“Java Jump”

Sabatino Panella, Jacopo Vasi

13 marzo 2025

Indice

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Il progetto a noi commissionato mira alla realizzazione di un videogioco ispirato a "Doodle Jump". Si tratta di un Platformer 2D a scorrimento verticale in cui il personaggio, controllato dal giocatore, dovrà riuscire a saltare piattaforme più a lungo possibile, senza cadere mai oltre il bordo inferiore dello schermo, che comporterà il Game-Over e il dover ricominciare daccapo. In questo genere di giochi (denominati "Endless Auto-Jumpers"), il protagonista salterà automaticamente al tocco della superficie superiore di ogni piattaforma, facendosi strada in un livello infinito generato proceduralmente, senza interruzioni o caricamenti. Essendo un genere di videogiochi dal taglio arcade, nato inizialmente per partite rapide su smartphone, l'obiettivo del gioco sarà semplicemente provare ad arrivare più lontani della partita precedente, accumulando più punti, servendosi anche della raccolta di apposite *monete*, che incrementeranno il *punteggio* attuale di un piccolo bonus, ma che se ottenute in quantità potranno fare la differenza, permettendo al giocatore di raggiungere punteggi alti in minor tempo.

Requisiti funzionali

- **Generazione del livello giocabile**: il livello verrà generato man mano creando apposite piattaforme e monete;
- **Difficoltà dinamica**: la generazione del livello cambierà man mano che si avanza;
- **Gestione della camera**: l'avanzamento verticale della visuale di gioco;

- **Simulazione fisica del movimento**: calcoli di accelerazioni e decelerazioni con i propri parametri;
- **Gestione di collisioni**: quando il personaggio controllato dal giocatore tocca elementi di gioco;
- **Gestione del punteggio**: per tracciare l'avanzamento del gioco in termini numerici;
- **Stati di gioco**: il gioco potrà principalmente trovarsi in quattro stati:
  - **Menu Principale**: dove si avvierà una nuova partita o si uscirà dall'applicazione;
  - **Gameplay**: dove si giocherà effettivamente una partita;
  - **Pausa**: si potrà temporaneamente mettere in pausa il gioco;
  - **Game-Over**: quando il personaggio cade, il gioco termina;
- **Movimento del personaggio e input**: si potranno premere tasti per muoversi a sinistra o destra durante la partita, integrando dinamiche classiche del genere, come l'effetto Pac-Man (ovvero scomparire da un lato dello schermo, per ricomparire dall'altro).

Requisiti non funzionali

- Java Jump dovrà essere *estremamente ottimizzato* e non calare quasi mai di frame-rate, per ottenere partite quanto più immersive e avvincenti possibile;
- L'esecuzione del gioco dovrebbe avvenire correttamente, senza risultare in crash o bug che possono minare all'esperienza di gioco;
- L'interfaccia di Java Jump dovrebbe essere intuitiva e piacevole, garantendo un'ottima esperienza utente;
- Lo scorrimento del livello infinito dovrebbe essere fluido, possibilmente cambiando dinamicamente lo sfondo man mano che il giocatore prosegue e raggiunge punteggi elevati;
- La generazione del livello deve avvenire senza caricamenti, in real-time, gestendo correttamente gli elementi in gioco;
- La meccanica di salto del personaggio dovrà essere piacevole e modificabile, al fine di poter effettuare le dovute modifiche di Game Design successive;

- Le collisioni con le piattaforme avverranno in *maniera estremamente precisa*;
- Il gioco sarà dotato di un minimo di Sound Design per aumentare la componente immersiva.

1.2 Modello del Dominio

Il gioco si avvierà su un menù iniziale, che permetterà di avviare la partita, o uscire dal gioco. Quando il gioco verrà avviato, la generazione procedurale avrà inizio, creando piattaforme e monete, piazzando dunque il personaggio. Quest'ultimo potrà essere controllato tramite input da tastiera, e collliderà con le entità di gioco, "scalando" progressivamente il livello saltando di piattaforma in piattaforma. Lo scorrimento verticale della camera sarà opportunamente gestito quando il personaggio raggiungerà l'altezza di schermo desiderata. Quando il giocatore oltrepasserà il fondo dello schermo, superando la zona visibile (dato che la camera non scorrerà in basso), si attiverà il *game over* (ovvero avverrà la sconfitta), che mostrerà lo score accumulato confrontato con lo score migliore raggiunto. Si potrà quindi da qui tornare al menù e avviare una nuova partita. Durante la partita si potrà controllare il punteggio ottenuto in un punto dello schermo designato, vedendolo incrementarsi mentre si gioca. Si potrà anche mettere il gioco in pausa, per poi riprendere l'esecuzione quando si desidera. Gli elementi costitutivi il problema sono sintetizzati in Figura 1.1.

Alcuni requisiti non funzionali saranno oggetto di maggiore studio durante l'effettiva lavorazione del progetto (essendo molti non considerati come parti integranti del modello di dominio "puro"), per comprendere meglio come poterli raggiungere al meglio, basandosi sempre sull'infrastruttura di base raggiunta durante la progettazione iniziale svolta in questo capitolo.

