SMB
"Super Meat Boy"

Francesco Marcatelli, Nicola Graziotin, Mattia Galli, Martin Tomassi

16 Gennaio 2024

Indice

Capitolo 1

Analisi

L'applicazione proposta consiste nella realizzazione di un remake del gioco platform "Super Meat Boy". Il giocatore controlla Meat Boy, un cubo di carne, che deve salvare la sua fidanzata attraverso livelli intricati. La difficoltà cresce gradualmente, richiedendo abilità e riflessi veloci. Il gioco incoraggia la ripetizione e l'apprendimento, con tempi veloci e respawn istantaneo. Ogni livello presenta ostacoli come seghe circolari e spuntoni da evitare saltando sulle varie piattaforme per arrivare alla fine senza morire con il numero minore di tentativi possibile.

1.1 Requisiti

Requisiti funzionali

- Creazione di un livello con ostacoli statici, piattaforme e obiettivo di fine livello.
- Personaggio giocabile in grado di muoversi nelle 4 direzioni, saltare e aggrapparsi alle pareti.
- Grafica minimale ispirata al gioco Super Meat Boy.
- Gestione delle collisioni.

Requisiti non funzionali

- Il gioco dovrà essere utilizzabile su macchine Windows, Mac OS e Unix/Linux.

1.2 Analisi e modello del dominio

Sono presenti 3 tipi di entità : il personaggio principale, il personaggio che raffigura il traguardo e gli ostacoli. Il personaggio principale controllato dal giocatore dovrà cercare di evitare tutti gli ostacoli presenti nel livello arrivando al traguardo. Il personaggio secondario, che rappresenta il traguardo del livello, è statico, appena toccato dal personaggio giocabile terminerà il livello e si passerà al prossimo. Gli ostacoli rappresentano seghe circolari e spuntoni che una volta toccati terminano la corsa del personaggio, che dovrà ricominciare il livello da capo. Un controllo delle collisioni col giocatore, e agire di conseguenza se ne rileva una.

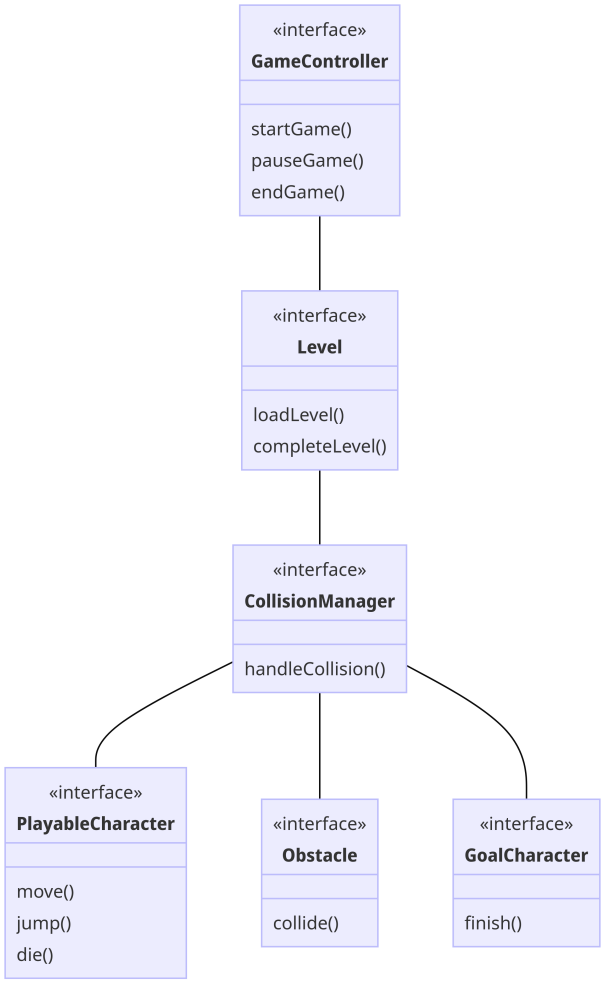

Figura 1.1: Schema UML delle principali entità

Capitolo 2

Design

2.1 Architettura

L'architettura di Super Meat Boy segue il pattern architetturale MVC, in modo da dividere i compiti specifici per ogni sezione del gioco, cioè:

- Model: rappresenta la logica del gioco e lo stato interno. Si occupa dell'aggiornamento continuo del modello di gioco in base alle azioni dell'utente e agli eventi di gioco. Fornisce metodi per aggiornare la posizione del giocatore, gli ostacoli e le piattaforme.

- View: rappresenta l'interfaccia utente e si occupa della visualizzazione dei componenti grafici del gioco. Fornisce metodi per renderizzare il giocatore, il traguardo, gli ostacoli e le piattaforme sulla schermata. Interagisce principalmente con l'utente mostrando gli elementi del gioco e rispondendo agli input dell'utente.

- Controller: gestisce l'input dell'utente e coordina il flusso di controllo del gioco. Ha responsabilità come la gestione degli input utente, l'aggiornamento dello stato del gioco e l'esecuzione del ciclo di gioco. Risponde agli input dell'utente chiamando i metodi appropriati del modello e della vista.

Grazie a questa scelta la view non porterà con sè cambiamenti anche sul model e sul controller.

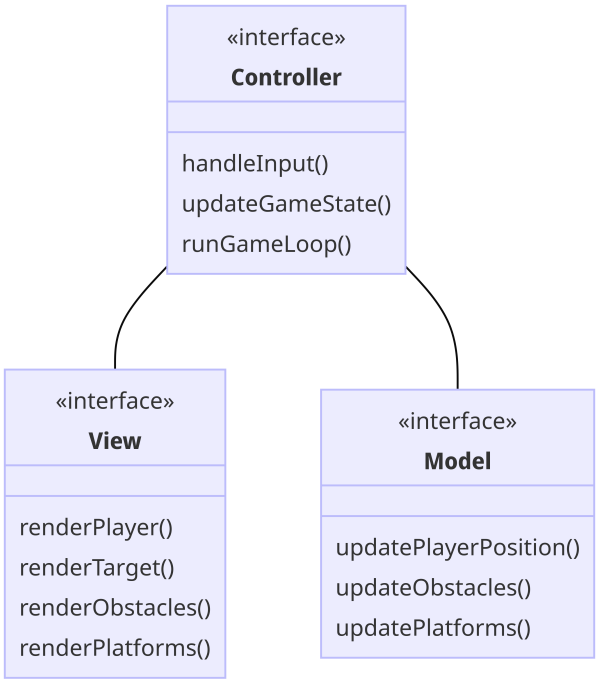

Figura 2.1: Schema UML dell'architettura

2.2 Design dettagliato

2.2.1 Francesco Marcatelli

Controllo e gestione delle collisioni
Il progetto Java in esame si basa sul pattern architetturale Model-View-Controller (MVC) per organizzare e strutturare il codice in modo modulare e manutenibile. Nel contesto di questo progetto, sono state implementate diverse classi, di cui ci concentreremo su tre principali: Hitbox¡T¿, CollisionCheckerImpl, e CollisionHandlerImpl. Queste classi svolgono ruoli chiave nella gestione delle collisioni e nell'aggiornamento del modello di gioco.

La Classe Hitbox$< T >$
La classe Hitbox$< T >$ rappresenta la dimensione reale di un'entità nel sistema di gioco. Essa è progettata con una struttura generica, dove T rappresenta la forma della hitbox. La classe fornisce metodi per l'aggiornamento della posizione, il recupero della forma della hitbox e il disegno a scopo di

test.

Problemi Risolti

1. **Gestione delle Dimensioni:**
   Problema Nel contesto di un gioco o di un'applicazione grafica, è fondamentale gestire con precisione le dimensioni spaziali di ciascuna entità. Le collisioni e le interazioni tra gli oggetti dipendono direttamente dalla loro effettiva dimensione nello spazio di gioco.

   Soluzione La classe Hitbox$< T >$ fornisce un'interfaccia standard per la rappresentazione della hitbox di un'entità. La genericità della classe consente di utilizzare diverse forme (T) per rappresentare la hitbox, adattandosi alle specifiche esigenze del progetto. In questo modo, la gestione delle dimensioni spaziali diventa modulare ed estensibile.

2. **Interazione Spaziale:** *Problema* La gestione delle collisioni e delle interazioni spaziali tra entità è cruciale per un'applicazione di gioco. Senza un sistema robusto, potrebbero verificarsi problemi di sovrapposizione o mancanza di rilevamento delle collisioni.

   Soluzione La classe CollisionCheckerImpl utilizza le hitbox delle entità (come CircularHitbox e RectangleHitbox) per rilevare collisioni in modo efficace. La hitbox generica fornisce un modo flessibile di gestire le diverse forme delle entità, permettendo un'implementazione modulare della logica di collisione.

