

Relazione del progetto di
“Programmazione ad Oggetti”

Lorenzo Antonioli
Eleonora Bianco
Giulia Fares
Luca Varale Rolla

18 giugno 2025

Indice

1 Analisi	3
1.1 Descrizione e requisiti	3
1.1.1 Requisiti funzionali	4
1.1.2 Requisiti non funzionali	4
1.2 Modello del dominio	4
2 Design	6
2.1 Architettura	6
2.2 Design dettagliato	8
2.2.1 Lorenzo Antonioli: implementazione mappa	8
2.2.2 Eleonora Bianco	16
2.2.3 Giulia Fares	25
2.2.4 Luca Varale Rolla	29
3 Sviluppo	32
3.1 Testing Automatizzato	32
3.2 Note di Sviluppo	33
3.2.1 Lorenzo Antonioli	33
3.2.2 Eleonora Bianco	33
3.2.3 Giulia Fares	34
3.2.4 Luca Varale Rolla	34
4 Commenti finali	36
4.1 Autovalutazione e lavori futuri	36
4.1.1 Lorenzo Antonioli	36
4.1.2 Eleonora Bianco	36
4.1.3 Giulia Fares	37
4.1.4 Luca Varale Rolla	37
A Guida utente	38
A.1 Menù Iniziale	38

A.2	Shop	39
A.3	Play	39
A.4	Gameover	41
B	Esercitazioni di Laboratorio	42
B.0.1	lorenzo.antonio2@studio.unibo.it	42

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Il gruppo si pone l'obiettivo di realizzare un videogioco ispirato a Crossy Road. Il giocatore dovrà attraversare strade, binari e fiumi evitando ostacoli mobili come auto e treni. L'obiettivo del gioco è cercare di raggiungere la distanza più lunga possibile evitando collisioni e cadute. Sarà presente un negozio con la possibilità di modificare il personaggio utilizzando le monete raccolte durante il gioco.



Figura 1.1: Immagine del gioco "Crossy Road"

1.1.1 Requisiti funzionali

- All'apertura del gioco apparirà una schermata iniziale in cui l'utente potrà consultare il regolamento, iniziare una nuova partita e acquistare delle skin (personalizzazione del personaggio).
- Il personaggio si potrà muovere nelle quattro direzioni.
- Saranno presenti ostacoli mobili e fissi.
- La visuale di gioco avanza costantemente, indipendentemente dal movimento del personaggio
- Il gioco gestirà la morte del personaggio.
- Lungo la mappa saranno presenti degli oggetti che il personaggio potrà raccogliere per ottenere una vita aggiuntiva.
- Assicurare il salvataggio del punteggio massimo raggiunto e delle monete raccolte

1.1.2 Requisiti non funzionali

- L'interfaccia dovrà essere semplice e intuitiva.
- Si cercherà di ottenere la massima fluidità di gioco.

1.2 Modello del dominio

L'utente controllerà un personaggio all'interno di una mappa infinita che avanza a ritmo costante. L'obiettivo del gioco è quello di avanzare nella mappa il più possibile. Durante l'avanzamento, il giocatore dovrà evitare degli ostacoli mobili (auto e treni) e fissi (alberi e fiumi). La mappa è composta da righe di terreno di diverso tipo: fiume, erba, strada e ferrovia. Gli alberi sono presenti solo nei terreni di erba e non permettono al player di avanzare nella cella in cui questi sono presenti. Nelle strade e nelle ferrovie saranno presenti rispettivamente le automobili e i treni. Quando il personaggio arriva a un fiume, per attraversarlo, dovrà salire su dei tronchi mobili che gli permetteranno di non cadere in acqua. Dopo un certo intervallo di tempo, la velocità di scorrimento della mappa e degli ostacoli mobili aumenterà, rendendo l'avanzamento del giocatore più difficile. Il personaggio muore quando viene colpito da un ostacolo mobile, quando cade nel fiume e se la visuale di gioco supera la sua posizione. Lungo il percorso saranno posizionate delle

monete che, se raccolte, si potranno utilizzare nel negozio. Inoltre saranno presenti degli oggetti collezionabili che offriranno una seconda vita al player.

