

Relazione OOP

“Paradox Platformer”

Ji Junkai, Keegan Falcao, Olivieri Michele

23 ottobre 2024

Indice

1	Analisi	3
1.1	Descrizione	3
1.2	Requisiti	4
1.3	Modello del Dominio	5
2	Design	8
2.1	Architettura	8
2.1.1	Controller	8
2.1.2	Model	9
2.1.3	View	9
2.1.4	Comunicazione tra i Componenti	10
2.2	Design Dettagliato	10
2.2.1	Keegan Carlo Falcao	10
2.2.2	Junkai Ji	15
2.2.3	Olivieri Michele	23
2.3	Conclusioni	26
3	Sviluppo	28
3.1	Testing automatizzato	28
3.1.1	Junkai Ji	28
3.1.2	Keegan Carlo Falcao	28
3.1.3	Olivieri Michele	29
3.2	Note di sviluppo	29
3.2.1	Junkai Ji	29
3.2.2	Keegan Carlo Falcao	29
3.2.3	Olivieri Michele	31
4	Commenti finali	32
4.1	Autovalutazione e lavori futuri	32
4.1.1	Junkai Ji	32
4.1.2	Michele Olivieri	32
4.1.3	Keegan Carlo Falcao	33

A Guida utente	34
A.1 Avvio del gioco	34
A.2 Controlli di gioco	34
A.3 Come Vincere i Livelli	35

Capitolo 1

Analisi

Il gioco preso in analisi è un platform game 2D chiamato **Paradox Platformer**. Il giocatore controlla un personaggio attraverso l'uso delle frecce direzionali, con l'obiettivo di raggiungere la fine della mappa, per passare al livello successivo. Tuttavia, il gioco introduce una serie di sfide controintuitive: oggetti che sembrano innocui possono rivelarsi letali, e viceversa. Ad esempio, alcuni oggetti come le monete e i monete letali appaiono identici, ma uno può essere raccolto senza conseguenze mentre l'altro uccide il giocatore. Pertanto la caratteristica distintiva del gioco è che molti degli ostacoli e degli elementi della mappa si modificano in risposta a specifici trigger attivati dal giocatore, i quali possono trasformare anche le condizioni di vittoria e sconfitta. Il gioco sfida le aspettative del giocatore, richiedendo attenzione non solo agli ostacoli evidenti ma anche a quelli che cambiano dinamicamente. Il concetto di paradosso è centrale nell'esperienza di gioco, poiché le meccaniche sfidano le aspettative del giocatore, spingendolo a interpretare in modo creativo le regole per superare i vari livelli.

1.1 Descrizione

Il software mira alla creazione di un platform game: **Paradox Platformer**, in cui il giocatore esplora una serie di mappe superando ostacoli e attivando meccanismi che modificano l'ambiente di gioco. Il giocatore, controllato tramite le frecce direzionali, deve muoversi con attenzione, poiché molti elementi apparentemente innocui possono trasformarsi in pericoli letali. Le monete raccolte possono contribuire al progresso o, in alcuni casi, causare la sconfitta. Gli ostacoli nel gioco possono variare la loro natura e comportamento, creando una sfida costante e spingendo il giocatore a prestare attenzione sia agli oggetti visibili che ai trigger nascosti che potrebbero attivarsi durante il percorso.

1.2 Requisiti

Requisiti funzionali

- Il gioco, **Paradox Platformer**, sarà un platform 2D in cui il giocatore dovrà esplorare una serie di livelli caratterizzati da sfide controintuitive e ostacoli unici. L'obiettivo principale sarà superare ostacoli e raggiungere la porta di uscita di ogni livello per passare al successivo.
- Ogni livello conterrà una combinazione di ostacoli letali o non letali. Gli ostacoli letali causeranno la sconfitta immediata del giocatore, mentre quelli non letali modificheranno temporaneamente le sue abilità o l'ambiente di gioco.
- I trigger saranno associati a determinati ostacoli e si attiveranno quando il giocatore li colpirà o soddisferà certe condizioni. Questi trigger potranno attivare effetti come cambiamenti nell'ambiente di gioco o nella fisica del personaggio.
- Ogni ostacolo presenterà una meccanica che sfida le aspettative del giocatore, creando un'esperienza di gioco complessa e imprevedibile.
- Il gioco sarà strutturato su livelli progressivamente più difficili, con nuove meccaniche introdotte in ogni fase.
- Il sistema di fine gioco varierà a seconda delle azioni del giocatore, come la raccolta di monete per passare al livello successivo. La raccolta di un certo numero di monete potrebbe sbloccare l'accesso ai livelli successivi.

Requisiti non funzionali

- Il gioco dovrà funzionare in modo fluido su diversi dispositivi, garantendo un framerate costante e tempi di risposta rapidi anche con livelli complessi e numerosi effetti attivi.
- Il sistema dovrà essere facilmente estensibile, consentendo l'aggiunta di nuovi livelli e meccaniche senza la necessità di rifattorizzare grandi porzioni di codice esistente.
- Il sistema dovrà essere portatile su diversi sistemi operativi, assicurando che la logica di gioco e le meccaniche funzionino correttamente su diversi piattaforme.

1.3 Modello del Dominio

In "Paradox Platformer", il **giocatore** si muove attraverso una serie di livelli caratterizzati da ostacoli e trigger, i quali influenzano il gameplay in modi spesso controintuitivi. Il protagonista, controllato dal giocatore, deve affrontare diverse sfide interagendo con l'ambiente per avanzare.

Gli **ostacoli** presenti nei livelli possono essere letali o non letali. Gli ostacoli letali, come seghe o talvolta monete, causano la morte immediata del giocatore al minimo contatto. Al contrario, gli ostacoli non letali non provocano la morte del giocatore, ma interferiscono con il suo movimento o alterano temporaneamente le regole del gioco. Ad esempio, un ostacolo potrebbe muoversi inaspettatamente, costringendo il giocatore a rivedere le proprie strategie.

Parallelamente agli ostacoli, vi sono i **trigger**, elementi che, al seguito del loro contatto, attivano intrinseci effetti sugli ostacoli o sull'ambiente circostante. Il passaggio del giocatore attraverso un trigger, ad esempio, potrebbe far comparire o scomparire un ostacolo, oppure modificare il comportamento di oggetti presenti nel livello.