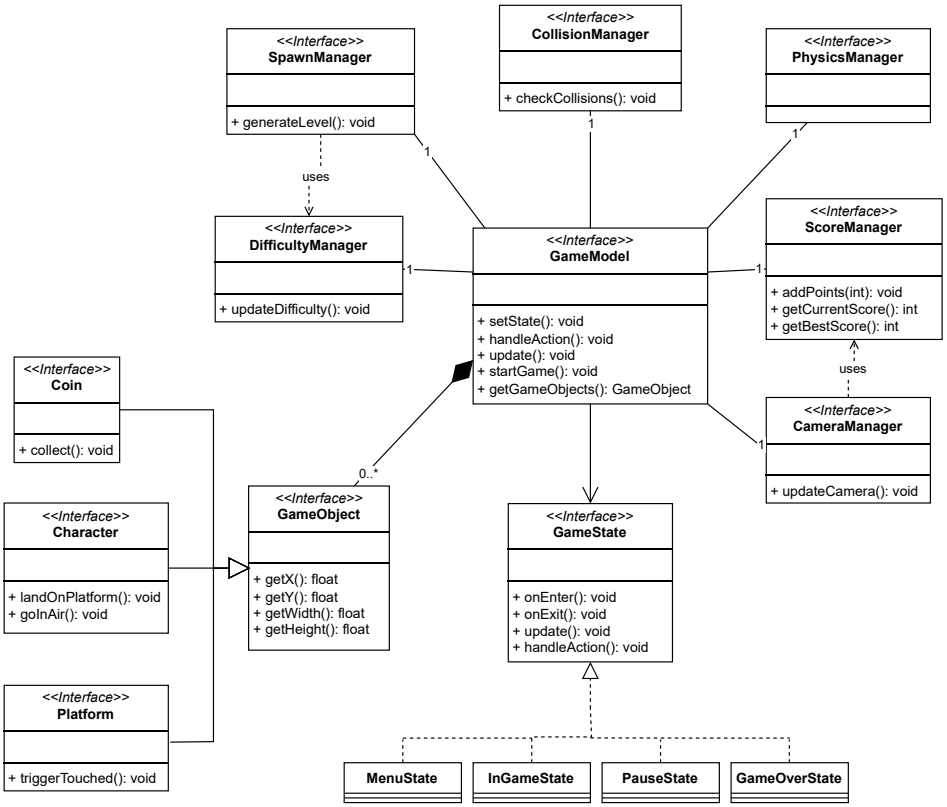

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

L'architettura di Java Jump segue il pattern architetturale MVC. Più nello specifico, si è cercato di mantenere come buon obiettivo di design la separazione tra model, view e controller, ove:

- il **model** descrive il modello di dominio esplicitato precedentemente, con i vari manager che svolgono funzioni specifiche, i quali vengono passati al model per essere utilizzati durante l' esecuzione, cercando di rispettare principi SOLID quanto più possibile;
- il **controller** gestisce il loop di gioco e riceve gli input effettivi da tastiera, che verranno poi passati al model per essere effettivamente processati come azioni di gioco. Il controller dunque svolge la funzione di aggiornare ad ogni frame model e view, durante il game loop;
- la **view** gestisce tutte le componenti visuali e le interfacce di gioco, disegnando quest'ultimo durante i vari game states (menu, in-game, pausa e game-over), e servendosi di un render manager per il rendering delle varie entità di gioco durante il gameplay, soluzione efficace soprattutto per quelle entità dotate di animazioni tramite sprite-sheets precaricate. La view infine si occupa anche della gestione degli aspetti sonori del gioco, quindi musica ed effetti sonori sono anch'essi slegati da logiche e dati del model.

La decisione di tenere model e view separati *non è stata semplice*, dato che nei videogiochi spesso aspetti visuali e dati sono finemente interconnessi. In ogni caso, il problema ci ha portati all'utilizzo dell'Observer Pattern, che nella nostra implementazione funziona seguendo questi passaggi:

1. la view aderisce al pattern observer, implementando dunque il metodo dell'interfaccia "GameModelObserver";
2. il model memorizza tutti gli observer che aderiscono all'interfaccia;
3. i vari game states del model aggiornano gli observer in tempo reale dei propri cambiamenti, notificandoli.

In Figura 2.1 è esemplificato il diagramma UML architetturale che mostra i rapporti tra le componenti principali dell'architettura.

Infine model, view e controller vengono inizializzate da GameInitializer.

In Figura 2.2 è esemplificato il diagramma UML che mostra l'inizializzazione dell'MVC.