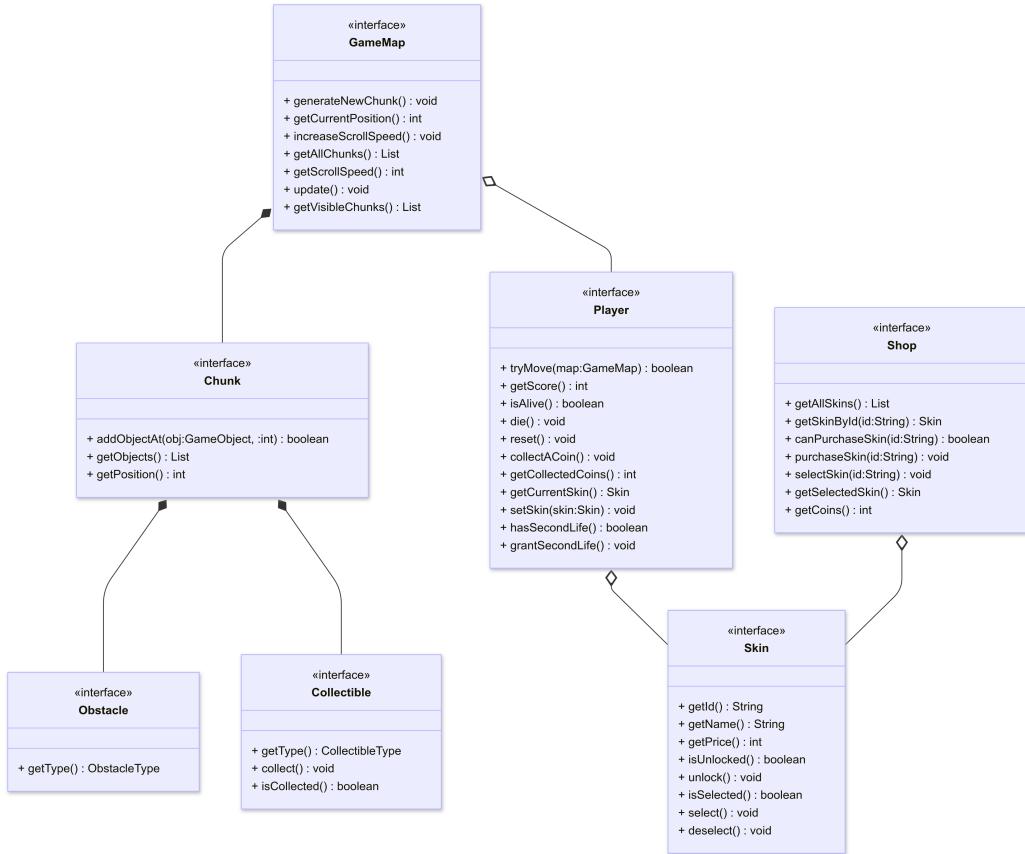


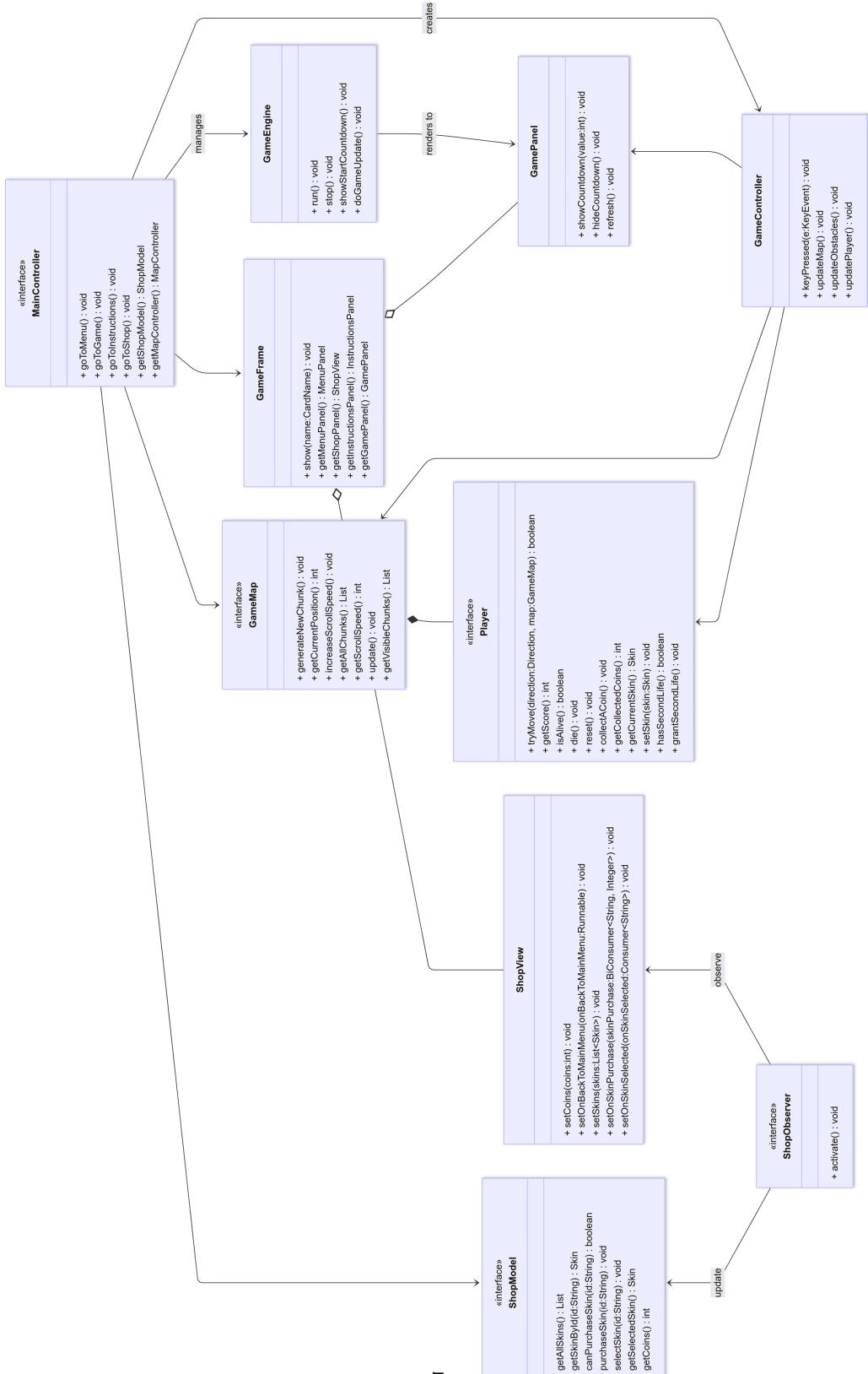
Figura 1.2: Schema UML dell’analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

Per lo sviluppo del progetto è stato adottato il pattern architetturale Model-View-Controller (MVC), con l'obiettivo di mantenere ben separati il model, la view e il controller. Il model gestisce la logica e i dati del gioco ed è organizzato all'interno di un'apposita cartella che contiene tutte le classi dedicate. La view si occupa della rappresentazione grafica dell'interfaccia. Anche la view è strutturata in una cartella dedicata. Infine, il controller funge da intermediario tra model e view, garantendo la comunicazione tra i due. Questa suddivisione permette di ottenere un'architettura modulare e facilmente estendibile, in cui la sostituzione della vista non comporta modifiche al modello o al controller, migliorando così la manutenibilità e la scalabilità del software.



2.2 Design dettagliato

In questa sezione ogni componente del gruppo descriverà più in dettaglio il design di cui si è occupato.

2.2.1 Lorenzo Antonioli: implementazione mappa

Composizione della mappa

Problema La mappa è composta da più tipi di terreno (Grass, River, Road, Railway).