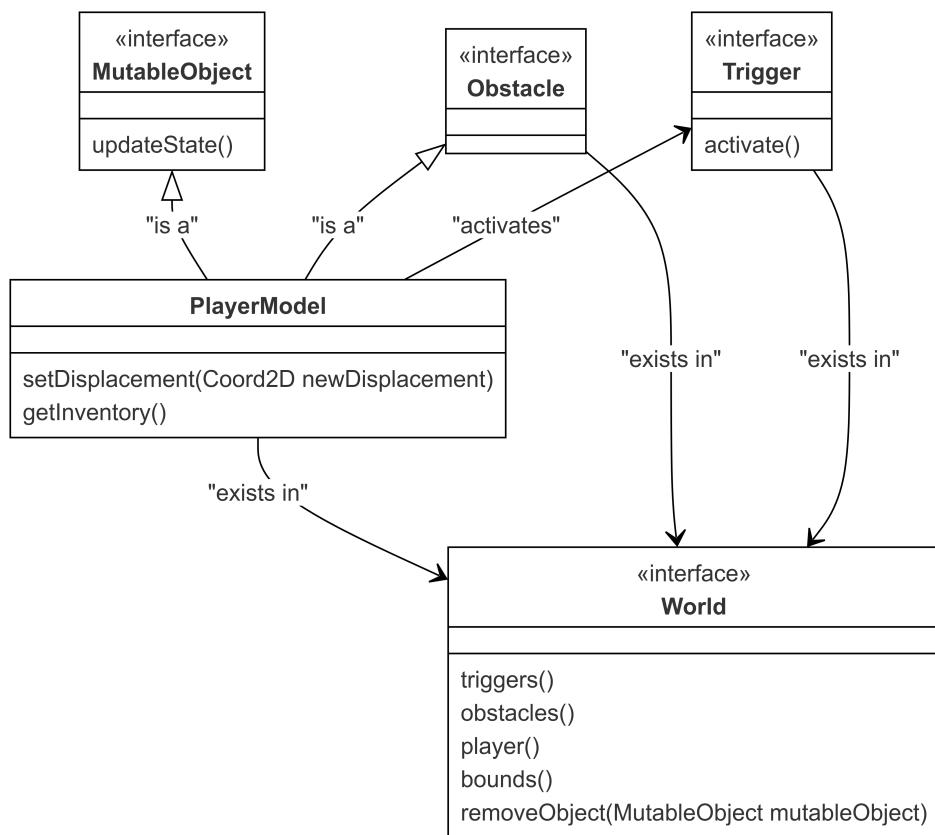


Figura 1.1: Modello del dominio

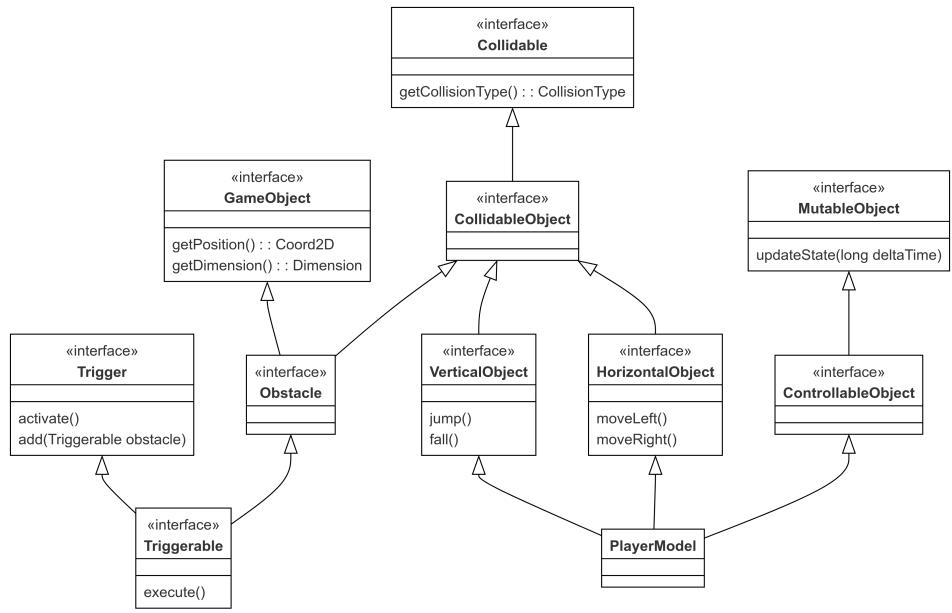


Figura 1.2: Gerarchia delle entità di gioco in uso

Capitolo 2

Design

In questo capitolo vengono illustrate le strategie adottate per soddisfare i requisiti identificati durante la progettazione di Paradox Platformer. Si inizia con una visione architettonale ad alto livello per chiarire come le componenti principali del gioco, come il controller, il modello e la vista, interagiscono e si coordinano tra loro. Successivamente, si approfondiscono i principali aspetti del design, mettendo in luce come le scelte architettoniche e le implementazioni specifiche contribuiscono a garantire un'applicazione modulare e scalabile, adatta a gestire le complessità del gameplay e dell'interazione utente.

2.1 Architettura

L'architettura di *Paradox Platformer* adotta il pattern architettonicale **Model-View-Controller** (MVC) per separare le responsabilità e garantire una struttura modulare e flessibile. Questo approccio facilita la gestione e l'evoluzione del sistema, evitando accoppiamenti stretti tra le componenti principali. Di seguito vengono descritti i ruoli e le interazioni dei principali componenti architettonici del progetto.

2.1.1 Controller

Il **Controller** è il cuore della logica di controllo del gioco e si occupa di orchestrare le interazioni tra il **Model** e la **View**. La separazione delle responsabilità all'interno del controller è fondamentale per mantenere la logica del gioco chiara e scalabile. I principali componenti del controller sono:

- **Controllo del Gioco:** gestisce il ciclo di vita del gioco, la sincronizzazione tra eventi, input dell'utente e visualizzazione. Coordinata anche le verifiche delle condizioni di vittoria e sconfitta tramite l'EndGameManager e le condizioni specifiche di fine gioco.

- **Controllo dell'input:** si occupa della gestione degli input del giocatore, catturando gli eventi di tastiera e aggiornando lo stato del personaggio e del mondo di gioco di conseguenza.
- **Gestione degli Eventi:** gestisce gli eventi asincroni tra le componenti del sistema, assicurando che il GameController, il Model e la View comunichino efficacemente tra loro.
- **Gestione di Fine Gioco:** verifica le condizioni di vittoria e sconfitta. Controlla se il giocatore ha completato il livello o subito una sconfitta e delega la gestione di ciascuno stato alle componenti appropriate.

2.1.2 Model

Il **Model** rappresenta la logica centrale del gioco, comprendente gli elementi che costituiscono il mondo di gioco e le regole che ne determinano il comportamento. I principali componenti del modello sono:

- **Mondo del Gioco:** contiene la rappresentazione dell'intera mappa di gioco, compresi gli oggetti, gli ostacoli e i trigger. Gestisce il posizionamento e lo stato di ogni elemento nel livello attuale.
- **Giocatore:** rappresenta il personaggio controllato dal giocatore, con attributi come posizione, salute e velocità. Il movimento del giocatore viene aggiornato in base all'input ricevuto dall'InputController.
- **Ostacoli:** include gli ostacoli letali e non letali che interagiscono con il giocatore. Gli ostacoli letali come il *DeathObstacle* causano la sconfitta immediata, mentre quelli non letali possono modificare temporaneamente le abilità del giocatore o l'ambiente circostante.
- **Trigger:** attivano specifici effetti sugli ostacoli o sull'ambiente circondante anche in assenza di collisione diretta.

2.1.3 View

La **View** si occupa della rappresentazione grafica del gioco e della visualizzazione dello stato attuale del modello. È progettata per separare la logica di rendering dalle meccaniche di gioco, garantendo un gioco reattivo e fluido. La divisione di responsabilità della view dal modello permette la costruzione di un gioco indipendente dal modello adoperato, diffatti il gioco può funzionare anche se il framework dell'applicazione è differente (e.g Swing, JavaFX o Console). Tale funzionalità è presente in questo progetto, in quanto ogni componente della vista è sviluppata secondo due componenti comuni per tutte le viste (i nodi e gli identificatori dei tasti). I principali componenti della vista sono:

- **Vista del Gioco:** l'area di rendering principale, responsabile del disegno della mappa, del personaggio, degli ostacoli e degli altri elementi di gioco. Viene aggiornata in tempo reale in base alle informazioni fornite dal GameController.
- **Gestione della Vista:** detiene lo stato dell'applicazione, funge da esecutore per qualsiasi azione compiuta dall'utente nei confronti dell'applicazione.
- **Libertà d'uso del Framework:** permette una scelta ampia del framework che si andrà ad adottare, senza andare a compromettere lo scheletro del gioco, ossia la costruzione delle componenti grafiche di esso. In pratica sostituendo il framework la vista funziona ugualmente.

2.1.4 Comunicazione tra i Componenti

La comunicazione tra i vari componenti del sistema è gestita principalmente dall'EventManager. Questo consente al Controller di coordinarsi con il Model e la View senza che queste siano direttamente collegate tra loro. Ad esempio, quando il giocatore raccoglie una moneta o attiva un trigger, l'EventManager notifica il GameController, che aggiorna lo stato del gioco e lo comunica alla View per il rendering.

2.2 Design Dettagliato

In questa sezione, esploreremo in dettaglio gli aspetti chiave del design del nostro gioco *Paradox Platformer*. Cominceremo con una panoramica delle classi `ViewManager` e `GameControllerImpl`, che forniscono la struttura generale del gioco. Successivamente, ci concentreremo sui componenti specifici: `PlayerModel`, `EndGameManager` e `JumpBehavior`, approfondendo le soluzioni adottate, i vantaggi, i possibili svantaggi e i pattern di design utilizzati.

2.2.1 Keegan Carlo Falcao

Ostacoli

Problema: Il problema affrontato dagli Ostacoli è la necessità di standardizzare i comportamenti comuni per diversi tipi di ostacoli in un gioco, consentendo al contempo che ogni ostacolo definisca il proprio comportamento specifico, come la gestione delle collisioni o gli effetti di attivazione. Senza un approccio strutturato, l'implementazione di questi comportamenti variabili tra ostacoli diversi porterebbe a codice ridondante e a una mancanza di coerenza.

Soluzione: La soluzione è fornita dall'uso del *Template Method*. Questo pattern consente alla classe AbstractObstacle di definire uno schema per i comportamenti comuni (come l'aggiornamento dello stato, la gestione del movimento e delle dimensioni), delegando invece gli aspetti specifici, come la gestione delle collisioni, alle sue sottoclassi tramite metodi astratti come `getCollisionType()`. Questo approccio permette di riutilizzare il codice e garantire una gestione coerente dei comportamenti tra gli ostacoli, mantenendo allo stesso tempo la flessibilità per implementare logiche specifiche nelle singole sottoclassi.