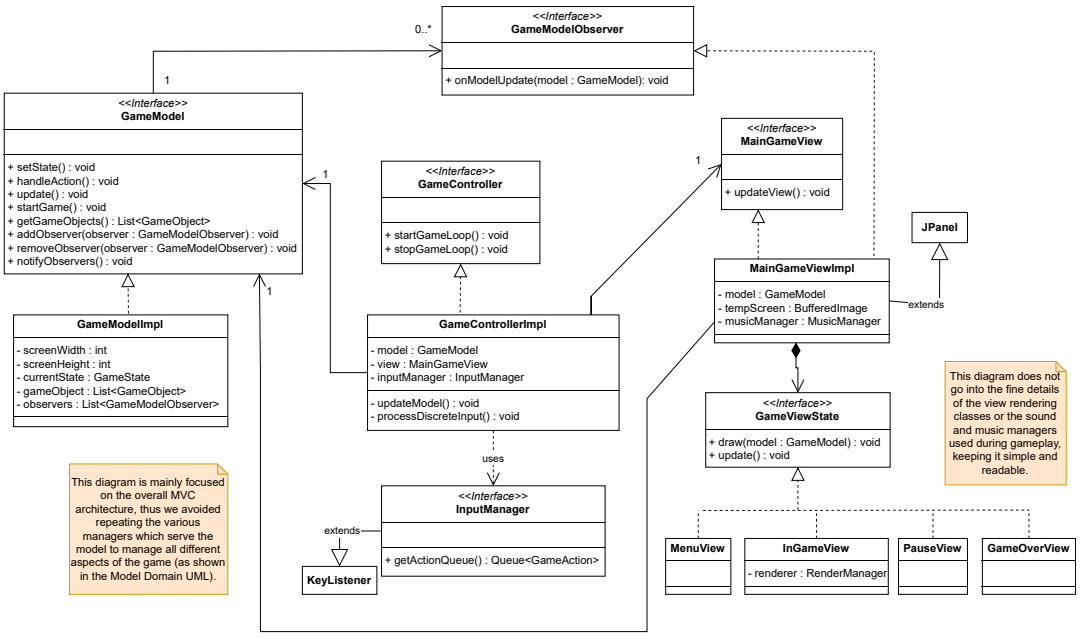

Figura 2.1: Schema UML architetturale di Java Jump.

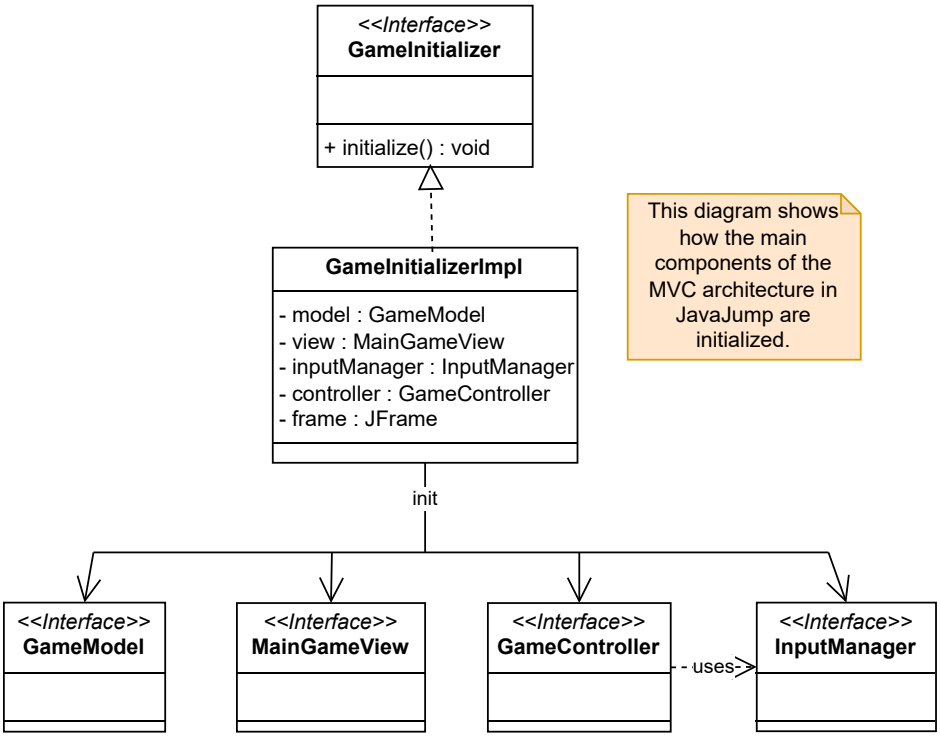

Figura 2.2: Schema UML che esemplifica l'inizializzazione dell'MVC

2.2 Design dettagliato

2.2.1 Sabatino Panella

Definizione delle entità di gioco

- **Problema**: Tutte le entità di gioco hanno in comune diversi aspetti, tra cui (ma non esclusivamente) posizione nel mondo di gioco, dimensioni proprie, metodi di aggiornamento, eventuali metodi da chiamare in caso di collisioni.