3. **Test Grafico:** *Problema* Durante lo sviluppo di un'applicazione grafica, è essenziale testare visivamente la posizione e le dimensioni delle hitbox per assicurarsi che la logica di collisione funzioni correttamente.

   Soluzione Il metodo draw della classe Hitbox$< T >$ fornisce un'opzione per disegnare la hitbox a scopo di test. Questo facilita il processo di sviluppo e debugging, consentendo agli sviluppatori di visualizzare graficamente le hitbox e di verificare la correttezza della gestione delle collisioni.

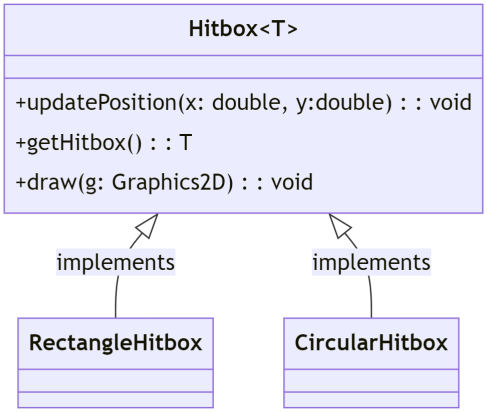

Figura 2.2: Schema UML dell'architettura

La Classe CollisionChecker
La classe CollisionCheckerImpl implementa l'interfaccia CollisionChecker ed è responsabile di verificare le collisioni tra diverse hitbox nel modello di gioco. Gestisce anche lo stato di collisione di un personaggio di gioco (MeatBoy), aggiornando la posizione in base alle interazioni.

Problemi risolti

1. **Collision Detection:**
   Problema Nell'ambito di un gioco, è cruciale identificare quando e come avvengono le collisioni tra l'entità controllata dal giocatore (MeatBoy) e gli elementi circostanti, come piattaforme, seghe e Bandage Girl.

   Soluzione La classe CollisionChecker implementa un sistema di collision detection attraverso il metodo isColliding(). Utilizzando le hitbox di MeatBoy, piattaforme, seghe e Bandage Girl, la classe verifica se ci sono intersezioni tra le diverse hitbox, aggiornando di conseguenza lo stato di collisione (CollisionState). Questo approccio fornisce un meccanismo robusto per gestire le collisioni nel contesto del gioco.

2. **Gestione dello Stato** *Problema* Nel corso del gioco, è necessario gestire diversi stati del personaggio di gioco (MeatBoy) in base alle interazioni spaziali, come essere a contatto con il terreno, una parete, una sega o la Bandage Girl.

Soluzione La classe CollisionChecker implementa lo stato del personaggio attraverso la variabile state di tipo CollisionState. La gestione accurata degli stati avviene nel metodo isColliding(), dove vengono identificati e aggiornati gli stati in base alle collisioni rilevate. Questa struttura consente una gestione chiara e modulare degli stati del personaggio all'interno del gioco.

3. **Movimento del Personaggio:** *Problema* È necessario controllare e gestire il movimento del personaggio di gioco (MeatBoy) in risposta agli input dell'utente, evitando movimenti impossibili dovuti a collisioni con ostacoli o limiti di gioco.

   Soluzione La classe CollisionChecker gestisce il movimento del personaggio attraverso i metodi horizontalCollision() e verticalCollision(). Questi metodi controllano e aggiornano la posizione di MeatBoy in base agli input dell'utente, assicurandosi che il movimento sia possibile solo se non ci sono collisioni con pareti, piattaforme o limiti del gioco.

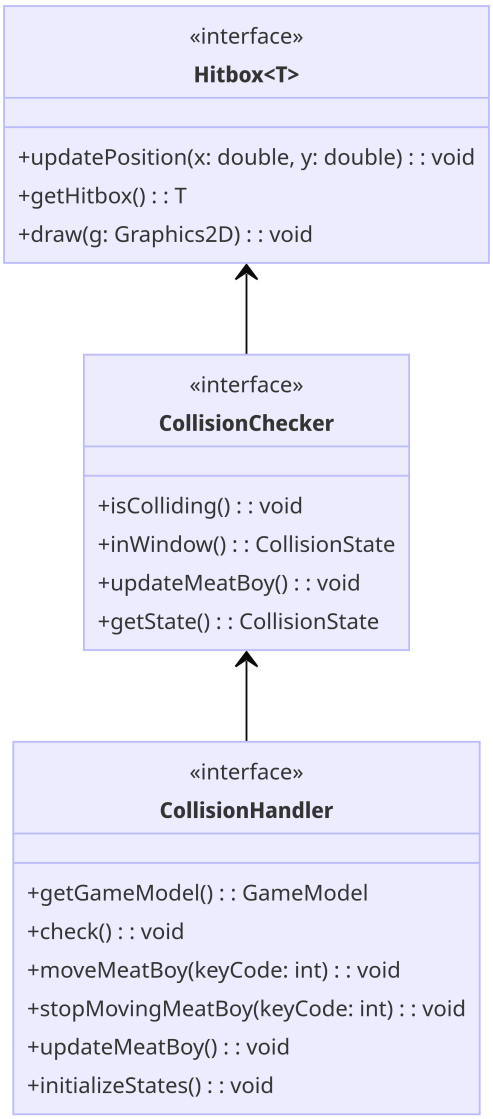

Figura 2.3: Schema UML dell'architettura

La Classe CollisionHandler
La classe CollisionCheckerImpl implementa l'interfaccia CollisionChecker ed è responsabile di verificare le collisioni tra diverse hitbox nel modello di gioco. Gestisce anche lo stato di collisione di un personaggio di gioco (MeatBoy), aggiornando la posizione in base alle interazioni.

Problemi risolti

1. **Gestione delle Collisioni Globali**
   Problema Un problema significativo è rappresentato dalla necessità di gestire le collisioni in modo centralizzato per l'intero modello di gioco. È essenziale determinare le azioni da intraprendere in base allo stato di collisione per garantire un comportamento coerente e corretto del gioco.

   Soluzione La classe CollisionHandler funge da componente centrale per la gestione delle collisioni globali. Essa utilizza un'istanza di CollisionCheckerImpl per verificare le collisioni, e successivamente, in base allo stato rilevato, aggiorna il modello di gioco attraverso il metodo check(). Questo approccio centralizzato semplifica la logica di gestione delle collisioni, rendendola più coesa ed estensibile.

2. **Aggiornamento del Modello**

   Problema La gestione delle collisioni non riguarda solo la rilevazione, ma anche l'aggiornamento del modello di gioco in risposta a eventi come la morte del personaggio (MeatBoy) o la vittoria raggiungendo la Bandage Girl.

   Soluzione La classe CollisionHandler affronta questo problema integrando la logica di aggiornamento del modello direttamente nel metodo check(). Quando rileva una collisione significativa, come la caduta fuori dallo schermo o il contatto con la Bandage Girl, esegue le azioni corrispondenti, ad esempio reimposta la posizione di MeatBoy o segnala la vittoria.

3. **Integrazione con il Pattern MVC**

   Problema Un aspetto critico è garantire l'integrazione efficiente della gestione delle collisioni con il pattern architetturale MVC, dove il CollisionHandler deve agire in armonia con il Model.

   Soluzione La classe CollisionHandler è progettata in modo da fungere da ponte tra il CollisionCheckerImpl e il modello di gioco (GameModel). Essa implementa l'interfaccia CollisionHandler, collegando il Controller (CollisionHandler) al Model (GameModel). Questa architettura è conforme al principio di separazione delle responsabilità del pattern MVC, dove il Controller gestisce gli input e le azioni, mentre il Model rappresenta lo stato del gioco. L'integrazione efficiente di queste componenti contribuisce a una progettazione modulare e ben strutturata.

2.2.2 Nicola Graziotin

La classe Entity

La classe `EntityImpl` implementa l'interfaccia `Entity` e fornisce un'implementazione generica di un'entità in un contesto di gioco o di simulazione. Essa offre supporto per le hitbox di tipo generico `H`, permettendo così la flessibilità nella definizione di vari tipi di hitbox.

Problemi Potenziali

1. **Immutabilità delle coordinate** Le coordinate dell'entità sono modificate direttamente dai metodi `setX` e `setY`, senza alcun controllo. Questo potrebbe causare inconsistenze nel sistema se le coordinate vengono modificate senza aggiornare correttamente la hitbox associata.

2. **Incapsulamento delle hitbox** La hitbox è accessibile direttamente dall'esterno attraverso il metodo `getHitbox()`, esponendo così la struttura interna dell'entità e violando il principio di incapsulamento.

Soluzioni

1. **Controllo delle modifiche alle coordinate** Per garantire la coerenza del sistema, potremmo aggiungere controlli all'interno dei metodi `setX` e `setY` per verificare se le nuove coordinate sono valide e, se necessario, aggiornare la hitbox associata.