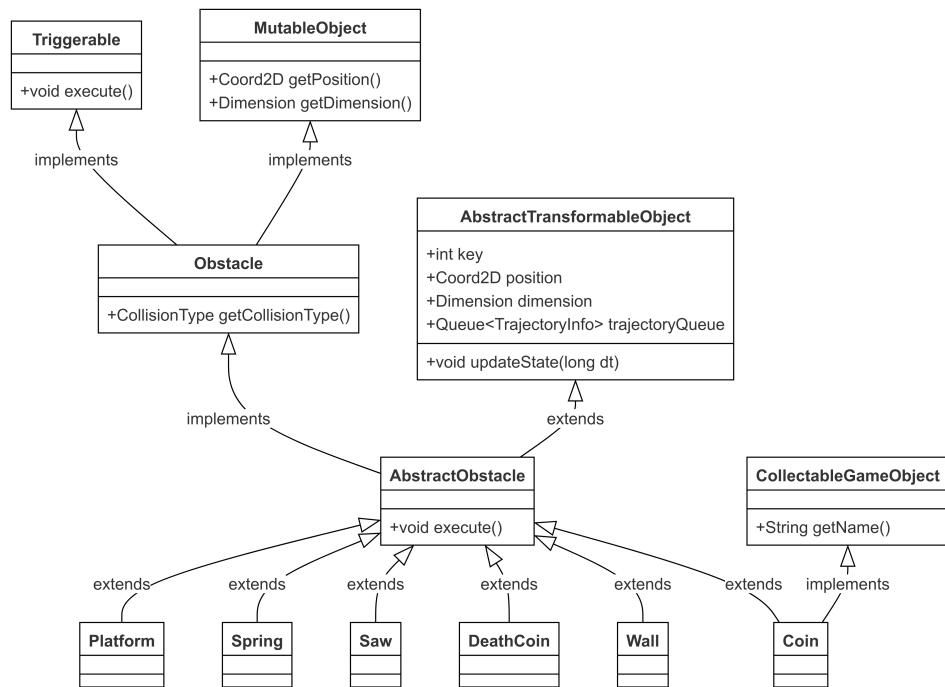


Figura 2.1: Schema UML di Obstacles

Componente Grafica

Problema: La maggior parte di noi avrebbe usato solo i controlli di un framework specifico e l'intera vista sarebbe ruotata attorno ad essa, rendendola prestabilita. Dunque si presentava il problema di utilizzare un componente grafico comune che adottasse diverse specifiche richieste dai diversi framework.

Soluzione: A fronte del problema si è scelto di usare il pattern Adapter che funge da intermediario tra il componente grafico comune e i diversi framework. L'Adapter traduce le richieste specifiche di ciascun framework

(come il settaggio della dimensione e della posizione) in comandi compatibili con il componente grafico comune. Inoltre è stato sfruttato il pattern *Template Method* per creare diversi tipi di controlli avente certe funzionalità in comune.

Vantaggi e Svantaggi: L'uso del pattern Adapter offre vantaggi come il riutilizzo del codice e una maggiore flessibilità, permettendo a un componente grafico comune di adattarsi a diversi framework senza riscrivere il codice. Questo approccio migliora la separazione delle responsabilità e facilita l'estensione dei comportamenti comuni grazie al pattern Template Method. Tuttavia, può aumentare la complessità del progetto e rendere il codice più difficile da mantenere. Inoltre, l'Adapter può introdurre un overhead che influisce sulle performance e richiede una buona conoscenza dei framework, complicando la gestione a lungo termine.

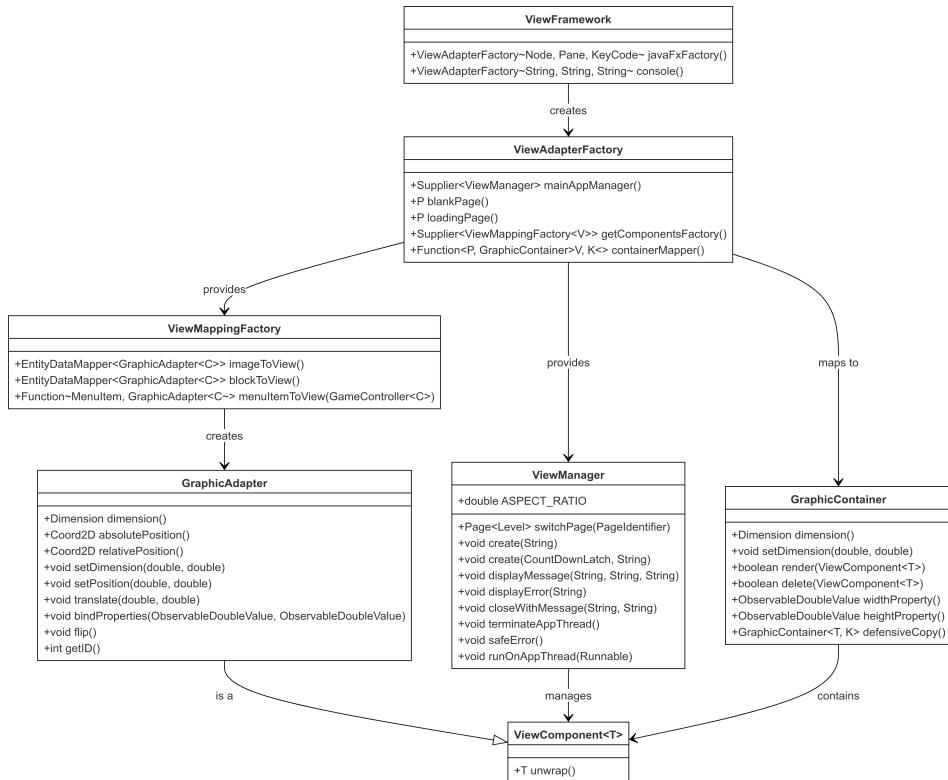


Figura 2.2: Schema UML di Componente Grafica

Modello dei Movimenti

Problema: All'Utente deve essere garantita la possibilità di poter usare differenti configurazioni di comandi per i movimenti del giocatore. Inol-

tre, certe configurazioni richiedono una combinazione di quelli basilari quali 'arrow keys' o 'wasd'.

Soluzione: *Factory* risulta il pattern di composizione più ragionevole nel creare diverse configuzioni ciascuna avente diversi input per gestire il movimento del giocatore, mentre *Decorator* è utile per creare una combinazione delle due: si crea un configurazione complessa mediante l'aggiunta degli input del decoratore specializzato 'arrow keys' con quello da decorare.

Vantaggi e Svantaggi: Il pattern Factory facilita la creazione di diverse configurazioni di comandi, migliorando la modularità e la scalabilità del codice. Tuttavia, può aumentare la complessità del sistema con molte classi e richiede una progettazione attenta. Mentre il pattern Decorator consente di estendere dinamicamente le configurazioni di comandi aggiungendo input specializzati, offrendo flessibilità e personalizzazione. Tuttavia, può rendere il codice più complesso e difficile da gestire a causa della proliferazione di decoratori.

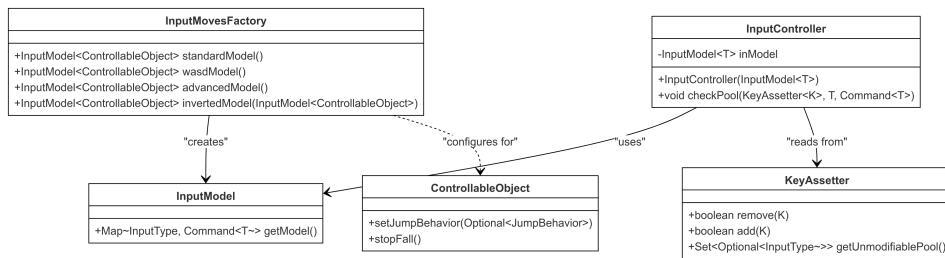


Figura 2.3: Schema UML di Modello dei Movimenti

Costruzione di World

Problema: Per poter implementare correttamente un contenitore delle entità di gioco è necessario costruirlo in modo sicuro e privo di comportamenti indesiderati come l'aggiunta di un'entità in posti non idonei (specialmente dopo aver cominciato il game loop).

Soluzione: Il pattern *Builder* soddisfa a pieno tale richieste in quanto permette la costruzione del mondo delle entità una sola volta, in fase di inizializzazione per esempio, inoltre, oltre a rendere il codice maggiormente leggibile, facilita la possibilità di modellare un **World** secondo una precisa configurazione di oggetti.

Vantaggi e Svantaggi: Il pattern *Builder* offre il vantaggio di garantire una costruzione sicura e coerente del contenitore delle entità di gioco, evitando l'aggiunta non autorizzata di entità e migliorando la leggibilità del codice.

