Relazione del progetto di “Programmazione ad Oggetti”

Alessandro Valmori, Yukai Zhou,
Gabriel Stira

29/02/2024

Indice

Capitolo 1
Analisi

1.1 Requisiti

Il progetto JJ si prefigge come obiettivo la realizzazione di un clone di JetPack Joyride, un gioco mobile arcade di azione a scorrimento sviluppato dalla Halfbrick Studios.

Per gioco arcade di azione a scorrimento, si intende un gioco che ha come obiettivo quello di schivare ostacoli e rimanere in vita per la più lunga distanza possibile, grazie a svariate dinamiche di movimento che il personaggio può acquisire.

Requisiti funzionali

- Il software dovrà gestire correttamente tutte le entità presenti in gioco e quindi il loro movimento, la loro visualizzazione e le interazioni che possono avvenire quando avvengono determinati eventi come la collisione delle stesse.

- Il software dovrà essere in grado di gestire un numero di entità variabile e quindi adattarsi dinamicamente senza riscontrare problemi.

- Il software dovrà essere in grado di adattarsi anche alla difficoltà crescente mano a mano che il giocatore avanza durante la partita e quindi rendere la sfida più impegnativa in modo incrementale.

Requisiti non funzionali

- Il gioco deve rimanere fluido per tutta la durata del gameplay, anche quando sono presenti molteplici entità e la velocità di scorrimento è elevata.
- Il gioco deve funzionare sui tre principali sistemi operativi Windows, Linux, MacOs.

1.2 Analisi e modello del dominio

Il gioco mette a disposizione del giocatore un menù di gioco che permette di navigare tra le diverse funzionalità disponibili, ossia cominciare una partita, aprire lo shop, o uscire dall'applicazione.
Lo shop permette al giocatore di spendere le monete raccolte per sbloccare potenziamenti.
Cominciata la partita, JJ deve garantire una corretta gestione delle varie entità di gioco presenti sullo schermo, oltre che a una gestione adeguata della fine della partita, con possibilità di rigiocare.

Se si viene colpiti da un ostacolo e non si è dotati di particolari potenziamenti provenienti da power-up o altri raccoglibili, la partita deve terminare e il punteggio raggiunto essere salvato insieme alla quantità di monete raccolte, utilizzabili per sbloccare potenziamenti nello shop.
Gli elementi costitutivi del problema sono sintetizzati in Figura 1.2.1.

Le difficoltà principali sono:

• Rendere la generazione degli ostacoli dinamica; ossia fare in modo che le varie posizioni e modi di muoversi degli ostacoli non siano casuali, ma determinate da specifici pattern per garantire un'esperienza meno "luck-based" e più "skill-based".

• Realizzare un sistema di riconoscimento delle collisioni tra entità il più efficiente possibile, in modo che non si muoia 'senza aver toccato l'ostacolo'.

• Gestire correttamente l'implementazione del movimento e del posizionamento di tutte le entità, rendendole inoltre scalabili in base alle dimensioni dello schermo sia prima che durante la partita.

• Una mappa con scorrimento infinito che presenta una serie di sfondi diversi, in grado di cambiare lo sfondo al momento opportuno, così arricchire la varietà del gioco.

• La creazione di un sistema di menu estendibile , così facilitando l'implementazione dei menu con funzionalità diverse.

• La generazione in modo casuale delle monete di forme diverse senza apparire fuori dal schermo e il numero di volte della moneta che apparirono, devono aumentare man mano con la difficoltà del gioco, in modo che i giocatori ricevono ricompense proporzionate alla difficoltà affrontata.

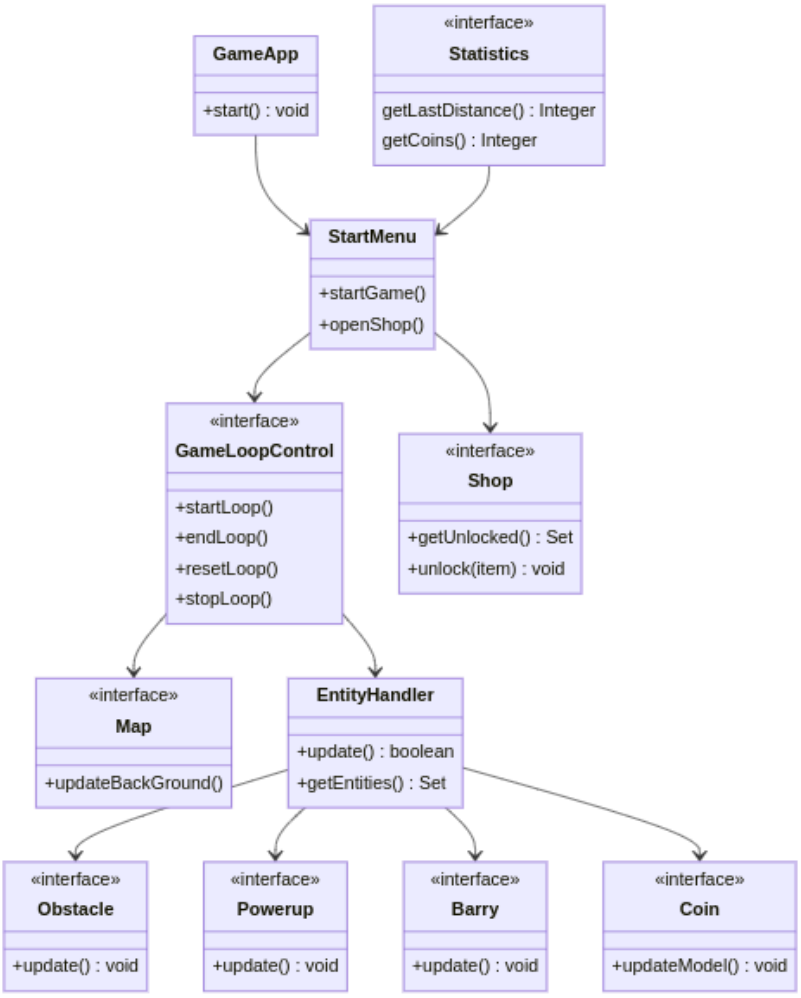

Figura 1.2.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro.

Capitolo 2
Design

2.1 Architettura

JJ non aderisce ad alcun pattern architetturale. La struttura portante (Game Loop) è stata pensata per fare da punto di intersezione tra le varie macro sezioni del software, delegando ad ognuna l'autogestione del rapporto dominio applicativo/view. Le varie entità all'interno del gioco sviluppano quindi una propria, individuale implementazione architetturale basandosi sulle soluzioni più adeguate al loro problema. Nello specifico, viene utilizzato il pattern architetturale MVC per separare il model dalla view di alcune tipologia di entità, mentre per altre, è presente un componente (EntityManager) che aggiorna il loro model in base all'input utente ricevuto dal thread di gioco (GameLoop), istanziando e aggiornando direttamente al suo interno una view che quindi è completamente passiva (in quanto non è essa a ricevere l'input, ma un'altra componente esterna al manager). La mappa, le statistiche e alcuni menù invece non utilizzano l'EntityManager dal momento che vengono gestiti direttamente dal GameLoop.
La gestione diretta della view all'interno del manager da una parte rende semplice la coordinazione con il model e con l'input proveniente dall'esterno, ma dall'altra richiede la modifica dei metodi del manager stesso in caso si decidesse di sostituire in blocco la view.

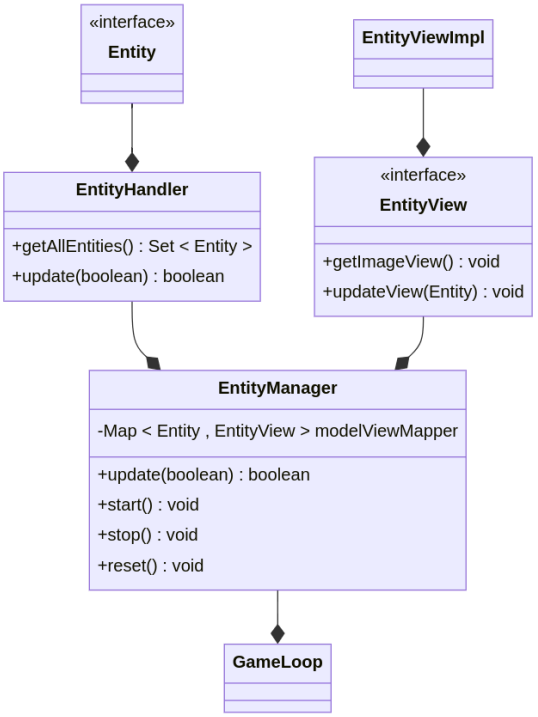

Figura 2.1.1: Schema UML architetturale del sistema di view condiviso dalle entità che estendono Entity.