Soluzione Ho suddiviso la mappa in una struttura gerarchica a tre livelli:

- GameMap: Rappresenta l'intera mappa di gioco
- Chunk: Rappresenta righe di terreno di diversi tipi
- Cell: Rappresenta le singole celle che compongono ogni chunk

La mappa è quindi organizzata come una griglia bidimensionale dove ogni chunk è una riga orizzontale composta da celle. Questo approccio permette una gestione efficiente dello scrolling verticale e del posizionamento degli oggetti.

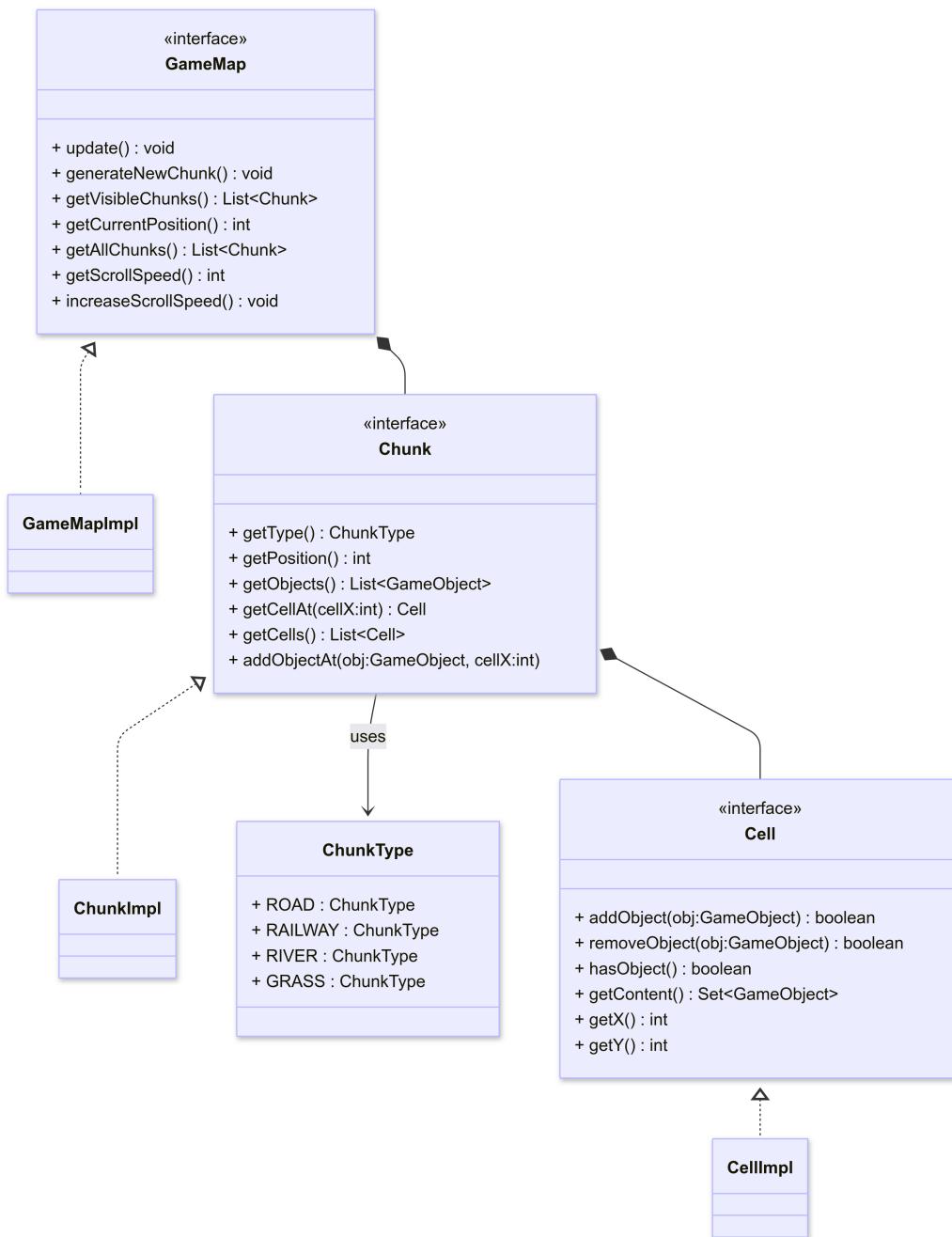


Figura 2.1: Schema UML di composizione della mappa di gioco

Generazione dinamica dei Chunk

Problema La mappa richiede la generazione di chunk eterogenei. È necessario evitare duplicazione di codice e semplificare l'aggiunta di nuovi tipi di chunk.