Tuttavia, può introdurre complessità aggiuntiva nel processo di costruzione e potrebbe risultare eccessivo per configurazioni semplici, comportando un aumento del tempo di sviluppo e delle risorse necessarie.

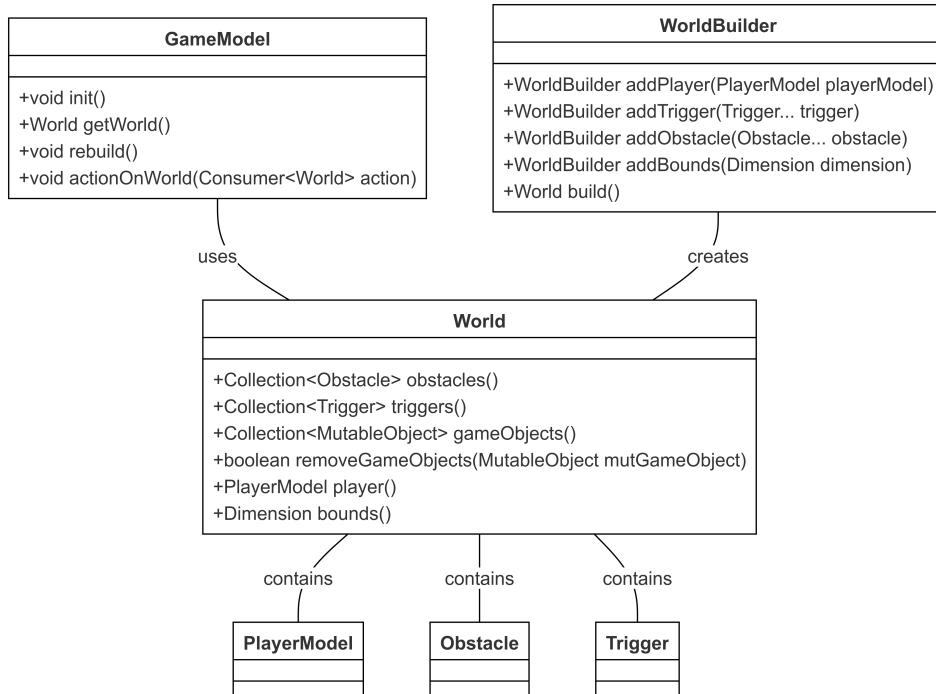


Figura 2.4: Schema UML di World

Coordinamento della View

Problema: Non è stata semplice la gestione della view, in quanto molte classi utilizzate dipendono dal framework utilizzato e non è possibile gestirle tutte in maniera dislocata, specie per il cambio della vista, la quale richiede l'accesso a diverse istanze legate all'applicazione.

Soluzione: Una soluzione ad occhio e croce sarebbe quella di implementare la gestione della view in un unico metodo, o attverso la costruzione in un' elaborata gerarchia di classi in modo da avere a disposizione tutte le istanze legate all'applicazione pronte per essere accedute. Tale soluzione venne subito scartata in quanto, oltre ad essere troppo corposa, molti metodi differenti dovevano richiedere l'accesso alla vista (magari per cambiarla) e non potevano raggiungerla a causa di una mancata centralizzazione di essa. Il pattern *Singleton* risponde esattamente ai mancati adempimenti della precedente soluzione, in quanto ogni classe ha l'accesso alla classe centralizzata ViewManager e tali operazioni restano sicure in quanto essa viene inizializzata in modo unico.

zata durante la creazione dell'applicazione e quindi prima di ogni eventuale modifica del gioco.

Vantaggi e Svantaggi: Il pattern *Singleton* offre una gestione centralizzata e sicura della vista, permettendo a tutte le classi di accedere a una singola istanza di ViewManager. Questo approccio garantisce coerenza e semplifica la modifica della vista, evitando problemi di sincronizzazione e accesso non coordinato. Tuttavia, può introdurre un accoppiamento rigido tra le classi e la vista, riducendo la flessibilità del sistema e rendendolo più difficile da testare. Inoltre, se non gestito correttamente, il Singleton può diventare un punto di congestione nel codice, con tutte le classi che dipendono dalla stessa istanza.

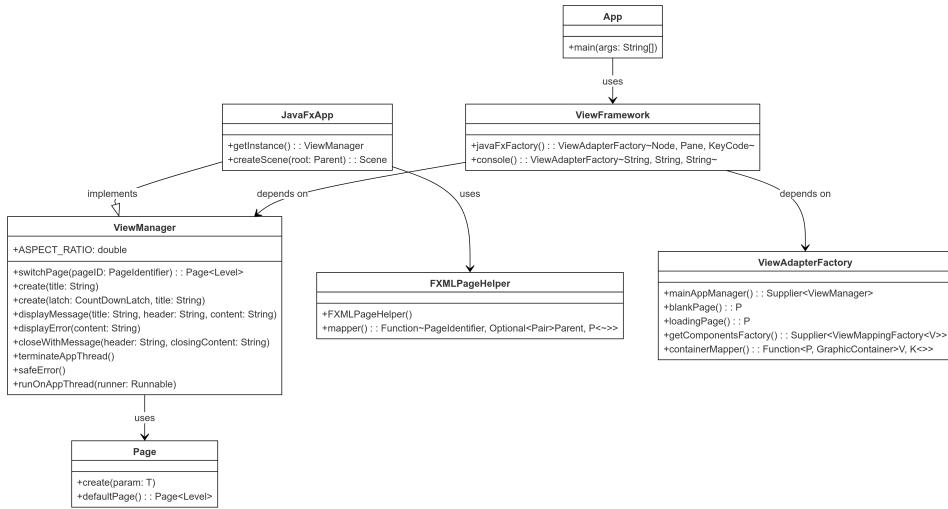


Figura 2.5: Schema UML di View

2.2.2 Junkai Ji

Effect

Problema: Con una struttura molto frammentata, comprendere come e a chi applicare gli effetti può diventare complesso. Ogni effetto deve sapere sia chi è il destinatario dell'effetto (target) che chi lo applica (self). Questa complessità aumenta quando gli effetti possono essere applicati non solo tra il giocatore e un oggetto, ma tra qualsiasi coppia di oggetti di gioco. La gestione di tali effetti richiede una chiara comprensione delle relazioni e delle priorità per garantire che gli effetti siano applicati e rimossi correttamente.

Soluzione Proposta: L'interfaccia Effect è progettata per gestire esplicitamente i dettagli dell'applicazione degli effetti. Il metodo apply consente

di specificare sia l'oggetto destinatario dell'effetto (target) che l'oggetto che applica l'effetto (self), permettendo una gestione chiara e flessibile delle interazioni. Questo approccio è sufficientemente generico da supportare effetti tra qualsiasi coppia di oggetti di gioco, non limitandosi solo al giocatore e agli oggetti con cui interagisce.

Vantaggi e Svantaggi: Il design proposto consente una gestione chiara e versatile degli effetti, facilitando l'applicazione di effetti tra qualsiasi coppia di oggetti di gioco. La struttura generica e la capacità di gestire effetti asincroni tramite CompletableFuture migliorano l'efficienza e la flessibilità del sistema. Tuttavia, la generica natura degli effetti richiede una gestione accurata per evitare confusione e garantire che gli effetti siano applicati e rimossi correttamente.

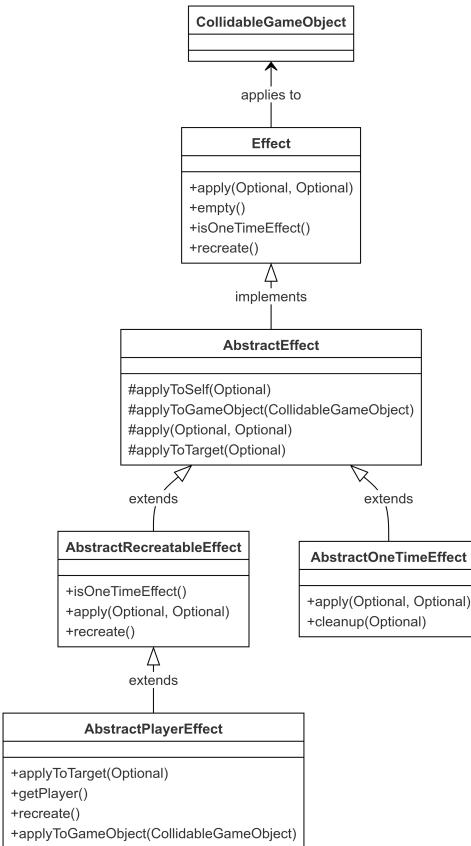


Figura 2.6: Schema UML di Effect

Trigger

Problema: In un gioco, è comune che gli effetti debbano essere attivati non solo in risposta a collisioni tra il giocatore e gli ostacoli, ma anche in base ad altre condizioni o eventi nel gioco. Questo richiede un meccanismo che possa attivare effetti anche in assenza di collisioni dirette, modificando dinamicamente la mappa del gioco.