- **Soluzione**: Utilizzo di *Template Method* pattern per la creazione di un generico "GameObject", che verrà poi esteso nelle varie entità effettive tramite ereditarietà. GameObject definirà struttura e comportamenti comuni, successivamente tutte le varie entità di gioco (personaggio, piattaforme, monete) estenderanno questa classe implementando o specializzando i metodi necessari. Questa soluzione permette di avere modularità nell'aggiunta di ipotetiche nuove entità di gioco future, ove basterà semplicemente definire una nuova specializzazione di GameObject, senza particolari modifiche al codice già esistente.
  I campi *protected* di GameObjectImpl permetteranno alle sottoclassi implementative delle varie entità (CharacterImpl, PlatformImpl, etc.) di accedere direttamente a queste proprietà senza dover usare metodi getter e setter, garantendo una maggiore flessibilità e facilità d'uso, pur mantenendo un certo grado di incapsulamento.

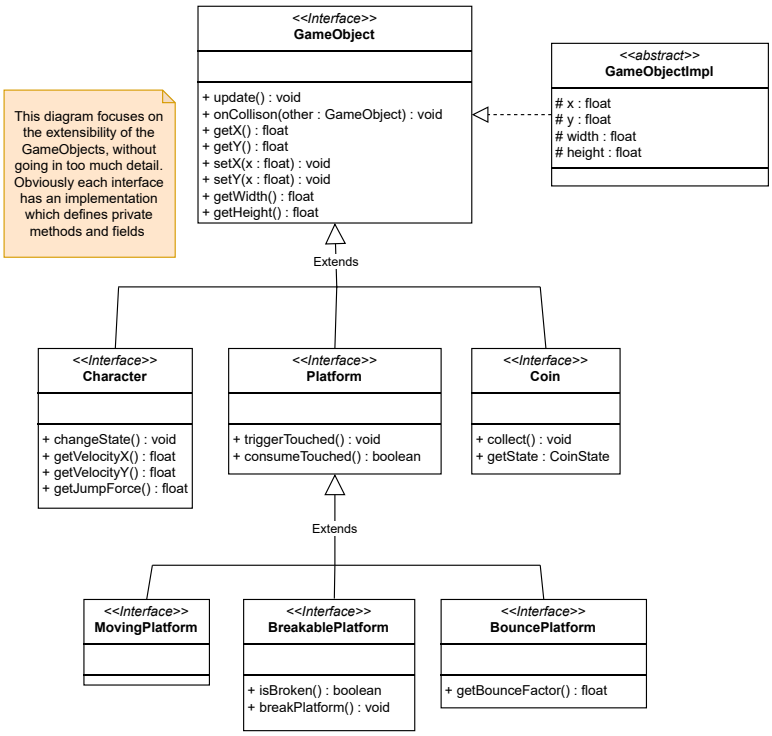

Figura 2.3: Il pattern Template Method è usato per consentire estendibilità in caso di aggiunta di nuove entità di gioco.

Gestione della comparsa del livello di gioco

- **Problema**: La comparsa del livello di gioco (funzionalità chiamata in gergo come "spawn") deve essere flessibile ed aperta a possibili modifiche future, con implementazione di possibili metodologie di spawn diverse e intercambiabili.

- **Soluzione**: Utilizzo di *Strategy* pattern, per associare al manager dello spawn del livello SpawnManager (che non si occuperà direttamente di come vengono creati gli oggetti) una strategia che implementi l'interfaccia SpawnStrategy. In tal modo, se in futuro si volessero adoperare comportamenti diversi di spawn, basterà creare una nuova strategia e farla usare a SpawnManager, senza modificare il suo codice. Questo garantisce estendibilità e una base per dover effettuare meno modifiche possibili al codice su una parte fondamentale del prodotto, ovvero la generazione del livello.

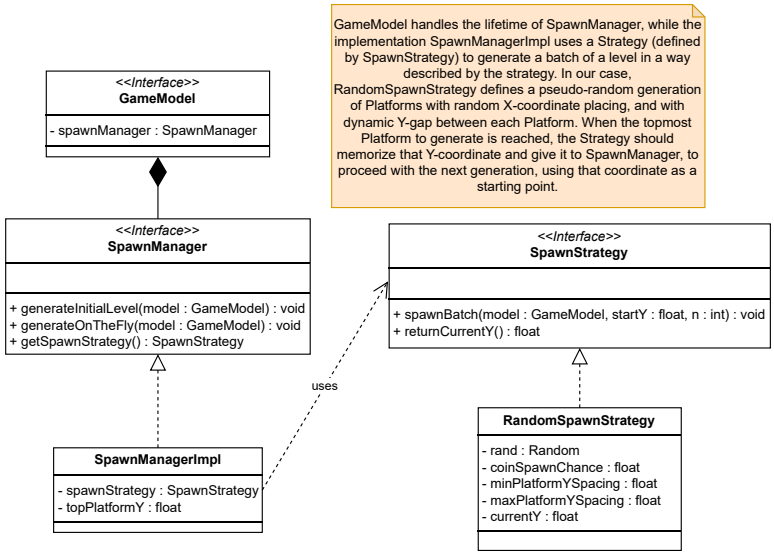

Figura 2.4: Il pattern Strategy è usato per consentire intercambiabilità in caso di aggiunta di nuove logiche di spawn.