Soluzione Ho scelto di utilizzare il pattern Factory Method, implementato tramite l'interfaccia ChunkFactory e la classe ChunkFactoryImpl. Ogni metodo (es. createGrassChunk, createFirstChunk) incapsula la logica di costruzione di un tipo specifico di chunk.

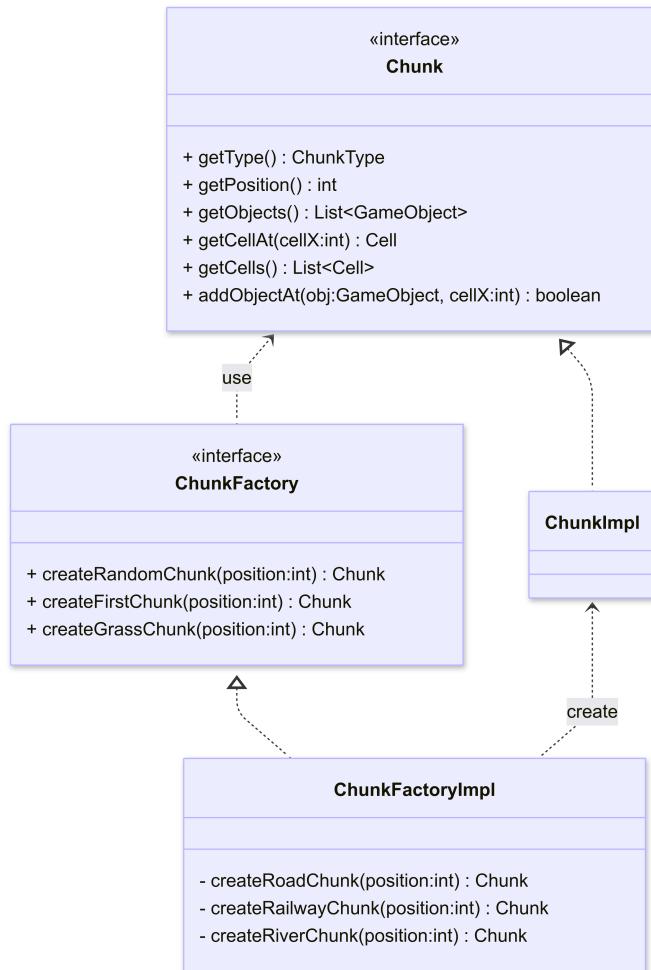


Figura 2.2: Diagramma UML del pattern Factory Method per la creazione dei Chunk

Piazzamento degli ostacoli e dei collezionabili

Problema Posizionamento degli ostacoli fissi e dei collezionabili all'interno delle celle nei Chunk.

Soluzione Ho utilizzato il pattern Strategy, creando l'interfaccia Object-Placer e la relativa implementazione. Ad ogni creazione di un chunk, in base al tipo di chunk, vengono o no chiamati i due metodi per posizionare gli oggetti nelle celle del Chunk.