Soluzione Proposta: L'uso di una interfaccia Trigger introduce metodi specifici per gestire l'attivazione di azioni o eventi associati a una collezione di ostacoli, consentendo di attivare effetti senza dipendere esclusivamente dalle collisioni.

Vantaggi e Svantaggi: Questo approccio arricchisce il gameplay e la complessità del gioco. Tuttavia, uno svantaggio significativo è che i trigger devono essere configurati e associati correttamente prima del loro utilizzo.

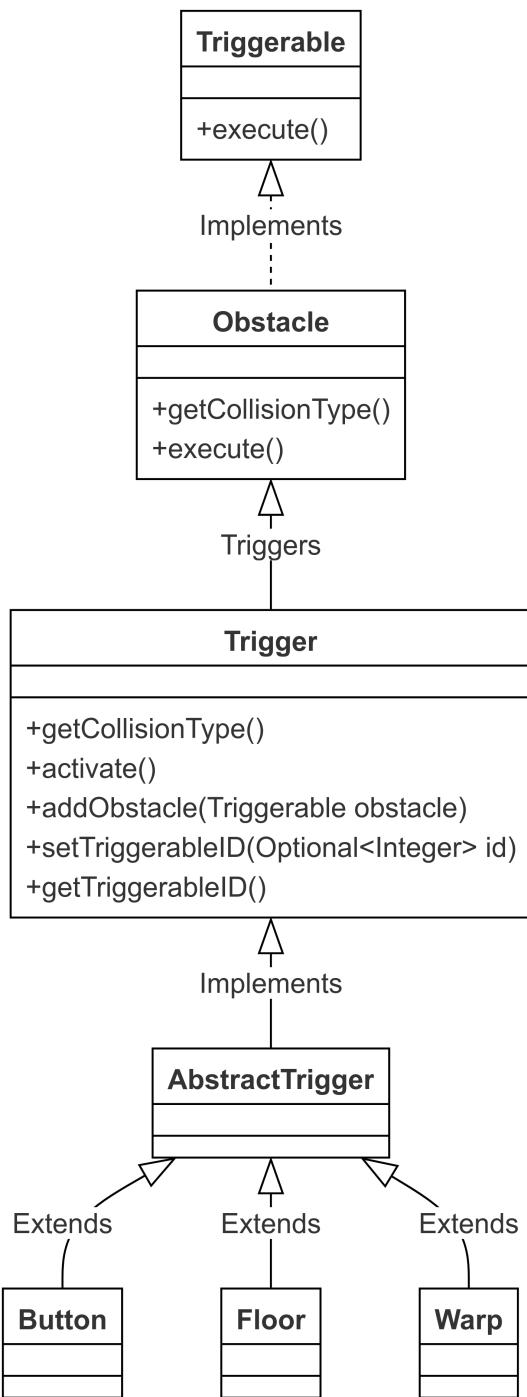


Figura 2.7: Schema UML di Trigger

EffectHandler

Problema: La gestione degli effetti applicati durante le collisioni tra il giocatore e gli oggetti di gioco può diventare complessa, specialmente quando si tratta di avere sequenze di effetti di lunghezza variabile e diversi livelli. È fondamentale garantire che gli effetti vengano applicati correttamente e che le risorse siano gestite in modo efficiente.

Soluzione Proposta: La classe `EffectHandlerImpl` gestisce l'applicazione e la gestione degli effetti nel gioco. Utilizza `TypeEffectsManager` e `ObjectEffectsManager` per separare e gestire gli effetti basati sui tipi di collisione e sugli oggetti specifici. Gli effetti possono essere sia generali (per tipo di collisione) sia specifici (per oggetti), e la classe è in grado di applicare e resettare questi effetti in modo coerente. Inoltre, è capace di ricreare gli effetti se necessario, utilizzando la riflessione per verificare e invocare il metodo `recreate` degli effetti.

Vantaggi e Svantaggi: Il principale vantaggio è che gli effetti sono completamente estratti dagli oggetti di gioco, eliminando la necessità di una relazione diretta tra effetti e oggetti. Questo approccio, centralizzato attraverso l'`EffectHandlerImpl`, migliora la struttura del codice e facilita la gestione degli effetti a livello globale. Tuttavia, l'uso di questa classe può introdurre problemi di performance a causa della ricreazione e eliminazione degli effetti.

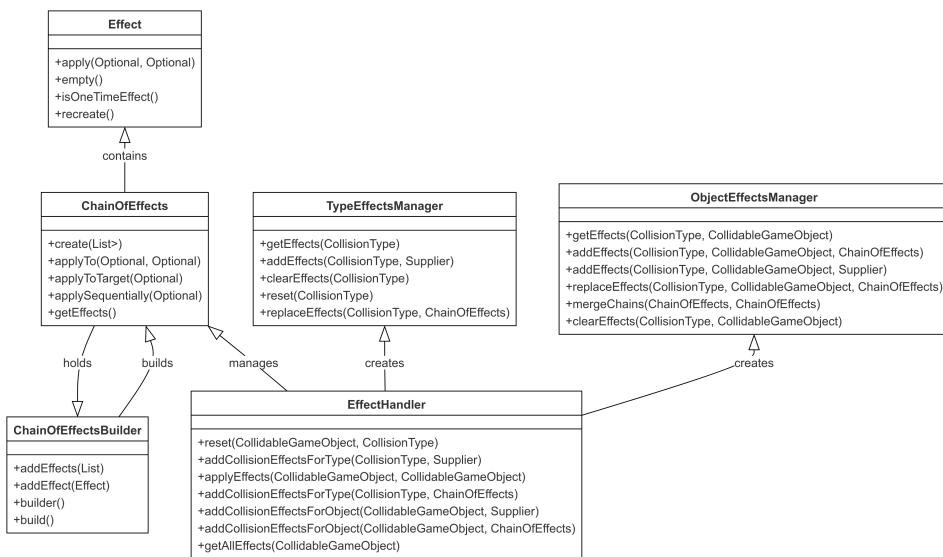


Figura 2.8: Schema UML di EffectHandler

CollisionManager

Problema: La gestione delle collisioni comporta una sequenza di passaggi complessi e ordinati. È necessario gestire non solo l'applicazione degli effetti, ma anche considerare le priorità tra le varie operazioni, come l'attivazione degli eventi di trigger prima dell'applicazione degli effetti e le operazioni di pulizia successivamente.

Soluzione Proposta: CollisionManager e CollisionObserver utilizzano un design modulare per affrontare questa complessità. CollisionManager si occupa della rilevazione delle collisioni, mentre CollisionObserver gestisce l'applicazione e il reset degli effetti. Utilizzando BiConsumerWithAndThen, è possibile creare una lista ordinata di BiConsumer, ciascuno rappresentante un passaggio specifico nel processo di gestione delle collisioni. Questo approccio permette di definire chiaramente le priorità tra le operazioni: prima vengono attivati gli eventi di trigger, poi applicati gli effetti, e infine eseguite le operazioni di pulizia e reset. La possibilità di concatenare questi BiConsumer in modo chiaro e modulare consente un codice pulito e facilmente manutenibile.

Vantaggi e Svantaggi: Il design modulare e l'uso di BiConsumerWithAndThen permettono una gestione ordinata e flessibile delle collisioni e degli effetti, rispettando le priorità tra le operazioni. Questo approccio migliora la modularità e la manutenzione del codice. Tuttavia, l'introduzione di passaggi aggiuntivi e la gestione delle priorità possono aumentare la complessità e richiedere un'attenta pianificazione per evitare conflitti tra le operazioni.