Gestione degli stati di gioco

- **Problema**: Il gioco Java Jump (come identificato durante la modellazione del dominio iniziale) si può effettivamente trovare in diversi stati, che andrebbero gestiti ciascuno in maniera simile, ma con implementazione diversa. Fare numerosi operazioni condizionali nel codice sarebbe una soluzione poco estensibile, rendendo l'aggiunta di ulteriori stati di gioco in futuro una richiesta molto gravosa, considerando soprattutto che nei videogiochi è generalmente una delle prime operazioni di aggiornamenti post-lancio (ad esempio: l'ipotetica aggiunta di un minigioco alternativo avviabile dal menù, o di modalità extra di creazione di livelli/personaggi fatti dagli utenti, etc.). L'ideale sarebbe far sì che l'aggiunta di nuovi stati di gioco vada a modificare il meno possibile gli stati già esistenti.

- **Soluzione**: Utilizzo di *State* pattern, uno dei pattern più utilizzati nei videogiochi (e non l'unico esempio di utilizzo in Java Jump, ma sicuramente uno dei più significativi). Il GameModel tiene un riferimento a un'interfaccia "GameStateHandler" e ogni stato identificato (nel nostro caso attualmente Menu, InGame, GameOver e Pause) implementa questo handler e quindi i metodi da esso definiti, delineando il comportamento specifico di quel determinato stato. Così facendo, il model

delega a questo oggetto lo svolgimento di azioni e aggiornamenti in maniera specifica di stato in stato, consentendo una gestione flessibile e modulare degli stati, ove l'ipotetica aggiunta futura potrà essere svolta senza stravolgimenti di codice e funzionalità già presenti.

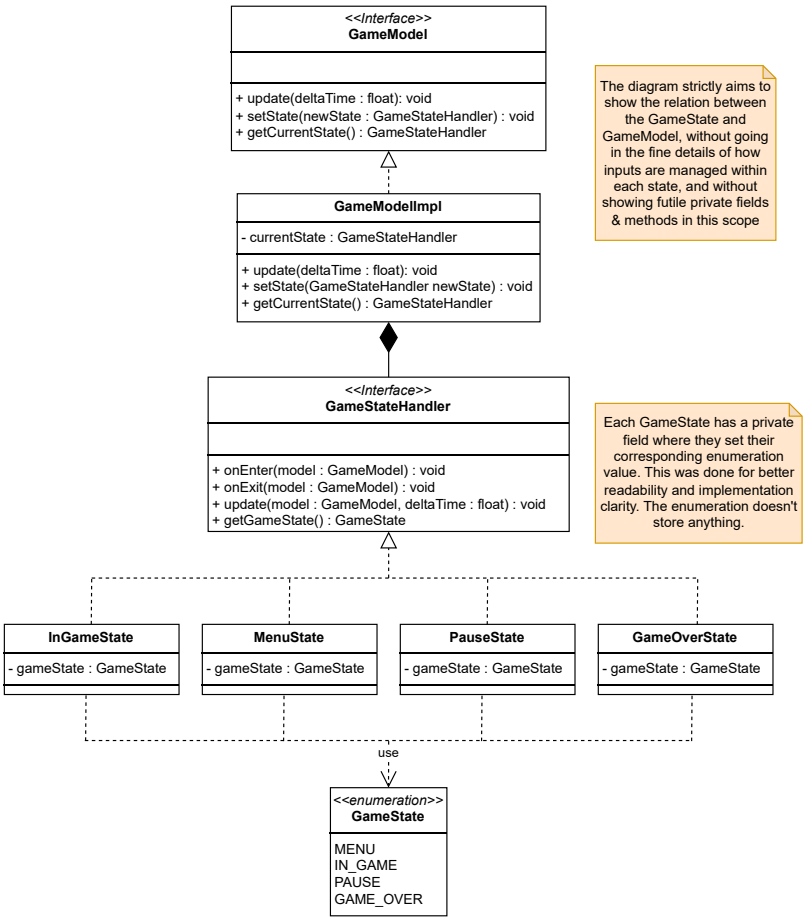

Figura 2.5: Il pattern State è usato per consentire flessibilità in caso di aggiunta di nuove parti di gioco.

2.2.2 Jacopo Vasi

Gestione degli stati della view

- **Problema**: La View del gioco Java Jump deve essere in grado di rappresentare visivamente i diversi stati di gioco (Menu, InGame, GameOver, Pause, ecc.), ognuno con un'interfaccia e un comportamento grafico specifico. Una gestione basata su controlli condizionali risulterebbe poco scalabile e difficile da mantenere, specialmente nel caso di future espansioni (ad esempio, l'introduzione di nuove modalità di gioco o schermate extra). Inoltre, separare la logica della View in più entità specializzate garantisce una maggiore modularità e riusabilità del codice.

- **Soluzione**: Per risolvere questa problematica, anche la View adotta lo State pattern, seguendo la stessa logica del Model. La classe principale della View, chiamata MainGameView, funge da gestore degli stati visivi (GameViewState), mantenendo un riferimento all'oggetto corrente che rappresenta lo stato attuale della View. Ogni stato visivo (MenuView, InGameView, GameOverView, PauseView) implementa l'interfaccia GameViewState, fornendo una rappresentazione grafica specifica in base allo stato attuale.