2.2 Design dettagliato

2.2.1 Alessandro Valmori

Realizzazione del personaggio principale Barry

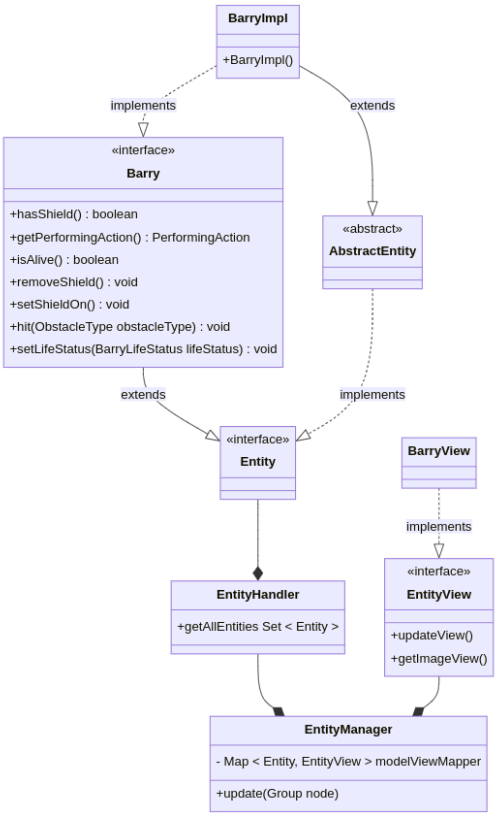

Figura 2.2.1.1: Schema UML che illustra l'allacciamento dell'implementazione di Barry con la struttura generale.

Problema

JetPack Joyride inizia innanzitutto come un gioco in stile "flappy bird", perciò un gioco che consente al personaggio, attraverso l'uso di un singolo tasto, di muoversi verticalmente cercando di evitare gli ostacoli che incorrono verso di esso.

Soluzione

Implementazione di un personaggio principale che riesca quanto meglio possibile ad adattarsi alla struttura generale preimpostata, ovvero cercare di sviluppare un sistema gerarchico organizzato di gestione dei vari tipi di entità.

Come soluzione ho pensato quindi di rendere il personaggio, a cui mi riferirò d'ora in poi come "Barry" per semplicità, un implementazione dell' interfaccia Barry e un'estensione della classe AbstractEntity, le quali rispettivamente implementano e estendono l'interfaccia Entity.

L’interfaccia Barry assume quindi tutti quei comportamenti e proprietà che sono comuni a una qualsiasi entità di gioco che possiede un movimento, una hitbox e uno status.

La componente grafica è completamente separata dal modello, e si relaziona ad esso tramite la classe sopracitata EntityManager. la quale si occupa di reperire, tra le altre entità, le informazioni dall’entità barry, per poi fornirle alla corrispettiva view, in questo caso la classe BarryView, che implementa l’interfaccia EntityView.

Shop

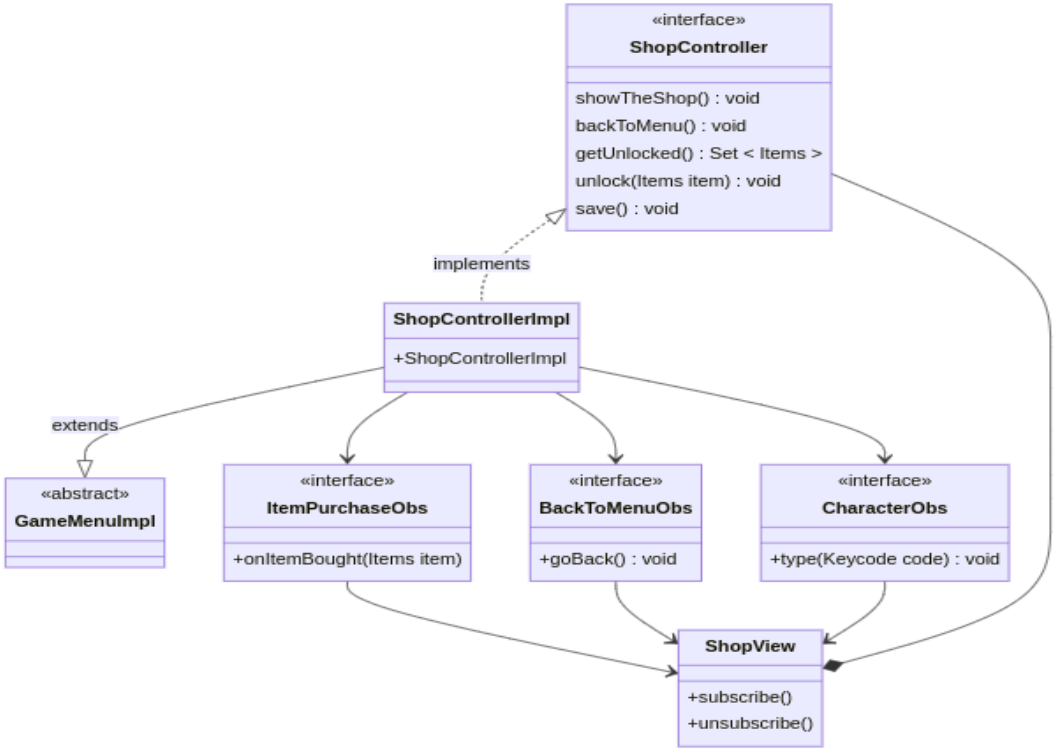

Figura 2.2.1.2: Schema UML che illustra le componenti dello shop, utilizzando l’observer pattern.

Problema

Fornire all’utente un’interfaccia grafica per poter consultare la quantità di monete collezionate e per poter sbloccare alcuni dei powerup e pickup presenti nel gioco, con possibilità di essere in futuro estesa, magari con l’avvento di nuovi powerup o collezionabili. Rendendo quindi necessario lo scambio di alcune informazioni con altre entità di gioco.

Soluzione

Implementazione seguendo l'observer pattern di un negozio a cui è possibile accedervi sia dal main menu, sia dal game over menu.

Il negozio fornisce la possibilità di sbloccare i vari powerup sia acquistandoli con l'uso di monete, sia attraverso l'utilizzo di una password segreta. Il tutto implementato attraverso l'uso di observers che gestiscono gli eventi della ShopView.
L'applicazione del pattern permette la facile estensione di tutto quello che è l'insieme di eventi legati a cambiamenti di stato nella classe observable, in questo caso ShopView, come ad esempio il click su determinati bottoni oppure il pigiare un tasto sulla tastiera. Ciò è consentito grazie alla presenza di metodi subscriber e unsubscriber che permettono agli observer di registrarsi all'observable.

Salvataggio/caricamento delle modifiche

Problema:

Questo sistema prevede un salvataggio e/o un caricamento degli elementi comprati nell'attuale sessione di gioco o in precedenza.

Soluzione (non implementata)

In primo luogo, io e il collega Yukai, avevamo pensato a un metodo di serializzazione comune per salvare su file questi dati, ma ciò ha a lungo andare comportato un inutile aggregazione di dati su una singola classe condivisa, che si appoggiava su file binari, non human readable, che rendeva difficoltoso il processo di debugging e il testing.

Soluzione

E' stato perciò deciso di implementare un sistema di salvataggio/caricamento dati usando file di testo posizionati nella home directory dell'utente, scelta che ha di molto semplificato la struttura di questa parte di progetto.