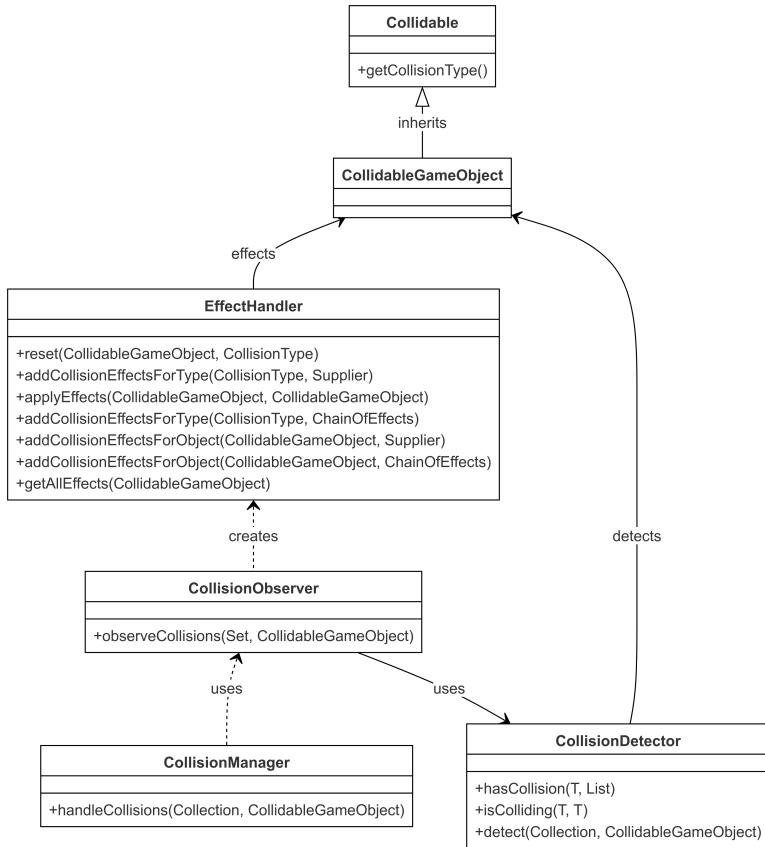


Figura 2.9: Schema UML di Collision Manager

EventManager

Problema: La comunicazione tra classi diverse può diventare complessa, specialmente in un'applicazione con molteplici componenti e moduli. Passare istanze tra le classi può creare un forte accoppiamento e rendere il codice meno manutenibile. È necessario un meccanismo che permetta la comunicazione tra classi senza creare dipendenze dirette tra di esse.

Soluzione Proposta: Il EventManager utilizza un design centralizzato per gestire gli eventi e le azioni correlate. Implementa il pattern Singleton per assicurare che solo una istanza gestisca tutti gli eventi, migliorando la coerenza e facilitando l'accesso globale. Gli eventi e le loro azioni sono gestiti tramite una mappa (eventMap) che associa i tipi di eventi (eventType) a gestori di eventi (BiConsumerWithAndThen). Questo approccio permette di:

- Isolare la Comunicazione: Le classi possono comunicare tra loro attraverso eventi senza riferimenti diretti.

- Flessibilità: È possibile passare qualsiasi tipo di oggetto come parametro agli eventi, rendendo il EventManager altamente generico.
- Gestione degli Eventi: Gli eventi vengono pubblicati e gestiti centralmente, semplificando la gestione e la manutenzione del codice.

Vantaggi e Svantaggi: Il EventManager offre un design flessibile e centralizzato per la comunicazione tra classi, migliorando la manutenibilità e riducendo il forte accoppiamento. Tuttavia, questa flessibilità può essere pericolosa, poiché chi pubblica un evento deve garantire che i sottoscrittori abbiano metodi compatibili con i parametri dell'evento pubblicato. È essenziale che chi utilizza il EventManager sia consapevole delle specifiche degli eventi e dei parametri per evitare errori di runtime e garantire che le azioni siano eseguite correttamente.

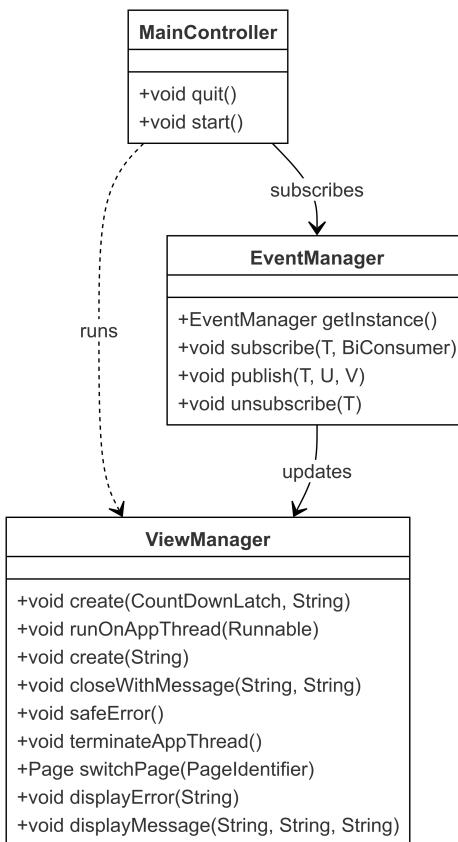


Figura 2.10: Schema UML di Event Manager

2.2.3 Olivieri Michele

PlayerModel

Problema: Il modello del giocatore deve rappresentare e gestire lo stato del personaggio, comprese la posizione, la velocità e il comportamento di salto. Deve inoltre interagire con gli ostacoli e gli oggetti del livello. È fondamentale che il modello del giocatore possa comunicare efficacemente con il resto del sistema di gioco per riflettere i cambiamenti di stato e interagire con altri componenti.

Soluzione Proposta: PlayerModel incapsula lo stato e le azioni del giocatore. Utilizza il pattern *Strategy* per gestire il comportamento di salto, permettendo di cambiare dinamicamente le strategie di salto in base alle esigenze del gioco. Questo approccio consente una gestione flessibile e modulare del comportamento del giocatore, facilitando l'integrazione con il resto del sistema di gioco.

Vantaggi e Svantaggi: Il pattern *Strategy* per il comportamento di salto offre grande flessibilità e facilità di estensione, permettendo di aggiungere nuovi tipi di salto senza modificare il modello del giocatore. Tuttavia, questo può aumentare la complessità del codice, richiedendo una gestione accurata delle strategie e delle loro interazioni. Inoltre, è importante garantire che il PlayerModel comunichi efficacemente con altri componenti come il GameControllerImpl per garantire una sincronizzazione accurata dello stato del gioco.

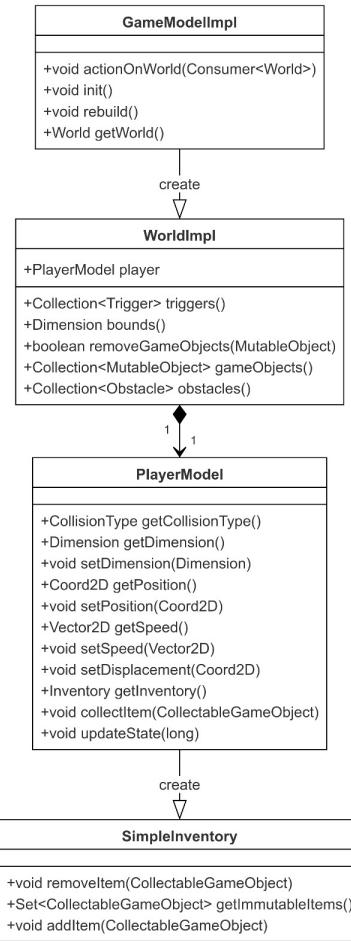


Figura 2.11: Schema UML di Player

EndGameManager

Problema: La gestione della fine del gioco deve determinare se il giocatore ha soddisfatto le condizioni di vittoria o di sconfitta. Questo richiede un sistema modulare che possa gestire diverse condizioni e reagire di conseguenza. Inoltre, è essenziale che il gestore delle vittorie e delle sconfitte possa comunicare efficacemente con il resto del sistema di gioco per garantire una transizione fluida tra i livelli e una corretta gestione degli stati finali.

Soluzione Proposta: Il **EndGameManager** utilizza un design basato sui pattern *Factory* e *Strategy* per gestire le condizioni di vittoria e sconfitta. La factory è responsabile della creazione delle condizioni di vittoria e sconfitta appropriate per il livello corrente, mentre le strategie concrete verificano se le condizioni sono soddisfatte. Questo approccio consente una gestione

modulare e flessibile delle condizioni di fine gioco, facilitando l'aggiunta di nuove condizioni senza modificare il gestore principale.

Vantaggi e Svantaggi: L'uso del pattern *Factory* facilita l'estensione del sistema di condizioni di vittoria e sconfitta, permettendo di aggiungere nuove condizioni senza modificare il `EndGameManager`. Il pattern *Strategy* consente una verifica modulare e flessibile delle condizioni. Tuttavia, la complessità del sistema aumenta con l'aggiunta di nuove condizioni, e la gestione di molteplici strategie può richiedere una maggiore attenzione. Inoltre, è cruciale garantire che il `EndGameManager` interagisca correttamente con il `GameControllerImpl` e altri componenti del sistema per una gestione fluida degli stati finali e delle transizioni tra i livelli.