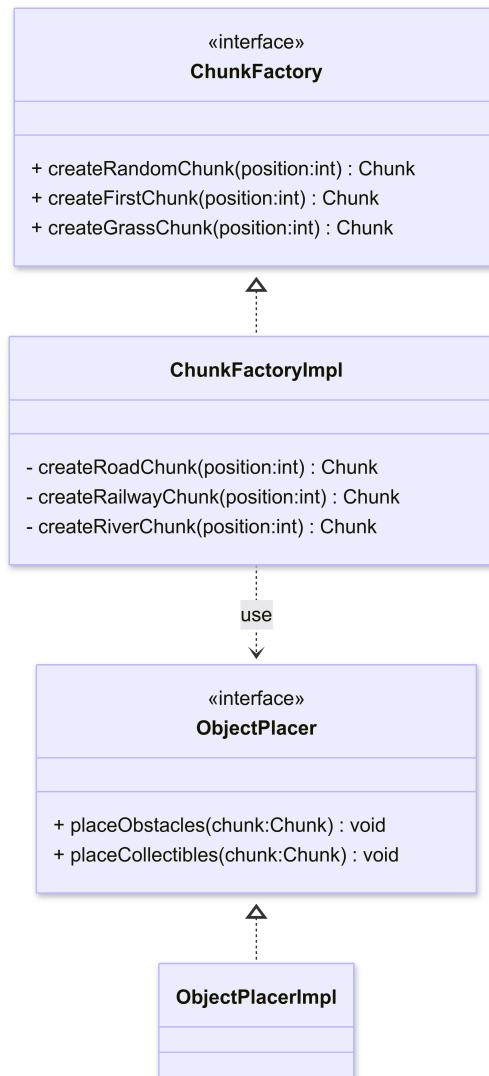


Figura 2.3: Schema UML del pattern Strategy per la gestione del posizionamento

Gestione ostacoli fissi e oggetti collezionabili

Problema Nei diversi chunk della mappa, devono essere presenti alcuni ostacoli fissi e oggetti collezionabili

Soluzione Ho creato un'entità GameObject generale dalla quale, per ereditarietà, creo le entità Collectible e Obstacle

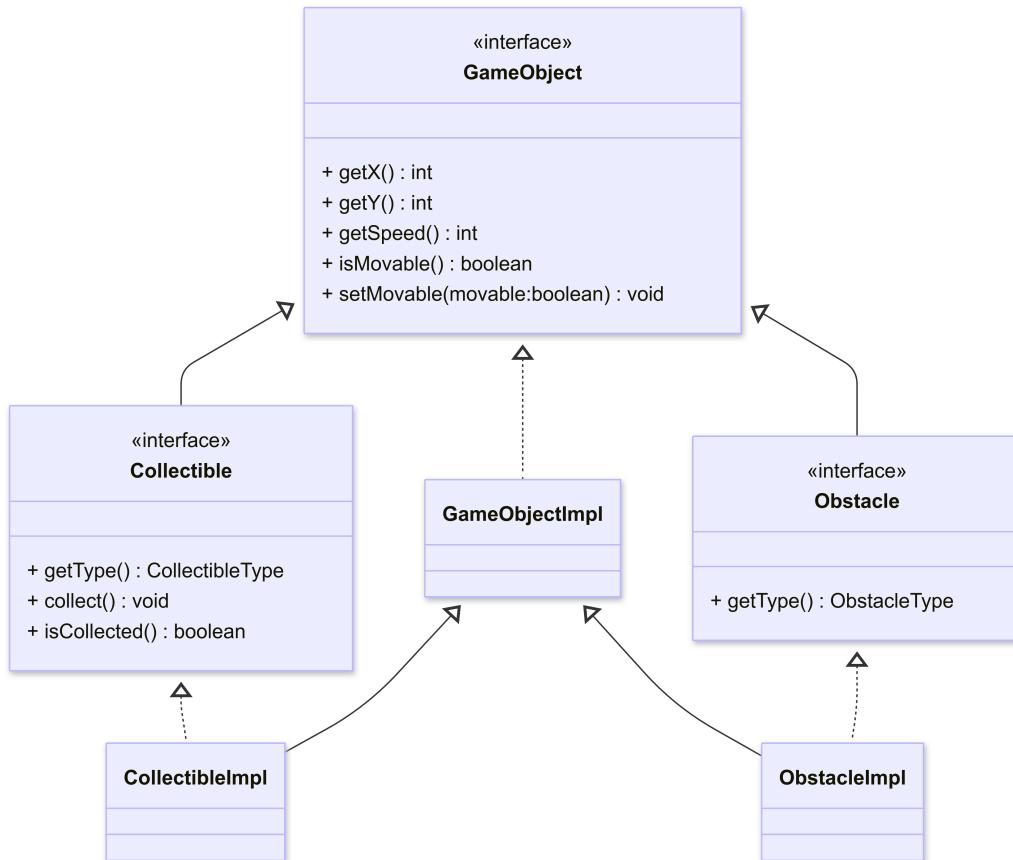


Figura 2.4: Schema UML della gestione degli ostacoli fissi e degli oggetti collezionabili