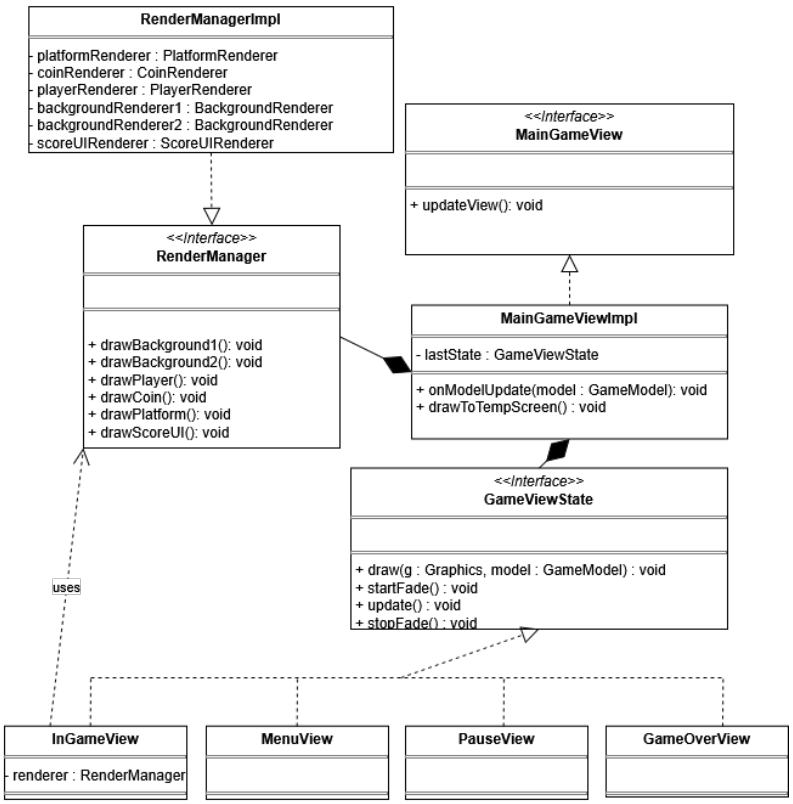

Figura 2.6: Il pattern State è usato per consentire estendibilità in caso di aggiunta di nuovi stati di view.

Gestione input

- **Problema**: La gestione degli input in Java Jump deve essere flessibile e facilmente estendibile, permettendo di mappare i comandi di gioco senza dover modificare direttamente il codice del Controller principale. Un approccio basato su controlli condizionali per ogni input renderebbe la gestione complessa e poco scalabile, specialmente se in futuro venissero introdotte nuove interazioni, come l'utilizzo di item o nuove modalità. Inoltre, è fondamentale separare la logica di gestione degli input dall'effettivo comportamento del gioco per mantenere il codice più modulare e manutenibile.

- **Soluzione**: Per affrontare questa problematica, il gioco adotta il Command pattern all'interno della classe InputManager. Questa classe estende KeyListener e agisce come un gestore centralizzato degli input, traducendo ogni pressione di tasto in un valore di un enum di comandi.

Questi comandi vengono poi passati al Controller del gioco, che si occupa di interpretarli e attivare la logica corrispondente all'interno del Model.

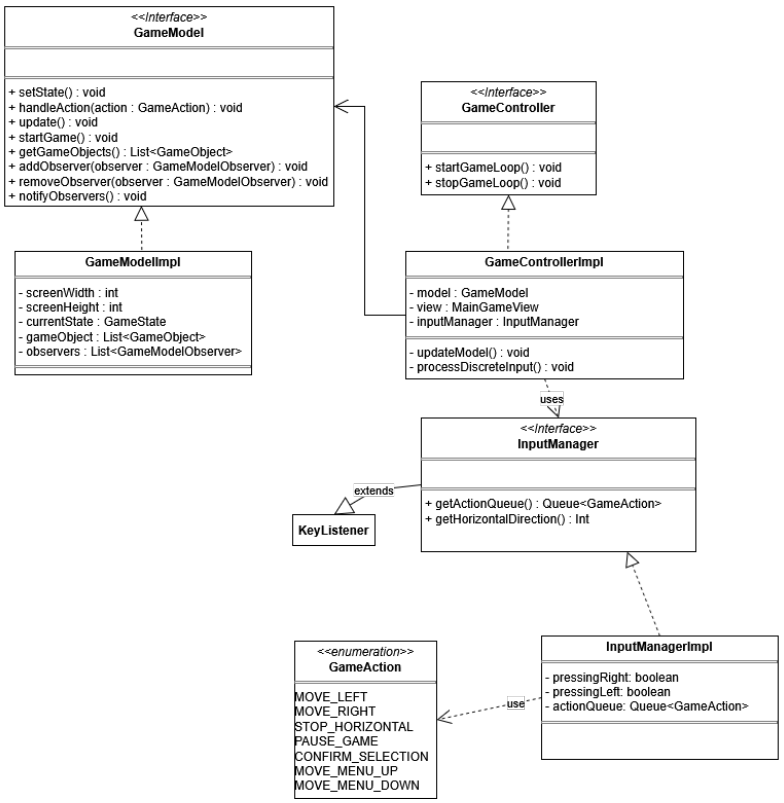

Figura 2.7: Il pattern Command è usato per consentire intercambiabilità in caso di aggiunta di nuovi tipi di azioni corrispondenti a un tasto premuto.