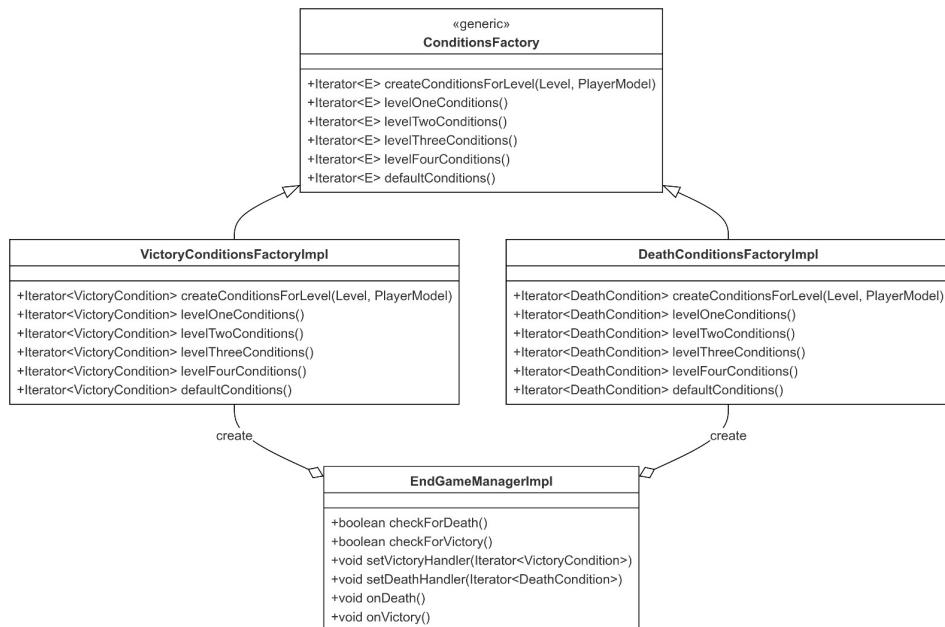


Figura 2.12: Schema UML di End Game Manager

JumpBehavior

Problema: Il comportamento di salto del personaggio deve essere gestito in modo flessibile, supportando diversi stili di salto. La soluzione deve essere estensibile e facilmente modificabile per adattarsi alle esigenze del gioco.

Soluzione Proposta: Il comportamento di salto è implementato utilizzando il pattern *Strategy*. L'interfaccia `JumpBehavior` definisce il contratto per le strategie di salto, mentre le classi concrete come `PlatformJump` e

`FlappyJump` forniscono implementazioni specifiche. Questo approccio consente di cambiare dinamicamente il comportamento di salto senza modificare il codice esistente.

Vantaggi e Svantaggi: Il pattern *Strategy* per il comportamento di salto offre grande flessibilità e facilità di estensione, permettendo di aggiungere nuovi tipi di salto senza modificare il codice esistente. Tuttavia, questo può introdurre una maggiore complessità nella gestione delle diverse strategie e richiedere un'attenta progettazione per evitare la proliferazione di classi.

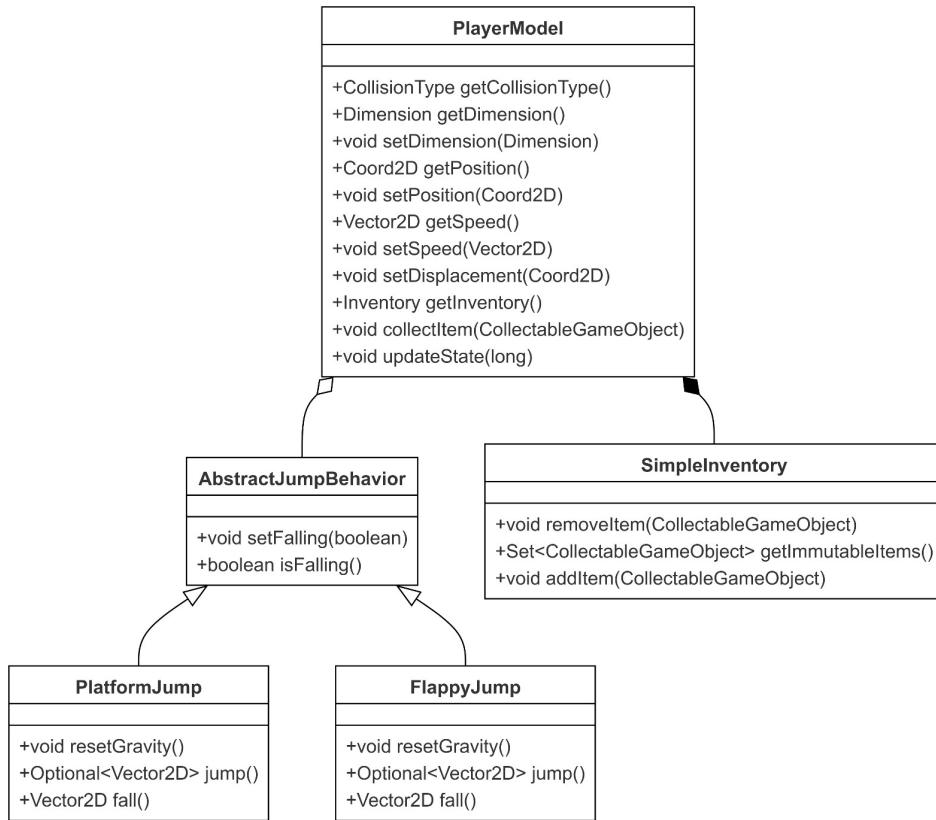


Figura 2.13: Schema UML di `JumpBehaviour`

2.3 Conclusioni

Il design di *Paradox Platformer* è stato progettato per garantire estensibilità, chiarezza e modularità. L'uso dei pattern di design come *Strategy*, *Factory* e *Observer* ha permesso di creare un'architettura modulare e flessibile, facilitando l'aggiunta di nuove funzionalità e la manutenzione del codice. Tuttavia, è importante monitorare la complessità introdotta dai pattern e

bilanciare la flessibilità con la semplicità per evitare problemi di gestione e performance.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Il testing automatico delle classi sviluppate è stato realizzato utilizzando **JUnit**. Ogni membro del gruppo è stato incaricato di creare test per le proprie classi.

3.1.1 Junkai Ji

- **TriggerTest:** verifica che il trigger attivi correttamente gli effetti associati e testa che il trigger possa attivare gli effetti degli ostacoli associati come previsto.
- **EffectTest:** controlla che gli effetti vengano applicati correttamente al giocatore.
- **EffectHandlerTest:** testa la gestione degli effetti da parte dell'EffectHandler, verificando che gli effetti vengano aggiunti, rimossi e applicati correttamente.
- **CollisionTest:** verifica che il CollisionManager rilevi e gestisca correttamente le collisioni tra il giocatore e gli ostacoli.
- **EventManagerTest:** testa che l'EventManager gestisca correttamente l'iscrizione e la notifica degli eventi.

3.1.2 Keegan Carlo Falcao

- **InputControllerTest:** gestione di molteplici input (simulati) e la costruzione di un game loop semplice. Tale test è esente dalla vista.
- **SimpleInventoryTest:** testa la correttezza del contenuto dell'inventario, come ad esempio la raccolta della stessa moneta.

- **GameModelTest**: testa la correttezza di inizializzare il modello di gioco e la eventuale gestione dei casi specifici.
- **WorldBuilderTest**: controllo dell'effettiva costruzione del mondo.
- **WorldTest**: verifica la corretta creazione di un mondo presentando diverse problematiche per casi specifici

3.1.3 Olivieri Michele

CoinCollectionVicotryConditionTest: gestione di una delle condizioni di vittoria del gioco.

3.2 Note di sviluppo

3.2.1 Junkai Ji

Utilizzo di Generic

Utilizzata in vari punti. Un esempio è <https://github.com/OlivieriMichele/OOP23-paradox-platformer/blob/main/src/main/java/com/project/paradoxplatformer/controller/event/EventManager.java>

Utilizzo di Stream e lambda expressions

Utilizzata in vari punti. Un esempio è <https://github.com/OlivieriMichele/OOP23-paradox-platformer/blob/main/src/main/java/com/project/paradoxplatformer/model/effect/managers/TypeEffectsManager.java>

Utilizzo di Reflection

Un esempio è <https://github.com/OlivieriMichele/OOP23-paradox-platformer/blob/main/src/main/java/com/project/paradoxplatformer/model/effect/impl/EffectHandlerImpl.java>

Utilizzo di Custom Biconsumer

Utilizzata in vari punti. Un esempio è <https://github.com/OlivieriMichele/OOP23-paradox-platformer/blob/main/src/main/java/com/project/paradoxplatformer/utils/collision/CollisionObserver.java>

3.2.2 Keegan Carlo Falcao

Utilizzo di Generic

Utilizzata in vari punti. Un esempio è <https://github.com/OlivieriMichele/OOP23-paradox-platformer/blob/aa3a3632b5f52afbd8fd600a9a28f06283fda177/>

```
src/main/java/com/project/paradoxplatformer/controller/input/InputController.java#L21
```