Gestione tipologie di ostacoli fissi e oggetti collezionabili

Problema Rappresentare le diverse tipologie di Chunk, ostacoli fissi e oggetti collezionabili in modo flessibile e scalabile.

Soluzione Invece di utilizzare enum tradizionali (che risulterebbero troppo rigide e poco flessibili per future estensioni), ho creato classi di utilità implementate come classi finali con costruttore privato e istanze statiche pubbliche. Questo approccio garantisce un insieme chiuso e immutabile di valori possibili mantenendo al contempo la flessibilità per future estensioni. Le classi implementate sono:

- **ChunkType**: Definisce i tipi di chunk (ROAD, RAILWAY, RIVER, GRASS)
- **ObstacleType**: Definisce i tipi di ostacoli (CAR, TRAIN, TREE, WATER, LOG)
- **CollectibleType**: Definisce i tipi di oggetti collezionabili (COIN, SECOND LIFE)

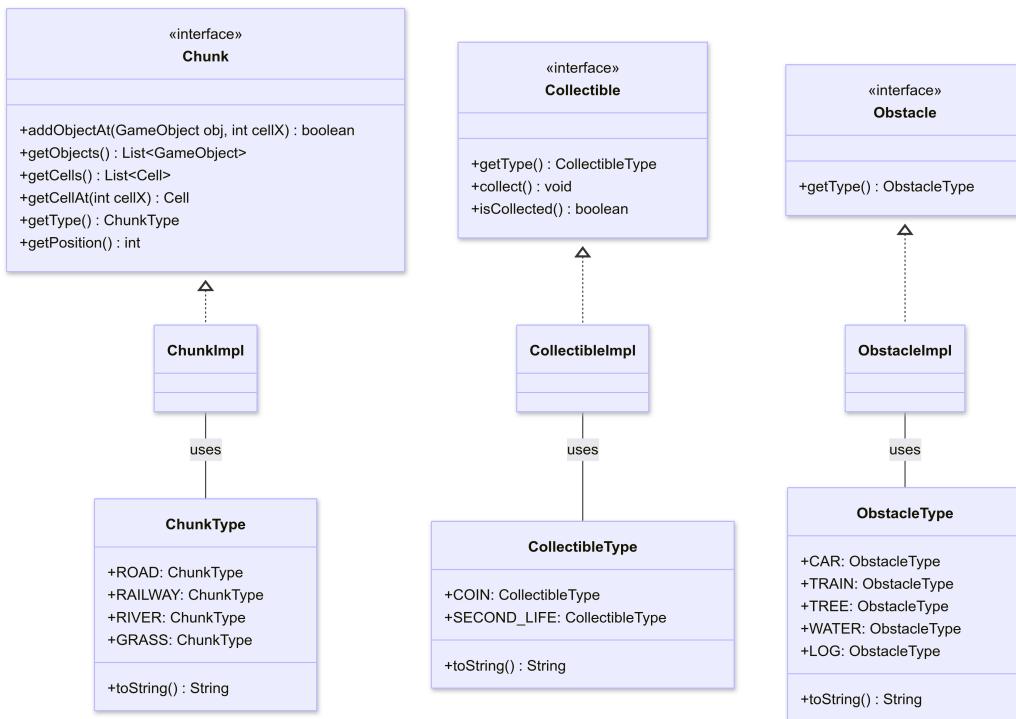


Figura 2.5: Schema UML raffigurante la gestione delle tipologie di Chunk, collezionabili e ostacoli fissi

Gestione dello stato di pausa e di game

Problema Si deve gestire la situazione in cui il gioco è in pausa o è in esecuzione. Il sistema necessita di comportamenti differenti a seconda dello stato corrente: durante il gameplay normale devono essere aggiornati gli ostacoli, il giocatore e la mappa, mentre durante la pausa il gioco deve rimanere fermo e mostrare le opzioni per riprendere o uscire.