2. **Incapsulamento della hitbox** Per mantenere l'incapsulamento, potremmo considerare l'eliminazione del metodo `getHitbox()` dall'interfaccia `Entity` e rendere la hitbox privata, fornendo invece metodi appropriati all'interno di `EntityImpl` per interagire con essa.

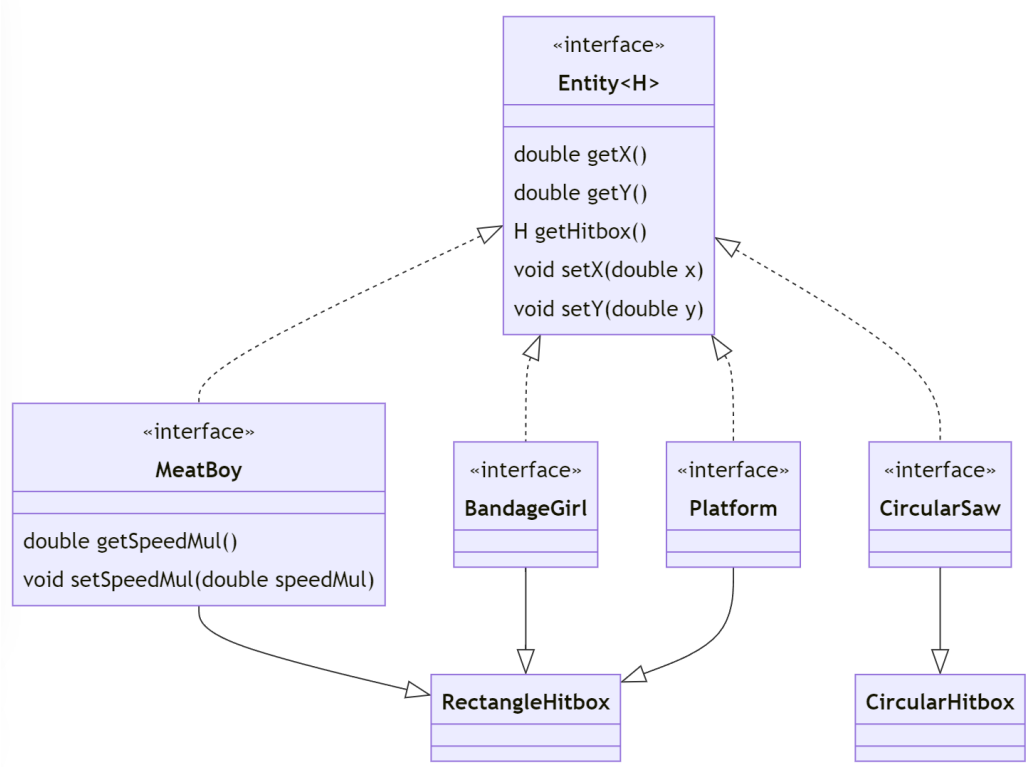

Figura 2.4: Schema UML di Entity

La classe MeatBoy

La classe `MeatBoyImpl` rappresenta un'implementazione specifica della classe `EntityImpl` per l'entità MeatBoy. Estende `EntityImpl` e specifica una hitbox di tipo `RectangleHitbox`, implementando così l'interfaccia `MeatBoy`.

Problemi Potenziali

1. **Dipendenza da Costanti Globali** La classe utilizza costanti globali per definire valori come la velocità, l'altezza massima del salto e la velocità di caduta. Questo potrebbe rendere la classe meno flessibile e difficile da testare e mantenere.

2. **Incapsulamento delle Variabili** La variabile `speedMul` è accessibile direttamente dall'esterno attraverso il metodo `getSpeedMul()` e `setSpeedMul()`, violando così il principio di incapsulamento.

Soluzioni

1. **Iniezione delle Dipendenze** Per ridurre la dipendenza dalle costanti globali, potremmo considerare l'iniezione delle dipendenze attraverso il costruttore, in modo che i valori possano essere passati alla classe anziché essere definiti all'interno di essa.

2. **Incapsulamento delle Variabili** Per mantenere l'incapsulamento delle variabili, potremmo rendere privata la variabile `speedMul` e fornire metodi pubblici per accedere e modificare il suo valore, garantendo così un maggiore controllo sull'accesso e l'aggiornamento della variabile.

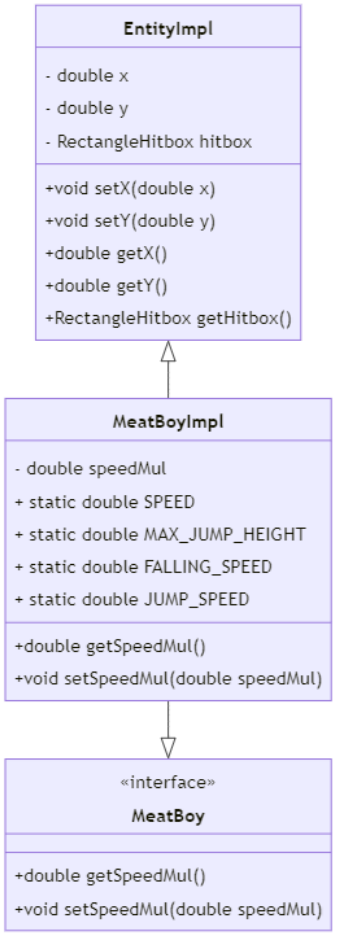

Figura 2.5: Schema UML di MeatBoy

La classe GamePanel

La classe `GamePanel` rappresenta il pannello principale per il rendering degli elementi di gioco. Estende `JPanel` e fornisce metodi per disegnare e gestire l'input da tastiera.

Problemi Potenziali

1. **Utilizzo di campi transient** L'uso di campi transient per immagini e controller potrebbe causare problemi in caso di serializzazione dell'oggetto `GamePanel`, ad esempio durante la gestione del salvataggio e del ripristino dello stato del gioco.

2. **Mancanza di modularità** La classe potrebbe diventare troppo grande e difficile da mantenere se non divisa in moduli più piccoli e gestibili, aumentando il rischio di errori e rendendo complicato il testing.

3. **Manutenzione dell'interfaccia utente** La gestione dell'interfaccia utente all'interno della classe stessa potrebbe rendere il codice difficile da leggere e mantenere, specialmente se l'interfaccia utente dovesse cambiare o essere estesa in futuro.

4. **Conflitti di gestione degli eventi** L'implementazione della gestione degli eventi potrebbe diventare complessa man mano che il gioco si sviluppa e richiede più interazioni da parte dell'utente.

Soluzioni

1. **Gestione delle immagini e del controller** Per risolvere i potenziali problemi di serializzazione, si potrebbe considerare l'utilizzo di metodi per caricare le immagini e iniettare il controller, anziché mantenerli come campi transient.

2. **Suddivisione in moduli** La classe potrebbe essere suddivisa in moduli più piccoli e gestibili, seguendo i principi di progettazione dei software modulare, come ad esempio il principio di singola responsabilità, in modo da rendere più facile la manutenzione e l'estensione del codice.

3. **Separazione della logica di UI** La logica dell'interfaccia utente potrebbe essere separata dalla classe `GamePanel`, ad esempio utilizzando un design pattern come Model-View-Controller (MVC), che separa la presentazione dall'interazione con il modello di dati sottostante.

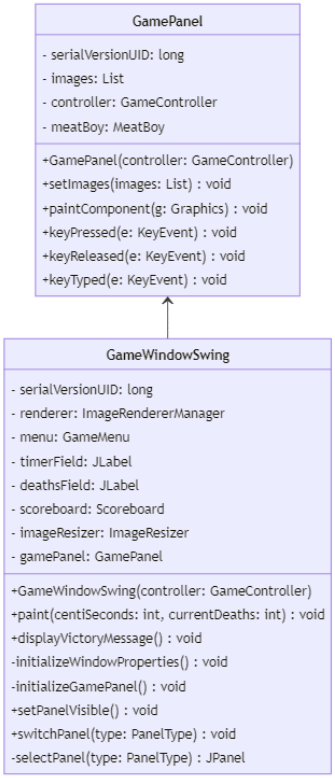

Figura 2.6: Schema UML del GamePanel

4. **Utilizzo di un design pattern per la gestione degli eventi** L'implementazione di un pattern come Observer potrebbe semplificare la gestione degli eventi nel gioco, consentendo a diverse parti del sistema di osservare e rispondere agli eventi senza dipendenze dirette tra loro.

La classe GameController

La classe `GameControllerImpl` implementa l'interfaccia `GameController`, fornendo la logica di controllo per il gioco. In questa relazione, esamineremo i problemi potenziali derivanti dall'implementazione di questa classe e le soluzioni apportate per mitigarli.