Utilizzo di Stream e lambda expressions e Collectors

Utilizzata in vari punti. Un esempio è <https://github.com/OlivieriMichele/OOP23-paradox-platformer/blob/c38556bb255d05a7f1e0d3def90e5e85b7abedbf/src/main/java/com/project/paradoxplatformer/view/GameViewImpl.java#L85>

Utilizzo di Sets con annesso union()

Utilizzata in vari punti. Un esempio è <https://github.com/OlivieriMichele/OOP23-paradox-platformer/blob/358460ab4b3e26d12c7c43acc5fbaef5306fd84f/src/main/java/com/project/paradoxplatformer/model/world/WorldImpl.java#L121>

Utilizzo di Reflection

Un esempio è <https://github.com/OlivieriMichele/OOP23-paradox-platformer/blob/aa3a3632b5f52afbd8fd600a9a28f06283fda177/src/main/java/com/project/paradoxplatformer/model/mappings/model/ModelMappingFactoryImpl.java#L80>

Utilizzo di Jackson Deserializaton

Un esempio è <https://github.com/OlivieriMichele/OOP23-paradox-platformer/blob/aa3a3632b5f52afbd8fd600a9a28f06283fda177/src/main/java/com/project/paradoxplatformer/controller/deserialization/DeserializerFactoryImpl.java#L24>

Utilizzo di Pair di guava

Un esempio è <https://github.com/OlivieriMichele/OOP23-paradox-platformer/blob/aa3a3632b5f52afbd8fd600a9a28f06283fda177/src/main/java/com/project/paradoxplatformer/utils/geometries/physic/api/Physics.java#L44>

Utilizzo di Function

Un esempio è <https://github.com/OlivieriMichele/OOP23-paradox-platformer/blob/aa3a3632b5f52afbd8fd600a9a28f06283fda177/src/main/java/com/project/paradoxplatformer/view/legacy/ViewAdapterFactory.java#L82>

Utilizzo di Optional

Un esempio è <https://github.com/OlivieriMichele/OOP23-paradox-platformer/blob/aa3a3632b5f52afbd8fd600a9a28f06283fda177/src/main/java/com/project/paradoxplatformer/controller/input/api/InputType.java#L95>

Utilizzo di @FunctionalInterface

Un esempio è <https://github.com/OlivieriMichele/OOP23-paradox-platformer/blob/aa3a3632b5f52afbd8fd600a9a28f06283fda177/src/main/java/com/project/paradoxplatformer/utils/geometries/interpolations/Interpolator.java#L12>

3.2.3 Olivieri Michele

Utilizzo di Generic

Un esempio è: <https://github.com/OlivieriMichele/OOP23-paradox-platformer/blob/4613fdbd0bc4ba4762f649eafdf9f0d2d7c30df8/src/main/java/com/project/paradoxplatformer/model/endgame/ConditionsFactory.java#L1>

Utilizzo di Stream e lambda expressions

Un esempio è: <https://github.com/OlivieriMichele/OOP23-paradox-platformer/blob/4613fdbd0bc4ba4762f649eafdf9f0d2d7c30df8/src/main/java/com/project/paradoxplatformer/model/player/SimpleInventory.java#L1>

Utilizzo di Optional

Un esempio è: <https://github.com/OlivieriMichele/OOP23-paradox-platformer/blob/4613fdbd0bc4ba4762f649eafdf9f0d2d7c30df8/src/main/java/com/project/paradoxplatformer/model/entity/dynamics/behavior/JumpBehavior.java#L1>

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Junkai Ji

Nel corso di questo progetto, sono abbastanza soddisfatto del lavoro che ho svolto, nonostante le sfide e le limitazioni che abbiamo affrontato. La nostra squadra ha dovuto gestire diversi problemi e vincoli di tempo, il che ha impedito una collaborazione coordinata tra i membri. Tuttavia, sono contento di essere riuscito a portare avanti le nostre parti singolarmente per poi combinare il nostro lavoro per ottenere un applicativo finale funzionante. La gestione del tempo è stata una sfida, e alcuni aspetti del lavoro sono stati affrettati a causa delle scadenze. Questo ha influito sulla qualità di alcune sezioni e ha limitato il tempo dedicato ai test e alla struttura del progetto.

4.1.2 Michele Olivieri

Nel complesso, condivido pienamente le riflessioni dei miei colleghi su questo progetto. Nonostante le difficoltà e le limitazioni incontrate, come la gestione del tempo e la mancanza di una collaborazione più coordinata, sono soddisfatto del risultato finale. Mi dispiace, tuttavia, non essere riuscito a raggiungere alcuni degli obiettivi che mi ero prefissato, specialmente per quanto riguarda la struttura del mio lavoro. Guardando indietro, penso che avrei potuto fare meglio in alcune aree. C'è stata forse una certa mancanza di coesione nella divisione dei compiti e nella definizione di una struttura comune, ma devo riconoscere l'alta competenza dei miei colleghi, che ci ha permesso di trovare una soluzione e combinare con successo i nostri contributi per ottenere un applicativo funzionante.

4.1.3 Keegan Carlo Falcao

Il progetto ha presentato sfide significative su molteplici fronti. Sul piano personale, questa esperienza di lavoro di gruppo mi ha reso profondamente consapevole dell'inestimabile valore della collaborazione e dell'impegno collettivo, elementi essenziali per condurre a buon fine un progetto accuratamente concepito e pianificato. Le difficoltà incontrate, mai affrontate prima nella mia carriera accademica, si sono rivelate tutt'altro che agevoli, ma mi considero fortunato per aver avuto l'opportunità di misurarmi con tali ostacoli e di superarli con successo.

Dal punto di vista progettuale, l'attività è stata portata avanti con dedizione, persino in seguito alla separazione di un membro del team, evento che ha reso la distribuzione dei compiti più complessa e nebulosa.

Ritengo, tuttavia, che avremmo dovuto dedicare maggiore attenzione a un'analisi più profonda della progettazione e a una strategia di comunicazione più efficace, poiché sono emerse evidenti carenze in queste aree. Una cura più rigorosa avrebbe certamente contribuito a un esito progettuale più fluido e armonioso.

Appendice A

Guida utente

Questa guida illustra come utilizzare l'applicazione *Paradox Platformer* e fornisce istruzioni chiare per iniziare a giocare. Anche se il gioco è intuitivo, è utile spiegare alcune funzionalità fondamentali per i nuovi utenti.

- **Morte del giocatore:** Se il giocatore muore durante un livello, verrà riportato al punto di partenza di quel livello.
- **Vittoria:** Completando un livello, il giocatore passerà automaticamente al livello successivo.
- **Completamento del gioco:** Una volta superato l'ultimo livello, il giocatore verrà riportato al primo livello.

A.1 Avvio del gioco

1. Quando si avvia il gioco, verrà visualizzato un menu principale contenente l'elenco dei livelli disponibili.
2. Il giocatore può selezionare uno dei 4 livelli cliccando sul nome del livello desiderato. Il livello selezionato verrà caricato immediatamente, e il giocatore potrà iniziare a giocare.

A.2 Controlli di gioco

Per muovere il personaggio nel mondo di *Paradox Platformer*, il giocatore può utilizzare i seguenti tasti:

- **W o Freccia Su:** Salto
- **A o Freccia Sinistra:** Movimento verso sinistra
- **D o Freccia Destra:** Movimento verso destra

A.3 Come Vincere i Livelli

Qui di seguito sono riportati i suggerimenti e le strategie che dovrai seguire per completare con successo ogni livello di *Paradox Platformer*:

1. Fai Attenzione al Muro:

In questo livello, i muri sono pericolosi! Stai attento a non avvicinarti troppo o potresti perdere all'istante. Fai attenzione ai passaggi stretti e cerca di evitare il contatto non necessario con i muri per raggiungere l'obiettivo in sicurezza.

2. Torna Indietro Invece di Procedere:

Le cose non sono sempre come sembrano! In questo livello, invece di andare avanti, dovrai tornare indietro. A volte, tornare sui propri passi è l'unico modo per sbloccare la prossima parte del livello. Cerca percorsi nascosti o trigger che sono accessibili solo tornando in sezioni precedenti.

3. Premi il Pulsante per Vincere:

Un misterioso pulsante è la chiave per la vittoria in questo livello. Per vincere, trova il pulsante e premilo! Potrebbe essere nascosto o difficile da raggiungere, ma una volta attivato, la strada verso la vittoria si aprirà.

4. Raccogli Tutte le Monete per Passare:

Questo livello mette alla prova le tue abilità di raccolta. Devi raccogliere **tutte le monete normali** sparse per il livello per poter passare. Controlla ogni angolo nascosto, poiché se manchi anche solo una moneta potresti non riuscire a progredire.