2.2.2 Yukai Zhou

Menu estensibile

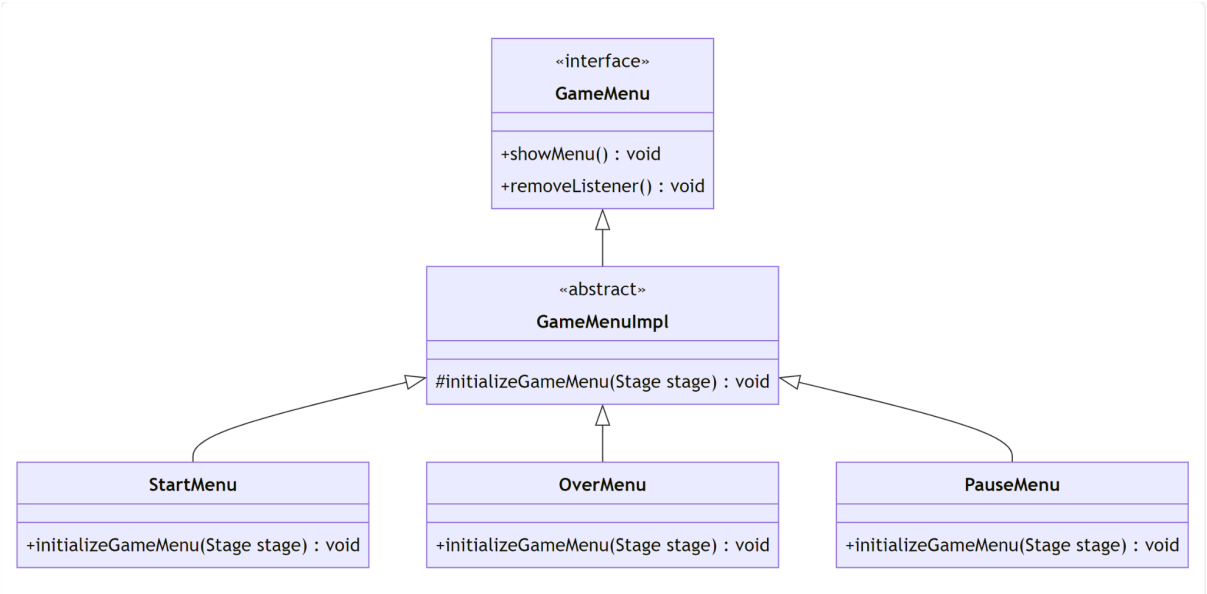

Figura 2.2.2.1: Rappresentazione UML del Template Method Pattern per realizzare un menu estendibile.

Problema

Nelle fasi diverse del gioco ci sono varie situazioni da gestire: l'inizio del gioco, la fine del gioco e la pausa. Quindi per ognuna di queste è necessaria la presenza di un menu corrispondente.

Soluzione

La mia idea è stata quella di creare un menu che può essere facilmente esteso, facilitando la creazione di nuovi menù.

Per realizzare un menu espandibile, la mia soluzione è stata progettare una classe astratta chiamata GameMenuImpl che implementa l'interfaccia GameMenu con il Template Method Pattern. GameMenuImpl fornisce una struttura e dei metodi comuni per tutti gli altri menu, e ha un unico metodo astratto con il nome initializeGameMenu, questo metodo astratto permette agli altri menu di definire il proprio layout, funzionalità uniche e i pulsanti necessari, in questo modo la creazione di un nuovo menu diventa più semplice e viene ridotta la duplicazione del codice.

Riuso del codice nella creazione dei pulsanti

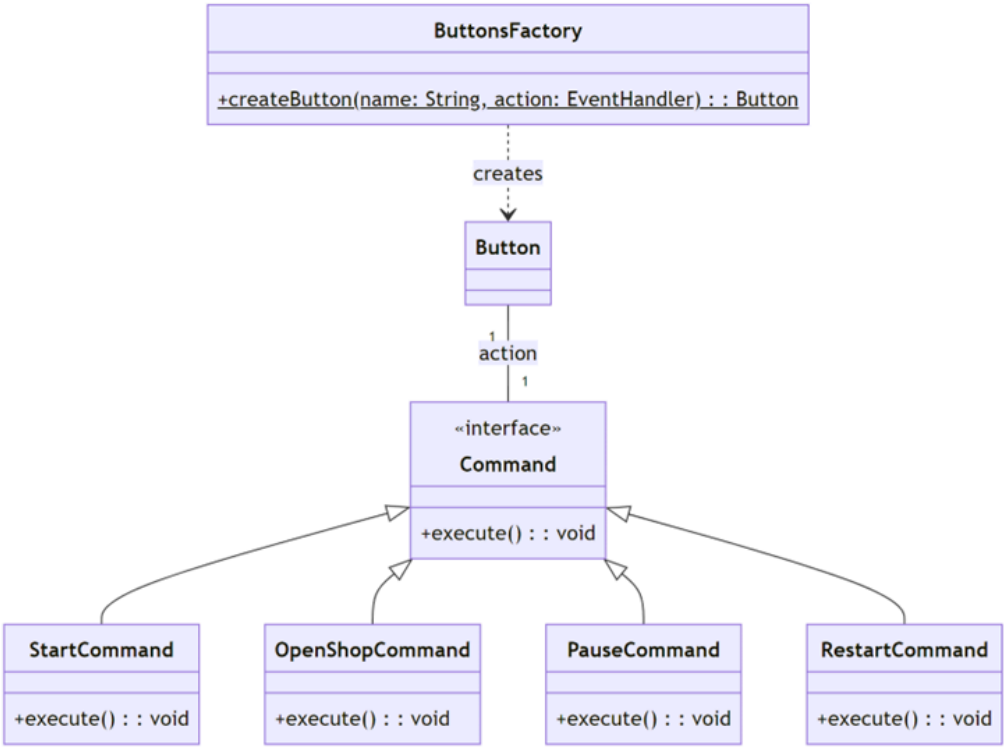

Figura 2.2.2.2: Applicazione del Command Pattern e static Factory Pattern per la creazione dei bottoni nel menu di gioco.

Problema

Durante l'implementazione del menù, mi sono reso conto che la creazione dei pulsanti e anche la loro funzionalità hanno molte somiglianze, causando duplicazione di codice.

Soluzione

Per risolvere questo problema ho progettato una classe denominata ButtonsFactory che utilizza lo Static Factory Pattern. La classe ButtonsFactory ha un solo metodo statico pubblico, cioè createButton(come nella Figura 2.3), che restituisce un pulsante con un'immagine specifica in base ai parametri passati, se il nome dell'immagine passata nel metodo non può essere trovata, viene restituito un pulsante predefinito.

Inoltre, per quanto riguarda le funzioni (o azioni) dei pulsanti, ho utilizzato il Command pattern, incapsulando queste funzioni in oggetti separati. Così, per i pulsanti che hanno la stessa funzione, basta istanziare un oggetto Command specifico, evitando di riscrivere il codice.

Riuso del codice per la creazione delle disposizioni delle monete

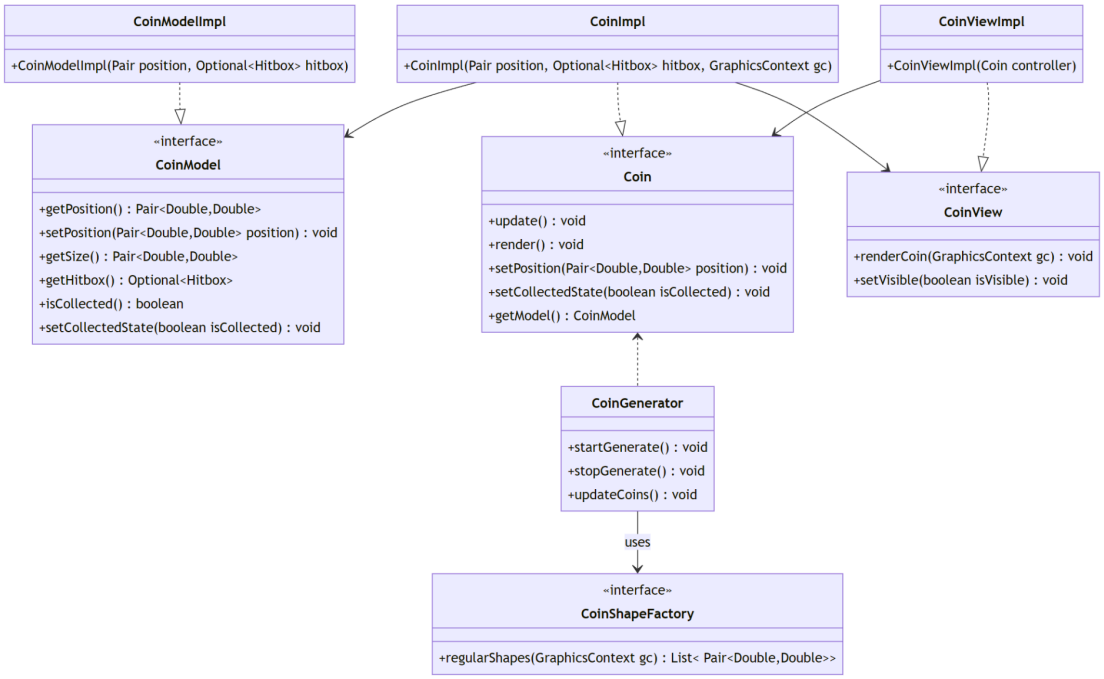

Figura 2.2.2.3: Implementazione della Moneta con MVC pattern e Factory method Pattern per generare le forme diverse della moneta.