Problemi Potenziali

1. **Accoppiamento Stretto** Uno dei potenziali problemi è l'accoppiamento stretto tra `GameControllerImpl` e `GameModel`. Questo potrebbe

rendere difficile il testing e la manutenzione in futuro, poiché modifiche al `GameModel` potrebbero richiedere modifiche anche al `GameControllerImpl`.

2. **Dipendenza da Timer** L'uso di `Timer` e `TimerTask` potrebbe portare a problemi di sincronizzazione e prestazioni inefficaci. Inoltre, potrebbe essere difficile controllare e gestire i tempi di gioco in modo accurato.

3. **Gestione degli Eventi** La gestione degli eventi potrebbe diventare complessa man mano che il gioco si sviluppa e richiede più interazioni tra utente e gioco.

4. **Mancanza di Modularità** La classe potrebbe diventare troppo grande e difficile da mantenere se non divisa in moduli più piccoli e gestibili.

Soluzioni

1. **Riduzione dell'Accoppiamento** Per ridurre l'accoppiamento stretto, è possibile introdurre un'astrazione intermedia tra `GameControllerImpl` e `GameModel`, come ad esempio un'interfaccia `GameModel`, che consentirebbe di sostituire facilmente l'implementazione del `GameModel` senza modificare il `GameControllerImpl`.

2. **Miglioramento della Gestione del Tempo** Al posto di `Timer` e `TimerTask`, potremmo considerare l'utilizzo di framework più avanzati per la gestione del tempo, come ad esempio `ScheduledExecutorService`, che offre maggiore flessibilità e controllo sulle attività pianificate.

3. **Utilizzo di un Design Pattern per la Gestione degli Eventi** L'implementazione di un pattern come Observer potrebbe semplificare la gestione degli eventi nel gioco, consentendo a diverse parti del sistema di osservare e rispondere agli eventi senza dipendenze dirette tra loro.

4. **Suddivisione in Moduli** La classe potrebbe essere suddivisa in moduli più piccoli e gestibili, seguendo i principi di progettazione dei software modulare, come ad esempio il principio di singola responsabilità, in modo da rendere più facile la manutenzione e l'estensione del codice.

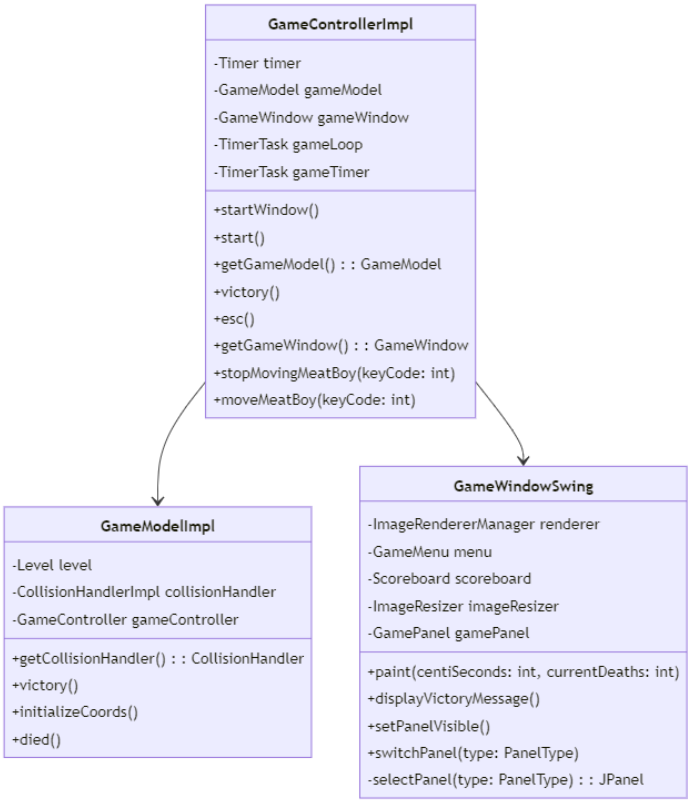

Figura 2.7: Schema UML del GameController

2.2.3 Mattia Galli

Sono state progettate varie classi per la realizzazione del progetto, per quanto riguarda gli obstacles del livello sono state realizzate la classe CircularSaw e Platform, con le loro rispettive interfacce; l'interfaccia Statistic per memorizzare in numero totale delle morti e il record di tempo per superare il livello e le classi GameMenu e Scoreboard;
Prenderemo in esame le classi GameMenu, Scoreboard e Statistic.

- **classe GameMenu**
  La classe GameMenu è un componente della View, rappresenta la raffigurazione grafica del pannello iniziale del gioco, dove l'utente può iniziare a giocare, visualizzare le proprie statistiche oppure uscire dall'applicazione.

– Problema
Il primo problema della realizzazione del GameMenu è stato il bisogno di dover cambiare pannello alla premuta del specifico tasto.

– Soluzione
La soluzione è stata quella di, tramite il controller, richiamare un metodo del GameWindow che avesse il compito di cambiare il pannello in base al tasto selezionato, switchPanel avrà il compito di settare il pannello corrente, richiedere il focus e renderlo visibile.

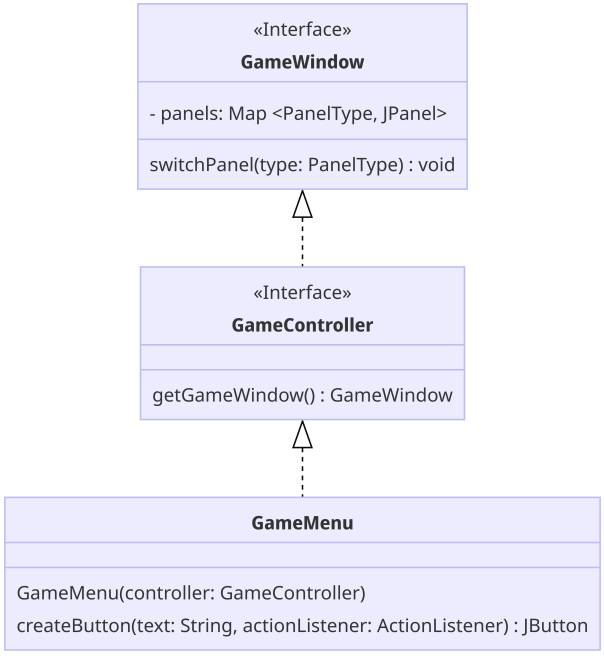

Figura 2.8: Schema UML dell'architettura

- **classe Scoreboard**
La classe Scoreboard rappresenta la raffigurazione grafica del pannello con le morti totali del giocatore e il suo tempo migliore per portare il personaggio al "traguardo". Scoreboard utilizza la classe Statistic per aggiornare le morti e il record richiamando due getter del costruttore,

getDeaths e getTimeRecord.

- Problema
  Un problema per la realizzazione della Scoreboard è stato il bisogno di aggiornare ogni volta le morti e il record in modo dinamico mostrandole a schermo nel panel.

- Soluzione
  La risoluzione è stata aggiungere un metodo addNotify che verifica che la Scoreboard venga inserita come pannello principale e richiama i metodi updateDeaths, updateTimeRecord e il repaint per aggiornare ogni volta l'interfaccia.

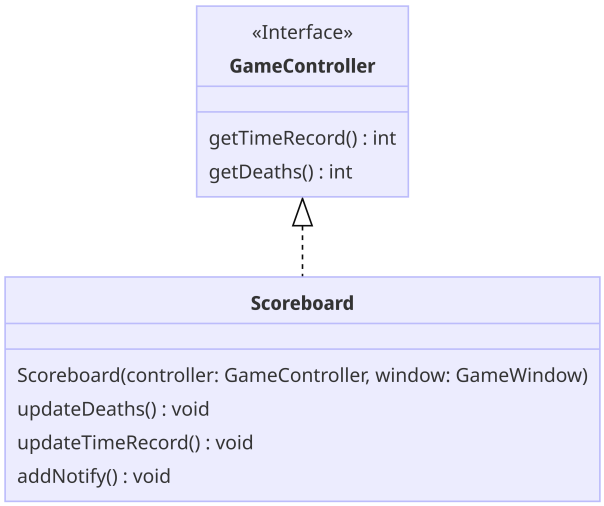

Figura 2.9: Schema UML dell'architettura

- **classe Statistic**
  La classe Statistic viene utilizzata all'interno del controller è aggiunge le morti al counter ogni volta che l'utente muore cadendo o collidendo una sega rotante; altra funzione della classe è quella di aggiornare il

record ogni volta che l'utente fa un tempo migliore.

- Problema
  Incrementare il contatore delle morti totali ogni volta che il personaggio collide con un ostacolo o cade dalle piattaforme.

