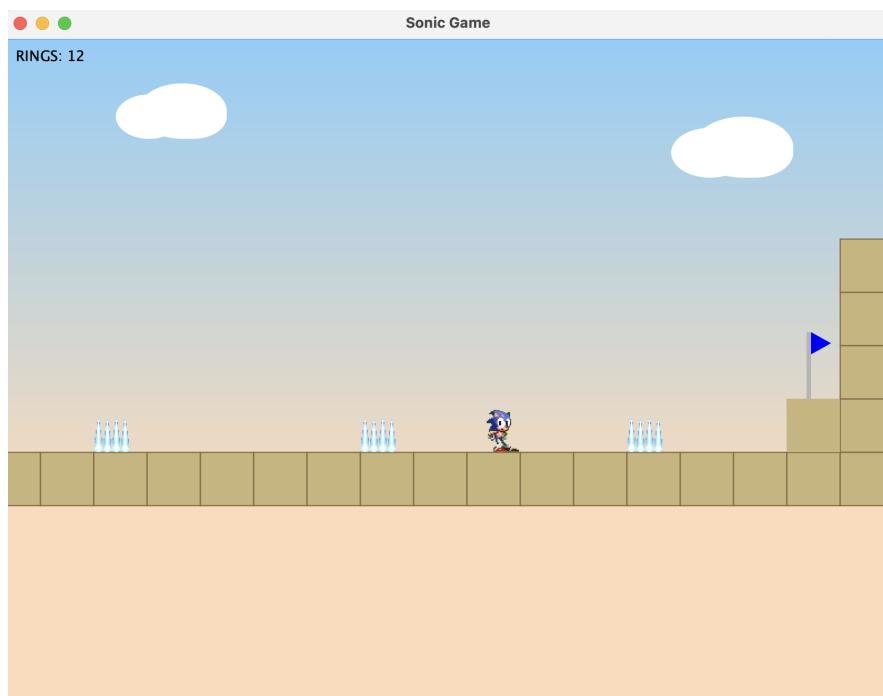


Figura A.2: GamePlay e traguardo

- È inoltre prevista una schermata di Pausa (Figura A.3), attivabile in qualsiasi momento durante la partita premendo il tasto P.
- Premendo nuovamente lo stesso tasto si riprende la partita, mentre per chiudere il gioco è possibile utilizzare il comando dedicato o i pulsanti di sistema.

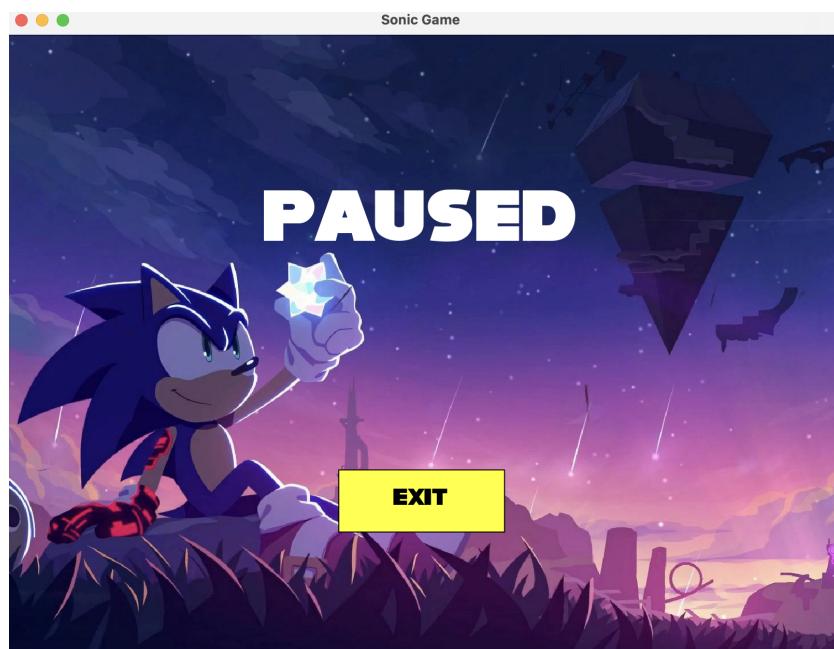


Figura A.3: Pausa