Gestione della creazione degli oggetti di gioco

- **Problema**: Il gioco Java Jump richiede la generazione dinamica di diversi tipi di oggetti di gioco (GameObject), come piattaforme, nemici, power-up e ostacoli. La creazione diretta di questi oggetti all'interno del codice della logica di gioco comporterebbe una forte dipendenza tra le classi e renderebbe difficile l'aggiunta di nuovi elementi senza modificare direttamente il codice esistente. Inoltre, avere un meccanismo centralizzato per la creazione degli oggetti permette una gestione più flessibile del loro comportamento e delle loro varianti.
- **Soluzione**: Per risolvere questa problematica, il gioco utilizza il Factory pattern attraverso la classe GameObjectFactory. Questa classe è

responsabile della creazione di tutte le istanze di GameObject, ricevendo come input il tipo di oggetto richiesto e restituendo un'istanza appropriata senza esporre direttamente la logica di creazione.

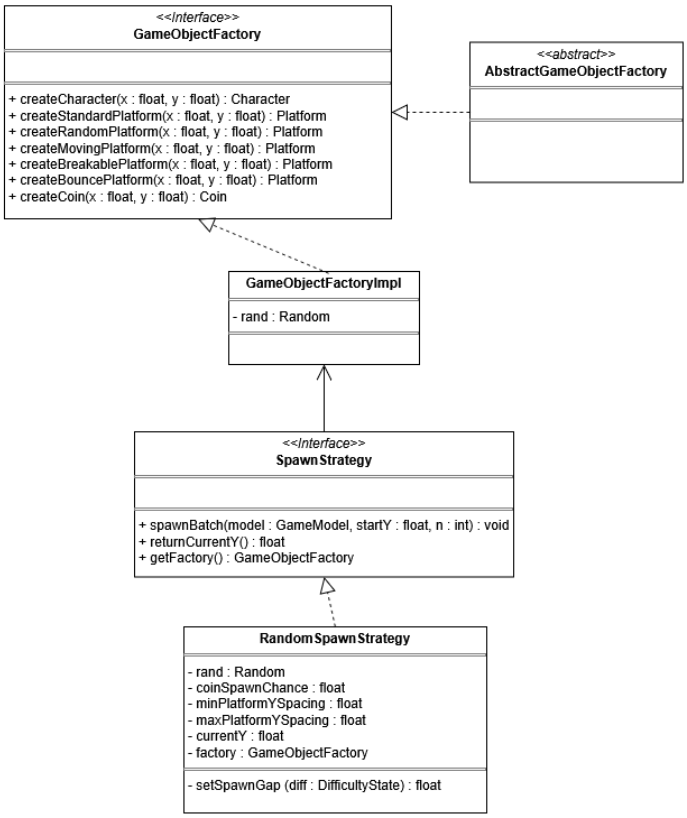

Figura 2.8: Il pattern Factory è usato per suddividere ed isolare la creazione dei vari GameObject in luogo centralizzato e ben leggibile, permettendo buona leggibilità e flessibilità.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Sono stati eseguiti i seguenti test su Java Jump:

- InputTest: controlla il corretto funzionamento e la corretta gestione degli input.
- CameraTest: controlla che la logica di scorrimento della camera venga correttamente applicata e incrementi lo score in caso di scorrimento.
- CollisionTest: controlla che le collisioni tra entities avvengano correttamente.
- GameStateTest: controlla che il corretto funzionamento dei GameState e il cambio corretto al verificarsi delle condizioni richieste.
- ModelTest: controlla che le funzionalità del model vengano eseguite correttamente (Score, HandleAction, ObserverNotification, GameInitializer, SetStateChangesState).
- PlayerTest: controlla che le physics e le funzionalià del player avvengano correttamente.
- DifficutlyTest: controlla il corretto incremento di difficoltà con l'aumento dello score.

3.2 Note di sviluppo

Bounding-Box Check tra entità di gioco (AABB)

BoundingBoxCheck basico trovato e riadattato per applicarsi al nostro caso (GameObject generici).
Permalink: `https://github.com/SabbaPanella/OOP24-JavaJump_Project/blob/725a928a616e405a98194f4c432811fccdb46df4/src/main/java/it/unibo/javajump/model/collision/CollisionManagerImpl.java#L75-L80`

GameLoop Thread management

Gestione del GameLoop effettuata per avviare l'esecuzione del gioco in tempo reale, rielaborato per adattarsi alle nostre esigenze. Risorse visionate:

- Java Design Patterns Website;
- Stack Overflow.