- Soluzione
  La soluzione è stata creare un metodo addDeaths che viene richiamato dal controller, ogni volta che rileva una collision Saw oppure una Fall, il contatore viene incrementato di uno.

- Note
  La soluzione scelta può essere estesa, sovrascrivendo i dati in un file e quindi avendo il contatore sempre aggiornato anche dall'uscita dall'applicazione, cosa che al momento non è possibile fare.

- Problema
  Sovrascrivere il tempo migliore impiegato dall'Utente per finire il livello.

- Soluzione
  La soluzione è stata creare un metodo updateRecord che viene richiamato dal controller nel metodo victory, ogni volta che il traguardo viene raggiunto, termina il game loop e se il tempo è minore del record, allora viene sovrascritto.

- Note
  La soluzione scelta può essere estesa, sovrascrivendo i dati in un file e quindi avendo il tempo del record sempre aggiornato anche dall'uscita dall'applicazione, cosa che al momento non è possibile fare.

2.2.4 Martin Tomassi

- **Estrazione di Informazioni dalla Mappa di Gioco da Documenti TMX**

- Problema:
  Si desidera implementare un sistema per estrarre informazioni direttamente, riguardanti la mappa di gioco, da documenti TMX (Tile Map XML).
- Soluzione:
  Il sistema si basa su DocumentExtractor che si occupa di definire i comportamenti di estrazione di informazioni, identificati da Tag, dalla mappa di gioco presente nel documento TMX. Il Factory Method DocumentExtractorFactory fornisce un meccanismo per creare istanze di DocumentExtractor in modo flessibile, configurabile e incapsulato.

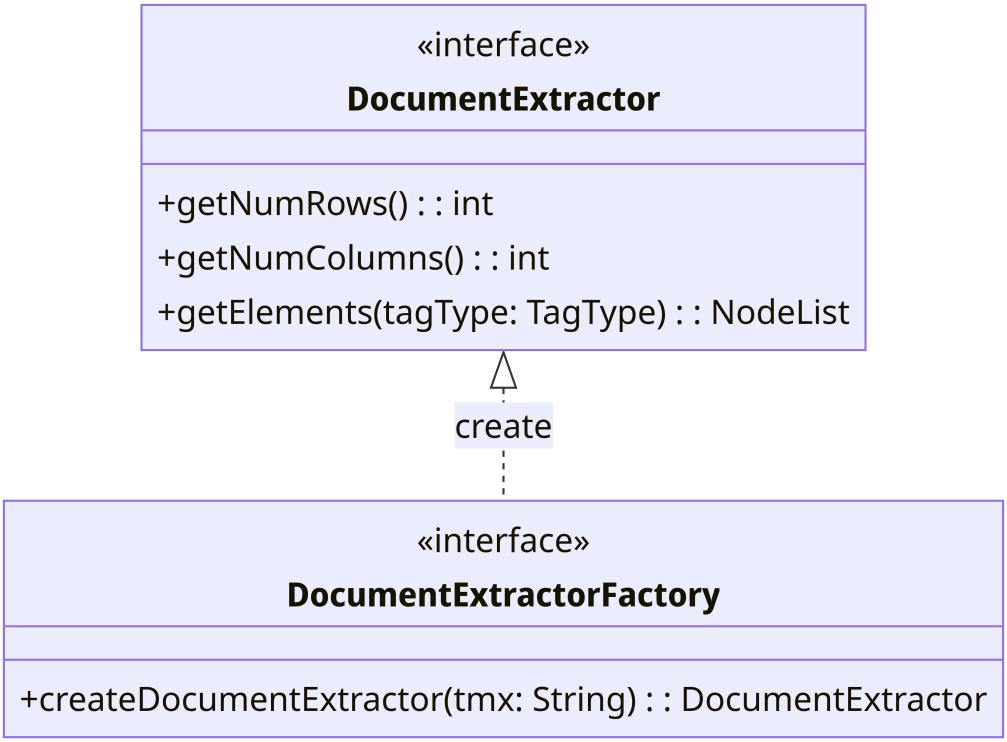

Figura 2.10: Schema UML di DocumentExtractor

- **Creazione di TileSet da Immagini Composite: Estrazione Efficiente di Tile individuali da Spritesheet**
  - Problema:
    Si desidera implementare un sistema che si occupa di creare TileSet da Immagini Composite per creare un set di Tile utilizzabili.

– Soluzione:
Il Sistema si basa su TileSet e DocumentExtractor che lavorano in sintonia per la la scelta e suddivisione delle immagini in Tile. I Factory Method TileSetFactory e DocumentExtractorFactory forniscono un meccanismo per creare istanze di TileSet e DocumentExtractor in modo flessibile e configurabile.

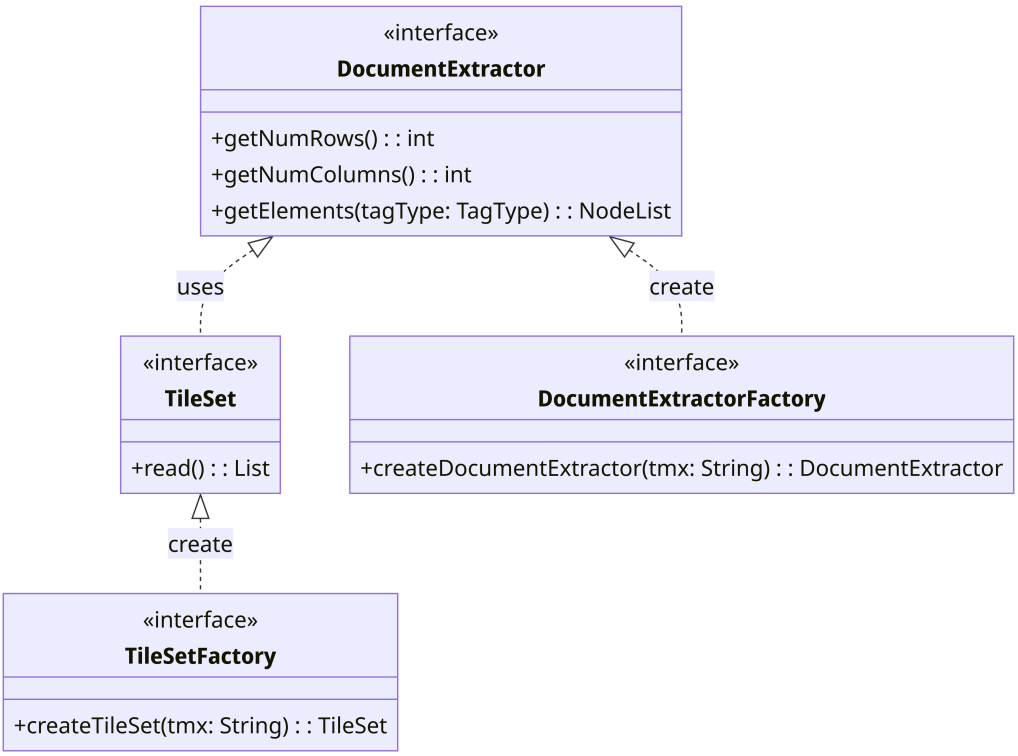

Figura 2.11: Schema UML di TileSet

- **Gestione Efficiente del Caricamento e della Gestione di Oggetti e Tiles da un documento TMX**

  – Problema:
  Si desidera implementare un sistema che si occupa di caricare i Tiles che compongono la mappa e i gruppi di oggetti di gioco dal documento TMX (Tile Map XML) di riferimento, in modo efficiente e modulare.

  – Soluzione
  Il sistema si basa sul TileLoaderManager, composto da TileLoaderGameObjects e TileLoaderStationary, che si occupa di gestire il

corretto caricamento dei tiles e degli oggetti di gioco, delegando il caricamento diretto ai suoi delegati. I Factory Method TileLoaderGameObjectsFactory e TileLoaderStationaryFactory permettono di creare istanze concrete di caricatori di oggetti e tiles. Il Factory Method TileLoaderManagerFactory fornisce un'istanza del gestore di caricamento. Questo sistema si occuperá di comunicare direttamente con il TileManager, successivamente spiegato.