Problema

Alcune delle principali difficoltà emerse durante l'analisi del modello riguardanti le monete, è il pattern di generazione delle monete in gruppi di forme diverse.

Soluzione

Per il problema di generare diversi pattern di monete, la mia soluzione è progettare una classe CoinShapesFactory, basata sull'idea del Factory Method Pattern. All'interno della classe, la classe presenta vari factory methods per generare varie forme, come retta, linea multipla, prisma ecc, che vengono poi fornite a Coin Generator, offrendo un certo grado di riusabilità e flessibilità .

Per quanto riguarda il pattern MVC utilizzato, esso segue il seguente schema:
La classe CoinModelImpl gestisce i dati come la posizione , la dimensione e lo stato, e fornisce i dati necessari (incapsulati in un metodo) al CoinImpl (Controller) che a sua volta è in grado di trasmetterli a CoinView.
La classe CoinViewImpl aggiorna la visualizzazione della moneta sulla base dei dati del Model forniti da CoinImpl (controller) .
La classe CoinImpl coordina il aggiornamento della classe CoinModelImpl e CoinViewImpl, e modifica lo stato del modello della moneta in caso di collisione tra moneta e player.

Sfondo a scorrimento infinito

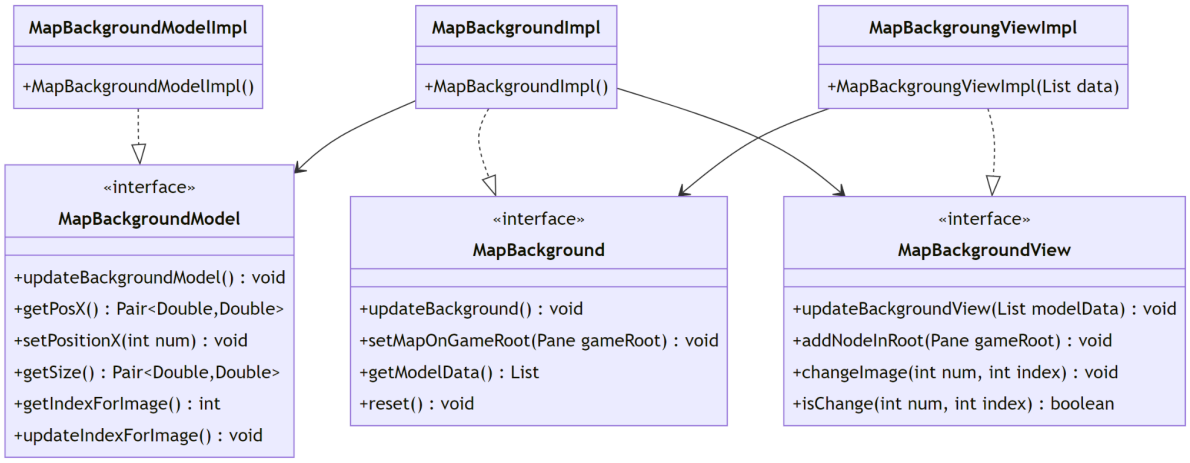

Figura 2.2.2.4: Implementazione dello sfondo a scorrimento con il Pattern MVC.

Problema

Per dare al giocatore un senso di progressione, è necessario creare uno sfondo a scorrimento infinito da destra a sinistra, e in grado di aggiornarsi in certi momenti.

Soluzione

Per quanto riguarda il meccanismo di movimento dello sfondo, ho utilizzato due immagini che si susseguono l’una all’altra costantemente, quando un'immagine esce completamente dall’inquadratura, viene posizionata all’inizio.

Per rendere lo sfondo della mappa modificabile durante il gioco, ho precaricato le immagini necessarie in una lista all'interno del View della mappa. Questa lista contiene sempre un numero pari di immagini. Inizialmente, vengono caricate le prime due immagini della lista su View. Successivamente, in risposta a specifici eventi di gioco, il controller della mappa aggiorna un indice nel modello che indica la coppia di immagini corrente nella lista. Questo indice viene poi passato alla View, che procede caricando la nuova coppia di immagini. Al raggiungimento dell'ultima coppia, il processo riparte dalla prima coppia di immagini, assicurando una transizione fluida e continua.

Salvataggio delle statistiche di gioco.

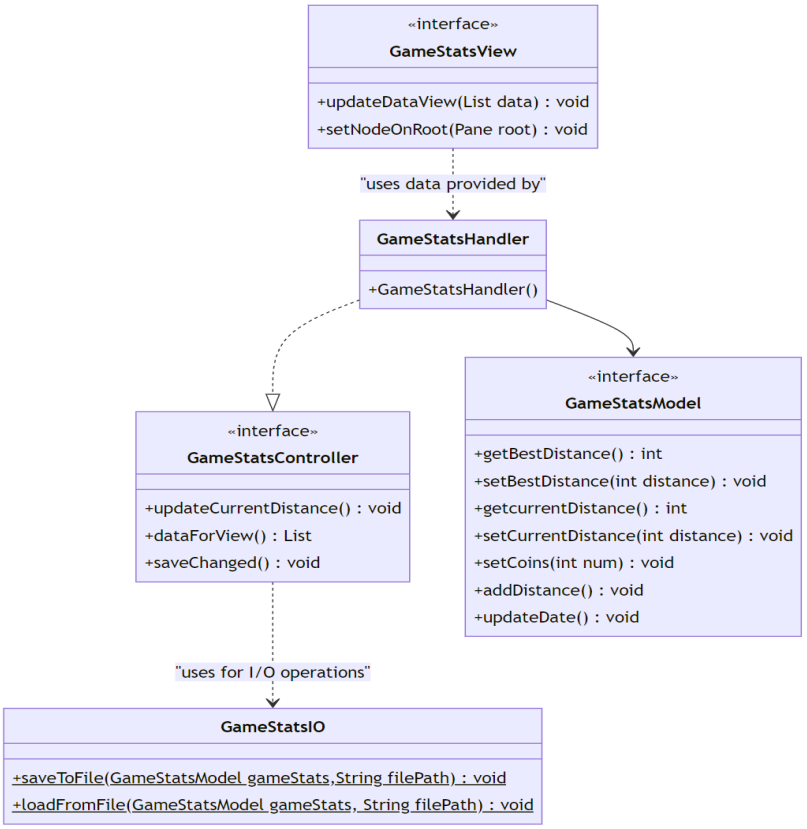

Figura 2.2.2.5: Implementazione delle statistiche di gioco con il pattern MVC, supportata da una classe di utilità per il salvataggio dei dati.

Problema

Nel gioco è importante salvare delle statistiche necessarie per tenere traccia dei progressi nella partita attuale e in quelle precedenti. Quindi questi dati devono essere salvati correttamente dopo la chiusura del gioco.

Soluzione

Soluzione (non implementata)

All'inizio, ho utilizzato il metodo di serializzazione per salvare e caricare i dati, poiché il model di GameStats era una classe serializzabile. Tuttavia, mi sono reso conto che questo approccio rendeva difficile la manutenzione del file, perché i dati venivano salvati in formato binario (non leggibile) e inoltre in caso di danneggiamento del file era impossibile recuperarli.

Soluzione

Per motivi di leggibilità e manutenzione, ho infine deciso di salvare questi dati in un file di testo trasformandoli in formato di testuale. Quindi ho progettato due metodi che sono stati messi in una classe che si chiama GameStatsIO. In questo modo i dati salvati diventano leggibili e modificabili(senza dover modificare il codice).