Permalink: `https://github.com/SabbaPanella/OOP24-JavaJump_Project/blob/725a928a616e405a98194f4c432811fccdb46df4/src/main/java/it/unibo/javajump/controller/GameControllerImpl.java#L54`

Dynamic resizable screen

Gestione del ridimensionamento dinamico della finestra di gioco in tempo reale ottenuta combinando le implementazioni trovate dalle seguenti risorse:

- RyiSnow - YouTube;
- M3832 - YouTube.

Permalink: `https://github.com/SabbaPanella/OOP24-JavaJump_Project/blob/725a928a616e405a98194f4c432811fccdb46df4/src/main/java/it/unibo/javajump/view/GameFrameImpl.java#L38-L41`

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Sabatino Panella

Parallelamente alla mia carriera universitaria, mi sono cimentato nello sviluppo di videogiochi a livello lavorativo. Per questo motivo, considero questo esame il connubio perfetto tra ciò che amo fare e l'opportunità di apprendere nuove metodologie, soprattutto a scopo didattico. Devo ammettere che, inizialmente, non è stato semplice porsi come obiettivo lo sviluppo di un gioco: so bene, ormai, che ci sono sempre complessità nascoste e fattori inaspettati (oltre alla naturale aspirazione di volerlo vedere realizzato *esattamente* come lo si era immaginato). Questi elementi possono causare numerosi problemi, specialmente in un progetto universitario, dove è richiesto non solo il funzionamento del gioco in sé, ma anche un significativo impegno di lavoro.

Ciononostante, ritengo che sia stata un'esperienza estremamente formativa, in cui ho cercato di infondere tutta la passione possibile. Ho provato a strutturare il progetto su una base relativamente solida, con l'intenzione di poterlo sviluppare ulteriormente in futuro, sempre con un'ottica personale e didattica. Sono molto soddisfatto del risultato: certo, ci sarebbero molte parti che si potrebbero migliorare, ma la mia (non troppo lunga) esperienza mi ha insegnato che la perfezione non esiste. Tuttavia, esiste sempre un margine di miglioramento, ed è con questo spirito che ho deciso di guardare al progetto che abbiamo creato.

4.1.2 Jacopo Vasi

Bilanciare questo progetto con la mia vita lavorativa è stata una sfida tanto stimolante quanto complessa. Ogni fase dello sviluppo ha richiesto un'attenta pianificazione per garantire che il lavoro procedesse in modo efficace senza compromettere altri impegni professionali. È stato proprio in questo contesto che ho potuto constatare quanto una buona organizzazione e una visione chiara del futuro siano fondamentali per il successo di un progetto di questa portata.

Durante lo sviluppo, è diventato evidente che una pianificazione strategica non è solo utile, ma essenziale per affrontare gli imprevisti che inevitabilmente si presentano. Ogni ostacolo incontrato ha rappresentato un'opportunità per affinare le mie capacità di problem-solving e per trovare soluzioni ingegnose, spesso frutto di creatività e versatilità. Questo è l'aspetto in cui mi ritrovo maggiormente: la capacità di analizzare una difficoltà da diverse prospettive e trasformarla in un'opportunità di miglioramento, senza lasciarmi scoraggiare dalle difficoltà iniziali.

Ritengo che questa esperienza abbia consolidato ulteriormente il mio approccio allo sviluppo: un equilibrio tra metodo e flessibilità, tra organizzazione e capacità di adattamento. Il progetto, pur essendo già un traguardo significativo, rappresenta per me un punto di partenza verso ulteriori approfondimenti e miglioramenti, spingendomi a continuare a crescere sia a livello tecnico che creativo.

Appendice A

Guida utente

L'obiettivo del gioco è saltare più piattaforme possibili, evitando di cadere. La partita proseguirà fino a quando non si verrà sconfitti.

Main Menu

- **ENTER**: avvia una partita;
- **ESC**: chiudi il gioco.

In Partita

- **FRECCIA DESTRA**: sposta il personaggio a destra;
- **FRECCIA SINISTRA**: sposta il personaggio a sinistra;
- **ESC**: metti il gioco in Pausa;

NB: il personaggio potrà *oltrepassare il bordo dello schermo*, riapparendo nella parte opposta. Essendo una feature di bilanciamento di gioco *piacevole*, è consigliato utilizzarla!

Pausa

- **FRECCIA SU**: sposta la selezione in alto;
- **FRECCIA GIU'**: sposta la selezione in basso;
- **ENTER**: seleziona la voce evidenziata.

Game Over

- **ENTER**: prosegui, tornando al Main Menu.

Bibliografia

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[Nys14] Robert Nystrom. *Game programming patterns*. Genever Benning, 2014.

- Musica di background usata:
  RockMan 6 - Shijou Saidai no Tatakai!! [**Complete Works**], Release Date: **1999** , Track used: PlantMan Stage Theme. Disc Serial Number: **SLPS-02379** , ©CAPCOM.
  Utilizzo non a fini commerciali e senza scopo di lucro;

- Effetti sonori utilizzati:
  Pixabay Royalty Free Sounds.
  Utilizzo non a fini commerciali e senza scopo di lucro;

- Grafiche: custom-made dagli autori del progetto!