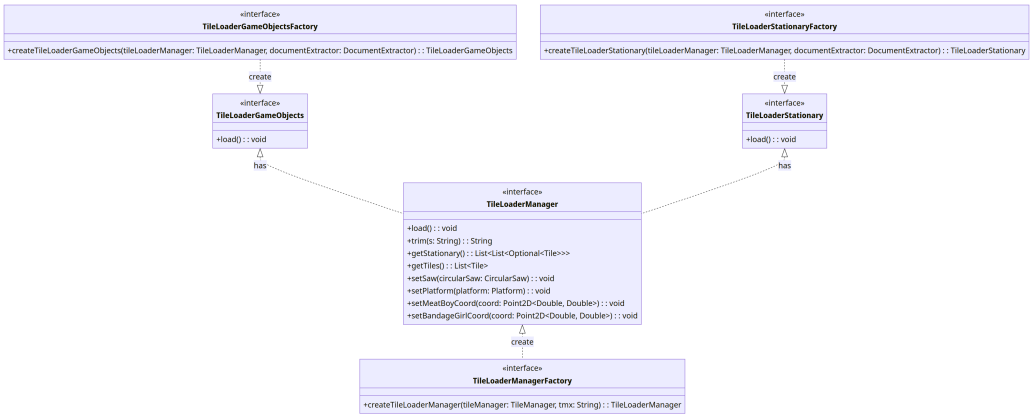

Figura 2.12: Schema UML di TileLoader

- **Gestione e Organizzazione dei Dati di Livello e Tile in un Gioco: Soluzione per la Manipolazione e l'Accesso Efficienti**
  - Problema:
    Si desidera implementare un sistema che si occupa della gestione e l'organizzazione di un livello di gioco basato su una mappa TMX. Questo include la manipolazione dei vari elementi della mappa, nonché la gestione della disposizione spaziale dei tile nella mappa e l'accesso alle informazioni relative ai personaggi e agli elementi presenti nel livello. In sostanza, un sistema responsabile di garantire che il livello di gioco sia correttamente rappresentato e funzionante all'interno dell'ambiente di gioco.
  - Soluzione
    Il sistema si occupa della gestione e dell'organizzazione di un livello di gioco basato su una mappa TMX. Questo comprende tutte le attività necessarie per rendere il livello giocabile e interattivo, tra cui: La gestione dei tile attraverso TileManager, che include il recupero delle informazioni sui vari elementi della mappa come piattaforme, seghe circolari e personaggi giocanti. La creazione

e la gestione del livello di gioco attraverso Level, che include il recupero delle informazioni sui personaggi giocanti come MeatBoy e BandageGirl, nonché sulle piattaforme, sulle seghe circolari e sulla disposizione dei tile nella mappa tramite TileManager. Il Level si occupa di comunicare con il sistema dedicato alle Collisioni con Hitbox: questa é la motivazione per cui viene creata una Classe come punto unico di Comunicazione con Classi esterne al Livello specificato. Inoltre, ciascun Factory Method fornisce un meccanismo per creare istanze in modo flessibile, configurabile e incapsulato.

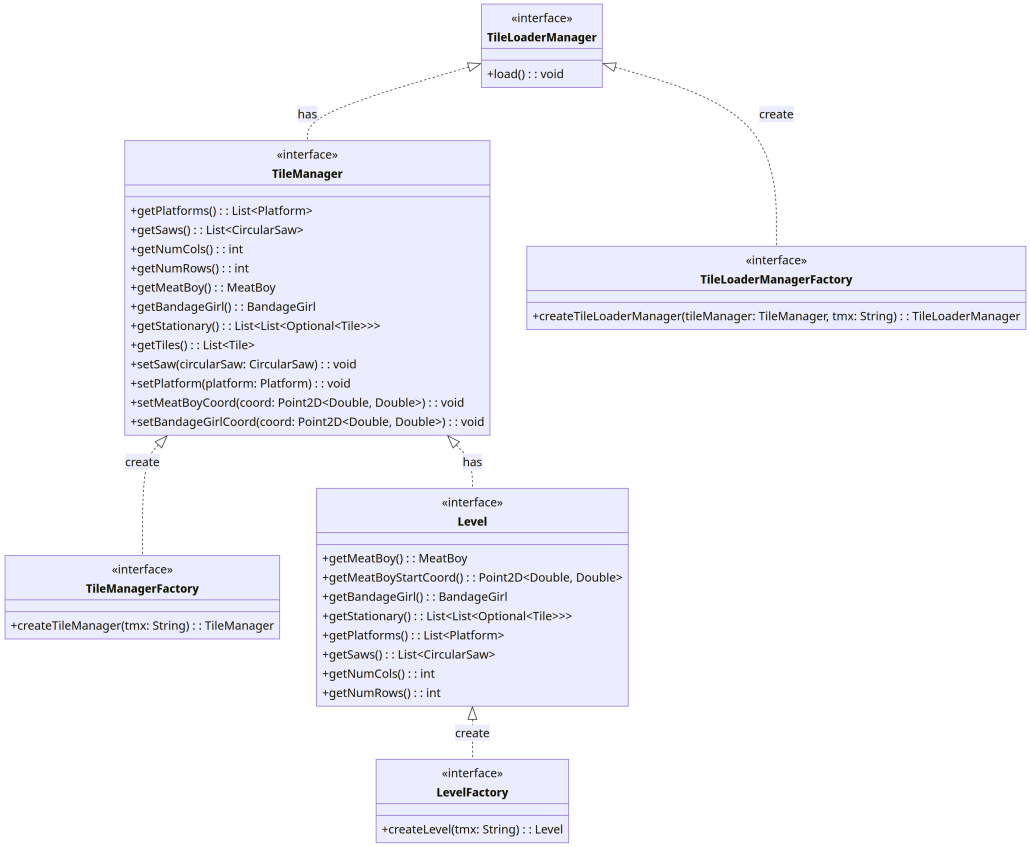

Figura 2.13: Schema UML di TileManager e Level

- **Rendering e Gestione delle Immagini nella GUI del Videogioco basato su Tile**
  - Problema:
    Si desidera implementare un sistema che si occupa della visualiz-

zazione grafica del videogioco basato su Tiles.

– Soluzione
Il sistema si occupa di caricare le immagini necessarie per il gioco, renderizza le seghe circolari, le piattaforme e i personaggi nella mappa di gioco e gestisce il processo di rendering complessivo tramite ImageRendererManager, che si occupa di coordinare l'uso tra delegati per generare una lista di immagini che rappresentano lo stato visuale attuale del gioco. Inoltre, ciascun Factory Method fornisce un meccanismo per creare istanze in modo flessibile, configurabile e incapsulato.

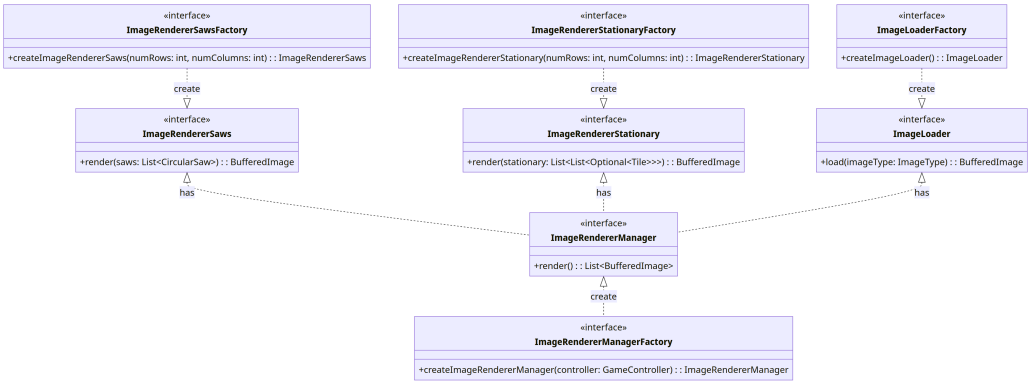

Figura 2.14: Schema UML di ImageRenderer

- **Ridimensionamento Scalabile delle Immagini della GUI per una Visualizzazione Ottimale su Diverse Risoluzioni**

  – Problema:
  Si desidera implementare un sistema che é in grado di adattare le dimensioni delle immagini della GUI in modo dinamico per garantire una corretta visualizzazione su schermi con risoluzioni diverse. In altre parole, consente di rendere le immagini della GUI proporzionali allo schermo dell'utente che sta giocando, migliorando così l'esperienza utente e assicurando che l'interfaccia grafica del gioco sia ben adattata a una varietà di dispositivi e risoluzioni.

  – Soluzione
  Il sistema si occupa di ridimensionare le immagini della GUI in base alle dimensioni dello schermo dell'utente che avvia il gioco. Quando il gioco viene avviato su schermi con risoluzioni diverse, le dimensioni delle immagini della GUI vengono adattate per

mantenere proporzioni appropriate e garantire una corretta visualizzazione. Il ImageResizer potrebbe essere chiamato per eseguire questo ridimensionamento dinamico delle immagini in base alle dimensioni dello schermo dell'utente. Inoltre, il Factory Method fornisce un meccanismo per creare istanze in modo flessibile, configurabile e incapsulato.