2.2.3 Gabriel Stira

Entità (Ostacoli, Powerups e Pickups)

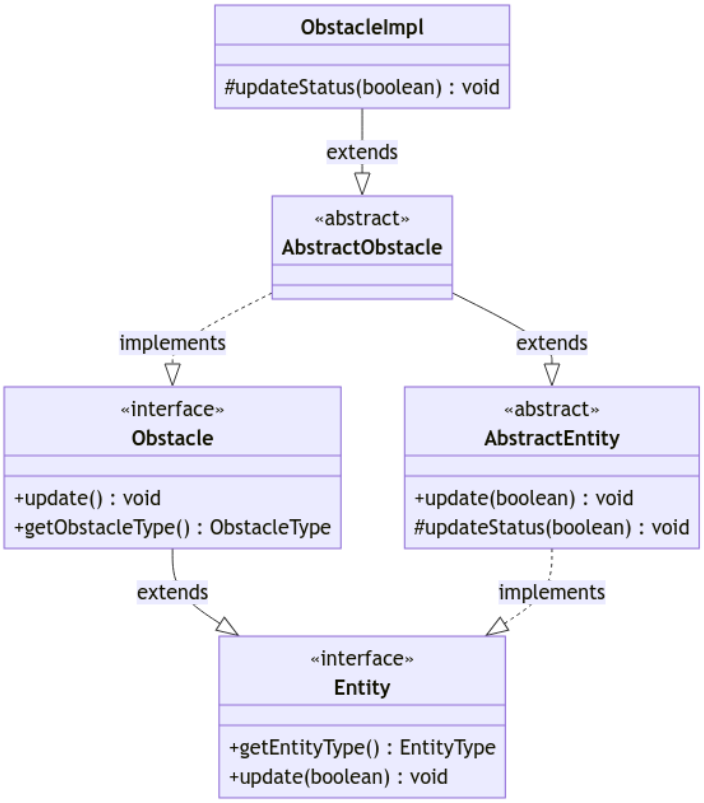

Figura 2.2.3.1: rappresentazione della struttura utilizzata per le entità. Sono state omesse le entità PowerUp e PickUp per semplificare l'UML, ma la loro implementazione è analoga a Obstacle.

Problema
JetPack Joyride deve gestire un numero variabile di entità che possono comportarsi ed interagire in vari modi tra loro.

Soluzione (non implementata)
L'idea è quella di trattare tutte le entità in maniera separata.
Gestire le entità in questo modo comporta limiti meno stringenti e permette maggiore libertà nell'implementazione delle strutture e dei metodi che costituiscono ogni singola entità, ma causa problemi di ripetizioni di codice, scalabilità, manutenibilità del codice…etc nel caso in cui si vogliano aggiungere nuove entità in futuro.

Soluzione
Le principali entità all'interno del gioco hanno tutte in comune alcuni comportamenti e caratteristiche. L'idea è quella di realizzare un'interfaccia comune Entity e una classe astratta AbstractEntity che ne implementa i metodi per ottenere una base di partenza da cui sviluppare poi tutte le entità. Ulteriori interfacce e classi astratte sono poi utilizzate per la definizione di tutte le entità per differenziare i loro comportamenti per tipo.

Comportamento entità

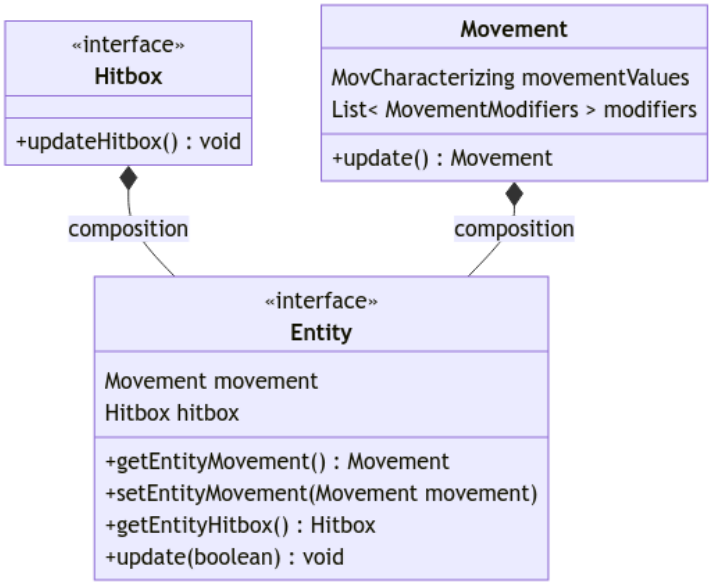

Figura 2.2.3.2: rappresentazione delle caratteristiche principali delle entità. Movimento e Hitbox costituiscono la base su cui tutte le entità si fondano.

Problema

Le entità in JetPack Joyride hanno comportamenti uguali, simili o anche all'occorrenza molto diversi tra loro a seconda del tipo. In particolare, ogni entità deve potersi muovere ed interagire con le altre in caso di collisione.

Soluzione (non implementata)

L'idea è di implementare il comportamento di ogni entità come un insieme di caratteristiche interne dell'entità stessa in maniera indipendente dalle altre entità, ripetendo sempre lo stesso codice per applicare la fisica di movimento, sempre gli stessi modificatori di status e collisione per entità simili, etc…
Poco riuso di codice, processo di generazione di un movimento prolisso, difficile manutenibilità, difficilmente scalabile all'aggiunta di altre entità, etc…

Soluzione

L'idea è quella di riassumere i comportamenti più significativi, movimento e collisione, in due classi Movement e Hitbox che vengono aggiornate ad ogni frame del gioco da un Template Method update(boolean) definito nella classe astratta AbstractEntity. Il metodo updateStatus() delle sottoclassi modifica diverse proprietà dell'entità prima che il controllo torni al metodo update(), che aggiorna successivamente movimento e collisioni in base alle nuove proprietà.
La classe di movimento è immutabile ma è re-istanziabile anche dopo la creazione in maniera semplificata grazie all'utilizzo di un Builder (pattern). La classe effettua automaticamente i calcoli di fisica del movimento e di modificatori personalizzati.
La classe di hitbox è anch'essa immutabile ed effettua automaticamente come per il movimento il calcolo di posizione dei vertici della collisione e il collision detection con le collisioni delle altre entità.

Factory per la generazione delle entità

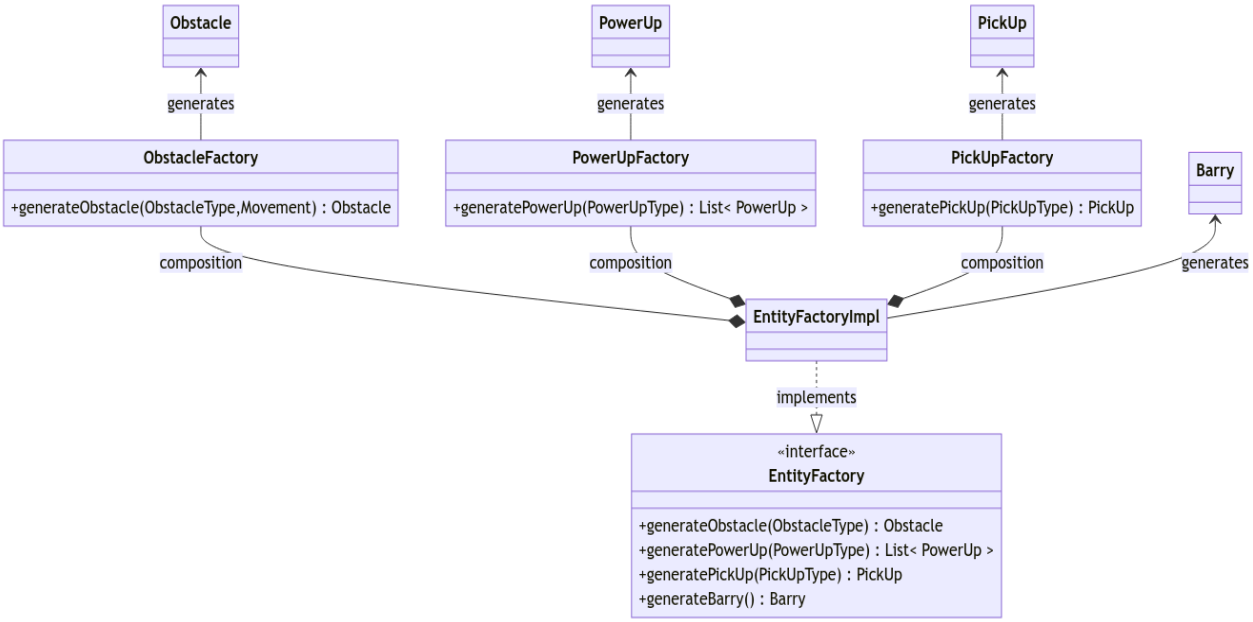

Figura 2.2.3.3: rappresentazione della struttura utilizzata per la creazione delle entità. Più factories specializzate per la generazione delle entità.