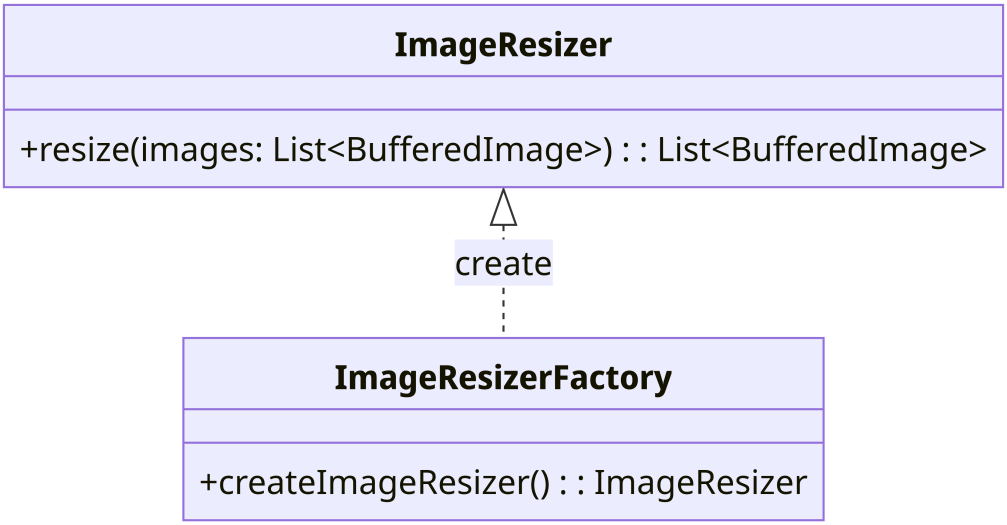

Figura 2.15: Schema UML di ImageResizer

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Durante il processo di sviluppo del software, abbiamo dedicato tempo e risorse per garantire che tutte le funzionalità essenziali fossero implementate senza errori. Abbiamo adottato un approccio automatizzato per il testing, utilizzando il framework JUnit 5. I test manuali sono stati limitati alla verifica del corretto funzionamento del programma su diverse piattaforme operative. Di seguito, riportiamo un elenco dettagliato dei test automatici sviluppati da ciascun membro del team:

3.1.1 Francesco Marcatelli

- **Hitbox**

  Obiettivo: L'obiettivo dei test descritti è verificare la corretta implementazione delle hitbox per diverse entità di gioco, tra cui MeatBoy, BandageGirl, Platform e CircularSaw. I test si concentrano sul confronto delle dimensioni e posizioni attese delle hitbox rispetto alle posizioni e dimensioni effettive delle entità di gioco. Inoltre, viene verificato se le modifiche alle posizioni delle entità influenzano correttamente le corrispondenti hitbox.

- **Collision**

  Obiettivo: L'obiettivo dei test è verificare la corretta gestione delle collisioni nel contesto del gioco. Si focalizzano sulla collisione di MeatBoy con vari ostacoli, come il terreno, le seghe circolari e

BandageGirl. Ci si assicura che si gesticano correttamente gli stati, assicurando un comportamento coeso e preciso nelle dinamiche di collisione del gioco.

3.1.2 Nicola Graziotin

- **Entity**

  Obiettivo: L'obiettivo dei test è verificare che le implementazioni delle entità nel gioco (MeatBoy, BandageGirl, CircularSaw e Platform) rispettino correttamente la logica di assegnazione delle hitbox. In particolare, ogni test confronta il tipo di hitbox restituito dalle entità con il tipo di hitbox atteso, per verificare la corretta creazione delle entità.

3.1.3 Mattia Galli

- **Statistic**

  Obiettivo: I test sono progettati per verificare il corretto funzionamento dei metodi per incrementare il contatore delle morti e l'aggiornamento del record del miglior tempo se questo viene battuto.

3.1.4 Martin Tomassi

- **GameController**

  Obiettivo: I test sono progettati per verificare il corretto funzionamento dei metodi di accesso e recupero delle informazioni relative al controller di gioco, come le tile stazionarie, le seghe circolari, le piattaforme, i personaggi di gioco (MeatBoy e BandageGirl) e le dimensioni della mappa.

- **GameModel**

  Obiettivo: I test sono progettati per verificare il corretto funzionamento dei metodi di accesso e recupero delle informazioni relative al model di gioco, come le tile stazionarie, le seghe circolari, le piattaforme, i personaggi di gioco (MeatBoy e BandageGirl), le dimensioni della mappa e l'handler delle collisioni.

- **DocumentExtractor**

Obiettivo: I test sono progettati per verificare il corretto funzionamento dei metodi di accesso e recupero delle informazioni estratte dal documento, come gli elementi di diversi tipi (object, object group, tile), nonché il numero di righe e colonne nel documento.

- **Level**

  Obiettivo: I test sono progettati per verificare il corretto funzionamento dei metodi di accesso e recupero delle informazioni relative al livello, come la posizione dei personaggi di gioco (MeatBoy e BandageGirl), le piattaforme, le seghe circolari e le informazioni sulle tile stazionarie.

- **TileManager**

  Obiettivo: I test sono progettati per verificare il corretto funzionamento del TileManager e delle sue funzionalità, inclusa l'inizializzazione, l'accesso alle informazioni sulle tile, il recupero delle piattaforme, delle seghe, delle tile stazionarie e dei personaggi di gioco.

- **TileLoader**

  Obiettivo: I test sono progettati per verificare il corretto caricamento dei dati da parte del gestore di caricamento delle tile (TileLoaderManager) e dei suoi "workers" (TileLoaderGameObjects e TileLoaderStationary), garantendo che le tile stazionarie ed oggetti di gioco vengano caricate correttamente insieme ai loro attributi, come ad esempio le dimensioni della griglia e la presenza di oggetti come seghe e piattaforme.

- **TileSet**

  Obiettivo: Il test è progettato per verificare il corretto funzionamento della lettura da parte del TileSet, il quale legge e restituisce un elenco di tile dal/dai sorgente/i specificato/i.

- **Tile**

  Obiettivo: I test sono progettati per verificare il corretto funzionamento delle funzionalità di base della classe TileImpl, comprese la restituzione delle coordinate X e Y della cella e il percorso dell'immagine associata ad essa.

3.2 Note di sviluppo

3.2.1 Francesco Marcatelli

- **Utilizzo di Stream e Lambda expressions**

  Usate in vari punti. Il seguente é un singolo esempio. Permalink: `https://github.com/NicolaGraziotin/OOP23-smb-clone/blob/22a49a1857750243ab7364bd9b58e800165711fc/src/main/java/it/unibo/model/collision/CollisionCheckerImpl.java#L41`

3.2.2 Nicola Graziotin

- **Utilizzo di classe generica**

  Usato una classe generica per la realizzazione delle entità. Permalink: `https://github.com/NicolaGraziotin/OOP23-smb-clone/blob/d93b5568d7be3baf1b68cfb6adb6f97c8ba1c1c7/src/main/java/it/unibo/model/entity/EntityImpl.java#L13`

3.2.3 Mattia Galli

- **Utilizzo di Lambda expressions**

  Usate nella realizzazione dei button. Il seguente é un singolo esempio. Permalink: `https://github.com/NicolaGraziotin/OOP23-smb-clone/blob/73a629a383bf8b21a40d2c58be3fe6793fc3f630/src/main/java/it/unibo/view/panel/GameMenu.java#L64`

3.2.4 Martin Tomassi

- **Utilizzo di Stream e Lambda expressions**

  Usate pervasivamente. Il seguente é un singolo esempio. Permalink: `https://github.com/NicolaGraziotin/OOP23-smb-clone/blame/22a49a1857750243ab7364bd9b58e800165711fc/src/main/java/it/unibo/model/tiles/loader/gameobjects/TileLoaderGameObjectsImpl.java#L73`

- **Utilizzo di librerie per il Parsing XML**

  Usate pervasivamente. Il seguente é un singolo esempio. Permalink: `https://github.com/NicolaGraziotin/OOP23-smb-clone/blob/`

22a49a1857750243ab7364bd9b58e800165711fc/src/main/java/it/unibo/model/documentextractor/DocumentExtractorImpl.java#L36

- **Utilizzo di Optional**

  Utilizzato in vari punti. Il seguente é un singolo esempio. Permalink: https://github.com/NicolaGraziotin/OOP23-smb-clone/blob/22a49a1857750243ab7364bd9b58e800165711fc/src/main/java/it/unibo/model/tiles/manager/TileManagerImpl.java#L131

Le risorse presenti nella cartella `src/main/resources` sono state acquisite dall'indirizzo https://github.com/danielpygo/Supermeatboy/tree/master/SupermeatBoy/resources.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Francesco Marcatelli

- **Punti di Forza:**

  1. Contributi al Codice: Nella realizzazione del progetto ho svolto la parte di gestione e creazione delle collisioni. Ho contribuito attivamente sia nella parte di ideazione e decision making, sia nella parte di revisione del codice sorgente.
  2. Capacità di Problem Solving: Ho dimostrato abilità nel risolvere sfide tecniche in modo efficiente, contribuendo al superamento di ostacoli durante lo sviluppo.
  3. Collaborazione: La collaborazione con gli altri membri del team è stata evidente attraverso partecipazione attiva alle discussioni e ai processi decisionali, il che ha migliorato il progetto nel suo complesso.

- **Punti di Debolezza:**

  1. Testing: Possibilità di perfezionare l'adozione di una strategia di testing più efficace, mirata a garantire una maggiore solidità del software.

- **Lavori Futuri:**

  1. Miglioramento delle Funzionalità: Possiamo migliorare la grafica sia dalla parte di menù che dalla parte grafica del gioco stes-

so, aggiungendo animazioni al movimento del personaggio o agli ostacoli, per rendere l'esperienza agli utenti ancora migliore.

2. Ottimizzazione del codice: Rivedere e ottimizzare il codice esistente per renderlo più efficiente e manutenibile.

4.1.2 Nicola Graziotin

- **Punti di Forza:**

  1. Contributi al Codice: Ho contribuito al progetto occupandomi della parte della creazione delle entità come il personaggio principale e il personaggio che rappresenta il traguardo. Mi sono occupato inoltre della creazione del pannello in cui si visualizza il gioco e il controller che mette in comunicazione la view e il model.
  2. Capacità di Problem Solving: Ho dimostrato di essere in grado di affrontare con successo sfide tecniche.
  3. Collaborazione: Ho collaborato efficacemente con gli altri membri del team, partecipando attivamente alle discussioni e ai processi decisionali per migliorare il progetto.
  4. Rispetto delle Scandenze: Ho lavorato basandomi sul rispetto delle scadenze in modo da consegnare il progetto nei tempi concordati.

- **Punti di Debolezza:**

  1. Testing: Posso migliorare nell'implementare una strategia di testing più efficace al fine di garantire una maggiore robustezza del software.
  2. Apprendimento Continuo: Sono consapevole della necessità di continuare a crescere professionalmente e di acquisire nuove competenze nel campo dello sviluppo software.

- **Lavori Futuri:**

  1. Miglioramento delle Funzionalità: Possiamo espandere il progetto introducendo nuove funzionalità, come nuovi livelli di gioco, personaggi aggiuntivi o modalità di gioco alternative, per rendere l'esperienza più coinvolgente per gli utenti.
  2. Potremmo lavorare sull'ottimizzazione del codice attuale al fine di potenziare le prestazioni del gioco, abbreviando i tempi di caricamento, ottimizzando l'allocazione delle risorse di sistema e migliorando la fluidità delle animazioni.

3. Ottimizzazione del codice: Possiamo rivedere e ottimizzare il codice esistente per renderlo più efficiente e ottimizzato.

4.1.3 Mattia Galli

- **Punti di Forza:**

  1. Contributi al Codice: Ho svolto la mia parte nel progetto, occupandomi della parte grafica dei pannelli e i metodi a loro affiliati nel controller e nel gameModel, implementando le classi Statistic e fornendo una base per le classi obstacles.
  2. Capacità di apprendere nuove strategie: Ho appreso tramite la risoluzione di problemi nuovi mai affrontati prima, l'utilizzo di strategie più efficaci per la risoluzione di eventuali obiettivi.
  3. Collaborazione: Ho lavorato in collaborazione con il team per sviluppare codice in linea con gli obiettivi concordati e seguendo gli standard di codice del team.

- **Punti di Debolezza:**

  1. Efficacia: Potrei migliorare l'efficacia nella risoluzione dei problemi a me proposti e semplificare al meglio le loro soluzioni.

- **Lavori Futuri:**

  1. Miglioramento della Gui: Data una possibile espansione del progetto, possiamo adattare la gui per poter selezionare futuri livelli, scelta di personaggi o più semplicemente migliorare l'aspetto grafico dell'applicazione
  2. Nuovi Ostacoli: Potremmo creare nuovi ostacoli dinamici con difficoltà maggiori rispetto ai "pericoli" statici presenti al momento.

4.1.4 Martin Tomassi

- **Punti di Forza:**

  1. Contributi al Codice: Ho fornito un contributo significativo al codice del progetto, partecipando attivamente alla scrittura e alla revisione del codice sorgente, occupandomi del Rendering delle Immagini in modo scalabile e della creazione del livello tramite tiles.

2. Coerenza e Organizzazione: Ho lavorato per mantenere una struttura coerente e organizzata all'interno del codice, facilitando la comprensione e la manutenzione del progetto.
3. Capacità di problem solving: Ho dimostrato di essere in grado di affrontare sfide tecniche in modo efficace.
4. Collaborazione: Ho collaborato efficacemente con gli altri membri del team, partecipando alle discussioni e ai processi decisionali per migliorare il progetto.
5. Rispetto delle Scadenze: Ho rispettato le scadenze assegnate, contribuendo così alla consegna tempestiva del progetto.

- **Punti di Debolezza:**

1. Testing: Potrei migliorare nell'implementare una strategia più efficace per il testing al fine di garantire una maggiore robustezza del software.
2. Apprendimento Continuo: C'è sempre spazio per migliorare e imparare nuove tecniche e tecnologie. Sono consapevole della necessità di continuare a crescere professionalmente e di acquisire nuove competenze nel campo dello sviluppo software.

- **Lavori Futuri:**

1. Miglioramento delle Funzionalità: Possiamo espandere il progetto aggiungendo nuove funzionalità, come nuovi livelli di gioco, personaggi aggiuntivi o modalità di gioco alternative, per rendere l'esperienza più coinvolgente per gli utenti.
2. Ottimizzazione delle Prestazioni: Possiamo ottimizzare il codice esistente per migliorare le prestazioni del gioco, riducendo i tempi di caricamento, ottimizzando l'utilizzo delle risorse di sistema e migliorando la fluidità delle animazioni.
3. Ottimizzazione del codice: Rivedere e ottimizzare il codice esistente per renderlo più efficiente e manutenibile.

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Martin Tomassi

Ho apprezzato molto il contenuto del corso, ma ho trovato alcune lezioni un po' troppo dense e difficili da assimilare in breve tempo. Sarebbe utile poter organizzare sessioni di revisione durante le quali gli studenti possono porre domande e chiarire eventuali dubbi sui concetti trattati. Queste sessioni offrirebbero agli studenti l'opportunità di rivedere e consolidare i concetti in un ambiente collaborativo e interattivo. Inoltre, segnalo il fatto che durante il corso abbiamo affrontato solo superficialmente gli esercizi che coinvolgono la gestione di logiche "ricorsive" e la progettazione di metodi ausiliari per manipolarli. La mancanza di una copertura approfondita su questo argomento ha reso difficile affrontare al meglio l'esame. Sarebbe stato utile avere un maggior approfondimento su questi concetti durante il corso, con esempi pratici e spiegazioni dettagliate, per permettere agli studenti di acquisire una migliore comprensione e padronanza. Nonostante le lacune menzionate precedentemente, il Prof. Viroli è stato molto disponibile e paziente quando ho richiesto chiarimenti. Le sue spiegazioni e risposte alle mie domande hanno contribuito a migliorare la comprensione degli argomenti.

Appendice A

Guida utente

- **Controlli di Base:**
  1. A : Muovi il Personaggio a Sinistra
  2. D : Muovi il Personaggio a Destra
  3. Barra Spaziatrice : Salto
  4. Shift : Corsa
  5. ESC : Uscita Livello
  6. Il salto è consentito anche sulle pareti delle piattaforme
- **Obiettivo del Gioco:**
  Il tuo obiettivo principale è raggiungere Bandage Girl alla fine di ogni livello nel minor tempo possibile. Durante il livello dovrai evitare di cadere nel vuoto e di andare a sbattere con gli ostacoli.

Appendice B

Esercitazioni di laboratorio

Martin Tomassi

martin.tomassi@studio.unibo.it

- Laboratorio 06: `https://virtuale.unibo.it/mod/forum/discuss.php?d=146511#p208417`
- Laboratorio 07: `https://virtuale.unibo.it/mod/forum/discuss.php?d=147598#p209297`
- Laboratorio 08: `https://virtuale.unibo.it/mod/forum/discuss.php?d=148025#p210279`
- Laboratorio 09: `https://virtuale.unibo.it/mod/forum/discuss.php?d=149231#p211464`
- Laboratorio 10: `https://virtuale.unibo.it/mod/forum/discuss.php?d=150252#p212758`
- Laboratorio 11: `https://virtuale.unibo.it/mod/forum/discuss.php?d=151542#p213979`