Problema

Jetpack Joyride deve essere in grado di generare un qualsiasi numero di entità in qualunque momento. La generazione delle entità deve essere facilitata e per quanto possibile automatizzata nella scelta delle caratteristiche di ogni entità.

Soluzione

L'idea è quella di poter utilizzare una serie di Simple Factory specializzate per generare uno specifico tipo di entità l'una, e riunire successivamente le chiamate ai loro metodi in una classe (EntityFactoryImpl) che quindi permette di generare una qualsiasi entità.
Questo meccanismo permette una loro costruzione semplificata, automatizzando il processo di assegnamento dell'hitbox e del movimento quando possibile. La scelta di utilizzare una factory che utilizza ulteriori factories è stata ponderata in base a diversi motivi: una struttura gerarchica di factories è semplice e flessibile se il numero delle diverse entità nel gioco è relativamente basso (ad esempio in questo software) e non si prevede l'aggiunta di ulteriori entità (anche se sarebbero facilmente implementabili grazie al modo in cui tutte le Entity sono organizzate).
Un'unica factory per generare tutte le entitá (invece che delegare la creazione a sotto-factories specializzate).

Gestione modelli entità

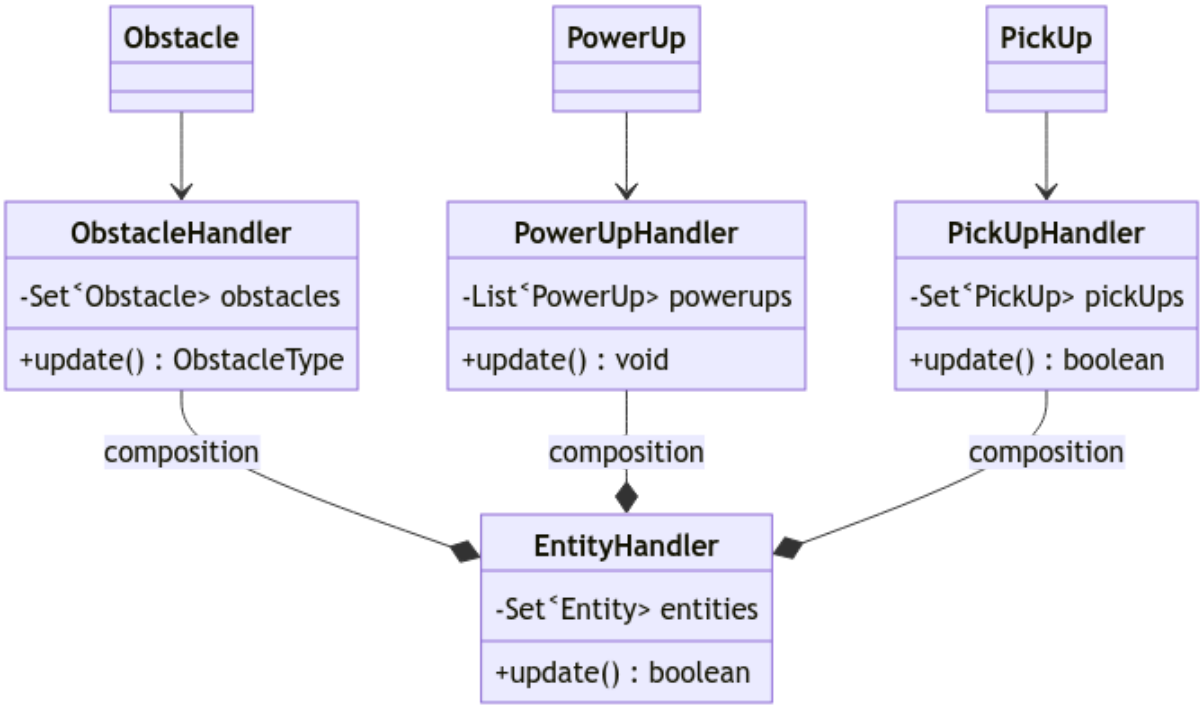

Figura 2.2.3.4: rappresentazione della struttura utilizzata per la gestione delle entità. Viene applicato il più possibile il Single Responsibility Principle delegando ad ogni classe un compito specifico.

Problema

JetPack Joyride deve gestire una moltitudine di entità nello stesso momento, ciascuna con proprie proprietà di movimento, stato, collisione… Deve essere inoltre garantita una corretta gestione delle interazioni tra le stesse.

Soluzione

Le classi Handler si occupano della gestione delle varie entità in gioco. Ogni tipologia di entità dispone di una propria classe handler che le raggruppa e semplifica la loro gestione. Gli handler mettono a disposizione un metodo update(boolean) che effettua l'aggiornamento di tutte le entità da essi gestiti (quindi aggiornano movimento, collisione, stato, etc…) e inoltre gestiscono la generazione e le interazioni tra loro e tra alcune entità di altri handler. Una classe EntityHandler si occupa poi di gestire tutti gli handler singoli coordinando le interazioni tra tutte le entità e notificando EntityManager, la classe che si occupa della coordinazione tra model e view, delle entità presenti in gioco.

Riuso del codice per la gestione della visualizzazione entità

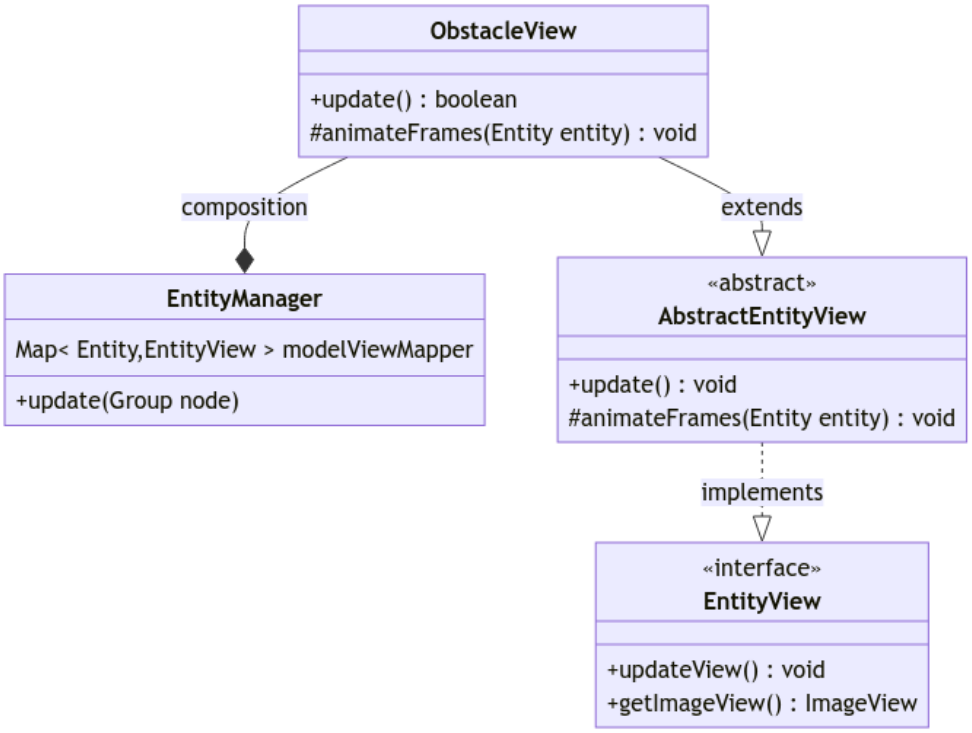

Figura 2.2.3.5: rappresentazione della struttura utilizzata visualizzare le entità.
Analogamente alla figura 2.2.3.1, sono state omesse le classi di animazione PowerUpView e PickUpView per semplificare l'UML, ma la loro implementazione è analoga a ObstacleView.

Problema
JJ deve permettere al giocatore di visualizzare correttamente in ogni istante tutte le entità presenti in gioco. JJ deve inoltre essere in grado di gestire tutte le immagini e le animazioni diverse associate ad ogni entità.

Soluzione (non implementata)
L'idea è quella di implementare un'unica classe per la gestione del caricamento delle immagini e una per il calcolo del frame di animazione adatto, magari attraverso l'utilizzo di un file. Questa soluzione è stata scartata a causa della troppa specificità della scelta dell'animazione in base ai diversi stati/comportamenti delle entità. Risulta più semplice anche se meno efficace a livello di riusabilità del codice implementare una view per ciascuna categoria di entità che presentano stati simili.

Soluzione
L'interfaccia EntityView definisce i metodi poi implementati dalla classe astratta AbstractEntityView che fornisce un'implementazione per scalare automaticamente le immagini in base alle dimensioni dello schermo. Questa classe viene estesa poi da una classe dedicata per ogni tipologia di entità che all'implementazione descritta aggiunge un metodo per animare correttamente le entità.
Viene quindi utilizzato un Template method (updateView()) definendo l'ordine in cui il processo di animazione, rescaling e assegnazione dell'immagine vanno effettuati.

Capitolo 3
Sviluppo

3.1 Testing automatizzato

JJ verifica attraverso dei test automatizzati la corretta implementazione delle varie componenti di gioco. Utilizzando la libreria JUnit vengono testati sia elementi di dominio del problema che le interfacce grafiche sviluppate, in particolare:

• MapBackgroundImplTest: testa se la mappa è in grado di ridimensionarsi in base alle informazioni sulle dimensioni dello schermo contenute nella classe GameInfo e se le due immagini che formano la mappa sono attaccate insieme in ogni momento.

• ButtonFactoryTest: testa la capacità di ButtonFactory di restituire un pulsante anche quando si verifica la mancanza di risorse dell'immagine.

•CoinShapeFactoryImplTest: testa che il metodo della classe CoinShapeFactory sia in grado di restituire sempre una lista che contiene le coordinate delle varie disposizioni di monete, e che le forme non appaiono in posizione sbagliata.

•GameStatsTest: testa il corretto funzionamento dei metodi di aggiornamento di GameStats e testa che il metodo utilizzato per il salvataggio funzioni correttamente.

• EntityTest: testa il corretto funzionamento di tutte le tipologie di oggetti che estendono l’interfaccia Entity (entità del gioco) e quindi in particolare le loro caratteristiche principali, movimento e collisione. Viene testata inoltre la factory per la generazione delle entità e viene fatta una semplice prova di collision checking.

• Player Test: fornisce metodi per testare la corretta inizializzazione del personaggio, e il corretto funzionamento delle relazioni con altre entità di gioco.

3.2 Note di sviluppo

3.2.1 Alessandro Valmori

Utilizzo di Stream e Lambda expressions

https://github.com/alevu3344/OOP23-JJ/blob/d942129454240dda7a7d2d3acaab616d7727ed07/src/main/java/it/unibo/jetpackjoyride/menu/shop/impl/ShopControllerImpl.java#L78C13-L78C87

https://github.com/alevu3344/OOP23-JJ/blob/d942129454240dda7a7d2d3acaab616d7727ed07/src/main/java/it/unibo/jetpackjoyride/core/movement/MovementModifierFactoryImpl.java#L96C1-L98C45 (Per git blame risulta appartenere al collega Gabriel, questo perchè la classe è stata realizzata in collaborazione in presenza con un solo pc)

https://github.com/alevu3344/OOP23-JJ/blob/d942129454240dda7a7d2d3acaab616d7727ed07/src/main/java/it/unibo/jetpackjoyride/core/handler/pickup/PickUpHandler.java#L116C13-L118C114

3.2.2 Yukai Zhou

Utilizzo di Stream e Lambda expressions

Permalink:
https://github.com/alevu3344/OOP23-JJ/blob/589ee5ad42584289d10adec78f5e25cb9eba9b2d/src/main/java/it/unibo/jetpackjoyride/core/entities/coin/impl/CoinGenerator.java#L204C6-L207C47

Permalink:
https://github.com/alevu3344/OOP23-JJ/blob/e6529c0726149ec650913478b76b9d8dc07f3265/src/main/java/it/unibo/jetpackjoyride/core/entities/coin/impl/CoinShapeFactoryImpl.java#L163C1-L172C57

Utilizzo di JavaFx per la creazione del menu

Permalink:
https://github.com/alevu3344/OOP23-JJ/blob/5a4a2debf101839986c0a39a1da29ac02b4d8170/src/main/java/it/unibo/jetpackjoyride/menu/menus/impl/GameMenuImpl.java#L6C1-L13C37

Utilizzo di java.util.logging.Logger
Permalink:
https://github.com/alevu3344/OOP23-JJ/blob/e6529c0726149ec650913478b76b9d8dc07f3265/src/main/java/it/unibo/jetpackjoyride/core/statistical/impl/GameStatsIO.java#L80C2-L82C80

Per l'implementazione del movimento dello sfondo in questo progetto, mi sono ispirato dal video **"Java 2D side scrolling Game Tutorial (part 1) - Scrolling background"**. Il video introduceva su come realizzare uno sfondo scorrevole utilizzando javax, ovvero l'uso di due immagini per creare un movimento di sfondo. Sebbene la mia implementazione finale sia stata realizzata utilizzando imageView di JavaFX, l'idea originale (cioè, utilizzare due immagini) di uno sfondo scorrevole deriva dal suddetto video.

3.2.3 Gabriel Stira

Utilizzo di Lambda e Stream ove possibile; qui solo alcuni esempi:
https://github.com/alevu3344/OOP23-JJ/blob/ba9b9a427a2e9b9147c0fc42bb8f68ca1ca7d454/src/main/java/it/unibo/jetpackjoyride/core/hitbox/api/AbstractHitbox.java#L99C9-L104C58

https://github.com/alevu3344/OOP23-JJ/blob/ba9b9a427a2e9b9147c0fc42bb8f68ca1ca7d454/src/main/java/it/unibo/jetpackjoyride/core/handler/obstacle/ObstacleLoader.java#L302C1-L312C50

https://github.com/alevu3344/OOP23-JJ/blob/ba9b9a427a2e9b9147c0fc42bb8f68ca1ca7d454/src/main/java/it/unibo/jetpackjoyride/core/handler/pickup/PickUpHandler.java#L120C1-L123C16

Utilizzo di Function, Predicate:
https://github.com/alevu3344/OOP23-JJ/blob/ba9b9a427a2e9b9147c0fc42bb8f68ca1ca7d454/src/main/java/it/unibo/jetpackjoyride/core/entities/obstacle/impl/Laser.java#L96C5-L100C6

https://github.com/alevu3344/OOP23-JJ/blob/ba9b9a427a2e9b9147c0fc42bb8f68ca1ca7d454/src/main/java/it/unibo/jetpackjoyride/core/movement/MovementModifierFactoryImpl.java#L29C1-L38C6

Utilizzo della libreria JavaFx, usata nelle classi componenti la view e nelle classi in cui avviene il caricamento delle immagini:
https://github.com/alevu3344/OOP23-JJ/blob/d942129454240dda7a7d2d3acaab616d7727ed07/src/main/java/it/unibo/jetpackjoyride/utilities/EntityImageLoader.java#L25C1-L26C33

Se uno sprite non viene trovato o caricato correttamente (nella classe EntityImageLoader), viene sostituito con un'immagine tutta rossa per poter comunque permettere di giocare e non bloccare tutto; un'esempio su come creare un'immagine tutta rossa usando Graphics è stato reperito dal sito web:
https://www.tabnine.com/code/java/methods/java.awt.image.BufferedImage/getGraphics

Capitolo 4
Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Alessandro Valmori

Per quanto riguarda il mio lavoro svolto, e specialmente come le parti da me svolte si relazionano alle parti dei miei colleghi, non posso che ritenermi soddisfatto di me stesso e del codice che ho scritto.
Le mie parti di progetto, insieme a quelle del collega Stira, hanno subito molto spesso dei rework ingenti, che però, nonostante il rallentamento causato, hanno contribuito sia alla mia esperienza educativa, sia a condurmi inevitabilmente alla realizzazione di codice il quanto più possibile coerente con la struttura generale, cosa avvenuta principalmente nei momenti finali della realizzazione.

Ci sono state alcune scelte di design che rimpiango, sia da parte mia nella mia parte, sia per quanto riguarda il quadro generale, che a mio parere sono il risultato di una concentrazione scarsa da parte di tutti sulla parte preliminare di design e pre-structuring, le quali hanno poi impattato negativamente alcune parti del software, compresa la mia, e che hanno causato un continuo riadattamento di varie sezioni del mio codice.

Nel complesso mi ritengo molto soddisfatto, nonostante le sopra citate mancanze, perché per quanto mi riguarda il mio impegno è stato massimo, e sono fiero di quello che è il risultato finale.

Per quanto mi riguarda, non ho intenzione di portare avanti questo progetto, poiché a mio avviso il mio obiettivo, insieme a quello complessivo, è stato raggiunto, ovvero creare un clone funzionante del gioco originale.

4.1.2 Yukai Zhou

Realizzare un gioco da zero in collaborazione con altri, utilizzando il linguaggio di Java, mi ha senza dubbio dato un’esperienza significativa e delle nuove conoscenze. Nello sviluppo del gioco, ogni problema superato mi ha permesso di approfondire e migliorare la mia comprensione delle conoscenze acquisite in precedenza.

Questa collaborazione mi ha fatto anche comprendere profondamente l'importanza dell'analisi, specialmente in un contesto di lavoro di gruppo. Nelle fasi iniziali dell'analisi, ho sottovalutato l'importanza della fase di design preliminare; non ho considerato a fondo le possibili intersezioni e influenze tra le diverse classi e non ho espresso bene la mia idea , portando a una certa confusione nella struttura del gioco e a una conseguente perdita di tempo per gli aggiustamenti. Inoltre, aver utilizzato per la prima volta JavaFX ha causato delle difficoltà nello sviluppo del gioco, che hanno successivamente portato alla realizzazione di una struttura di menu di gioco errata.

Guardando oggettivamente il mio lavoro, ci sono molti aspetti che potrebbero essere ottimizzati oppure per i quali potrebbero essere applicate soluzioni migliori. Tuttavia, sono relativamente soddisfatto del lavoro che ho completato; non ho incontrato difficoltà insormontabili, e anche se ci sono alcuni problemi irrisolti, sono solo questioni di tempo.

Infine, per quanto riguarda lo sviluppo futuro di questo progetto, credo che non sia necessario, poiché ritengo che abbiamo già realizzato con successo il nostro obiettivo iniziale, ovvero un clone di Jetpack Joyride.

4.1.3 Gabriel Stira

La realizzazione del progetto ha sicuramente contribuito in maniera importante al mio personale apprendimento del linguaggio Java, dell'importanza della parte di planning iniziale su cui basare il modo in cui un progetto è costruito (design) e della soluzioni da implementare per risolvere problemi di vario tipo.
È difficile fare un'autovalutazione oggettiva, almeno per quanto riguarda il mio caso, per diversi motivi.
Se nel complesso la mia parte può essere considerata realizzata in modo discreto (tra soluzioni adottare in base al design scelto e qualità del codice vero e proprio), ciò non significa che il livello di qualità sia uniforme in tutte le parti scritte da me. Infatti si può notare che mentre le parti iniziali sono realizzate scegliendo soluzioni non ottimali, realizzando codice qualitativamente scarso (mancanza di gestione dell'errore, mancanza di documentazione adeguata, poche scelte di design realizzate pensando alle implicazioni future, etc…), nelle ultime la qualità è invece molto più alta.
Questo fa notare il fatto che vi è stato nei mesi di sviluppo del progetto un graduale miglioramento dovuto al crescere delle mie conoscenze e capacità per quanto riguarda le parti precedentemente descritte.
Nel complesso, anche se i problemi che le parti iniziali avevano sono stati risolti riscrivendo il codice e ripensando a nuove idee implementative, purtroppo rimangono dei problemi globali causati da alcune scelte fatte da ognuno di noi nell'adattarsi alle parti degli altri e ciò causa poca coesione tra le parti di progetto sviluppate singolarmente, che interagiscono tra di loro in maniera poco efficiente.
Il gioco risulta comunque giocabile e non solo le parti obbligatorie, ma anche molte di quelle opzionali sono state realizzate e ciò fa del risultato complessivo qualcosa di abbastanza soddisfacente.
Non è prevista alcuna modifica futura almeno per quanto mi riguarda; questo non perché sia difficile provare ad implementare nuovi oggetti in maniera da adattarli a quelli già presenti (la mia parte è stata infatti realizzata provando ad essere ampliata dinamicamente il più possibile a proprio piacimento senza problemi), ma perché ritengo di aver completato ciò che era stato deciso come obiettivo iniziale, realizzare un clone di Jetpack Joyride, niente di più, niente di meno.

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Alessandro Valmori

La realizzazione di questo progetto è stata un’esperienza educativa per me importantissima, ho imparato prima di tutto il corretto utilizzo di Git fra collaboratori di uno stesso progetto, e soprattutto mi ha permesso di capire a pieno i diversi meccanismi della programmazione ad oggetti e in particolare di Java.
Durante la realizzazione del software, ho incontrato numerose difficoltà che mi sono a volte parse come insormontabili.
Essendo questo il mio secondo anno di programmazione, e il mio primo software realizzato da zero, mi sono spesso trovato in situazioni di stallo e di confusione. Ma nonostante i vari momenti di difficoltà, l’aiuto tra tra colleghi non è mai mancato, e sono riuscito infine a portare a termine la mia parte in maniera buona.

4.2.2 Gabriel Stira

Il progetto ha subito dei catastrofici momenti di bassi e dei produttivi momenti di alti.
Numerosi problemi si sono presentati per i più svariati motivi e ciò è la causa principale per cui il progetto se nelle parti realizzate può essere considerato realizzato discretamente bene, nelle parti di "allacciamento" e interazione tra le singole, l'implementazione è stata fatta in maniera poco efficiente, poco leggibile e poco coesa (specialmente nella parte di scelta del pattern architetturale da seguire)
Il problema più grande almeno per quanto mi riguarda, è stato il fatto di esserci ritrovati con un membro in meno a metà del tempo previsto per la realizzazione del software e dal momento che la sua parte comprendeva implementazione del personaggio e dello shop, due elementi che interagiscono molto con ostacoli e powerups di cui mi sono occupato io, e che al momento del ritiro dal progetto non è stata lasciato niente (nemmeno una base di partenza o un po' di codice scritto) di queste, mi sono ritrovato effettivamente a "navigare nel buio". Non avere nemmeno una base di realizzazione di queste entità mi ha costretto a provare ad "indovinare" quali comportamenti e implementazioni avrebbero avuto queste una volta completate.
Voglio menzionare il fatto che queste parti fondamentali non implementate dal collega che ha abbandonato siano state poi realizzate dal collega Alessandro (dal momento che quelle che doveva realizzare lui erano meno fondamentali) che quindi ha dovuto recuperare quello che non era stato fatto nel poco tempo rimanente.
Altri problemi di minore importanza (incomprensioni tra i membri del gruppo, idee completamente sbagliate che sono state da realizzare daccapo, etc..) si sono poi presentati ma sono stati superati nei modi più ottimali possibili.

Appendice A
Guida utente

L’applicazione all’avvio mostra un menu principale intuitivo con un pulsante per cominciare il gioco e uno per accedere al negozio.

Una volta cominciato il gioco, ci si muove soltanto con l’uso della barra spaziatrice, che muove il personaggio e i veicoli powerup in maniera unica fra questi.

Durante il corso della partita è possibile raccogliere due tipi di pickup, un veicolo oppure uno scudo raccoglibile.

Se si colpisce un ostacolo avendo uno scudo o un veicolo equipaggiati, questi si distruggono e si ritorna a controllare Barry.

Nel negozio è possibile cliccare sui pulsanti per sbloccare qualsiasi veicolo e lo scudo. Nello shop, inoltre è presente un meccanismo per sbloccare un powerup segreto, ovvero digitare (col focus sulla finestra di gioco) la password “truffleworm” : ciò sbloccherà il powerup Duke Fishron.