Soluzione Ho implementato lo State Pattern per rappresentare i diversi stati del gioco. Questo pattern permette di encapsulare i comportamenti specifici di ogni stato in classi separate, rendendo il codice più modulare e facilmente estendibile. L'implementazione prevede:

- GameState: interfaccia che definisce i metodi comuni a tutti gli stati
- OnGameState: implementazione concreta che gestisce lo stato di gioco attivo, eseguendo gli aggiornamenti della logica di gioco
- PauseState: implementazione concreta che gestisce lo stato di pausa, mostrando il pannello di pausa e gestendo le azioni di ripresa o uscita
- StateName: classe di utilità per identificare univocamente i diversi stati tramite costanti

Il GameEngine mantiene un riferimento al GameState corrente e delega ad esso la scelta delle operazioni di aggiornamento e rendering

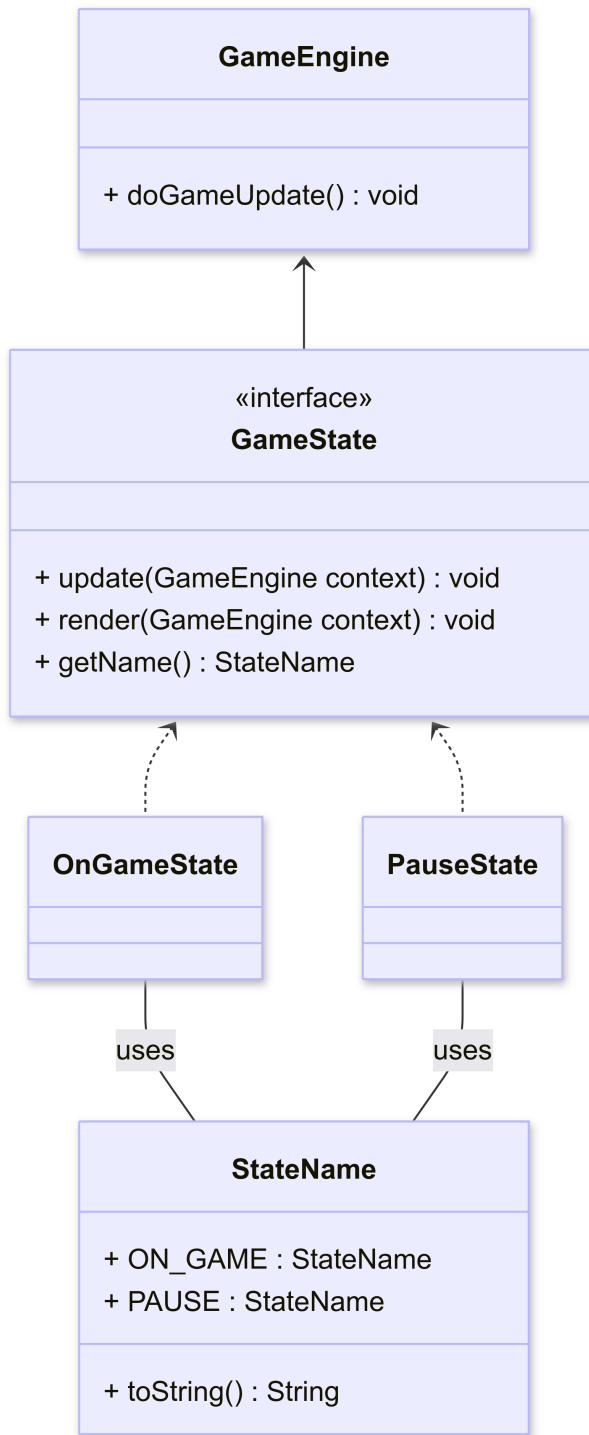


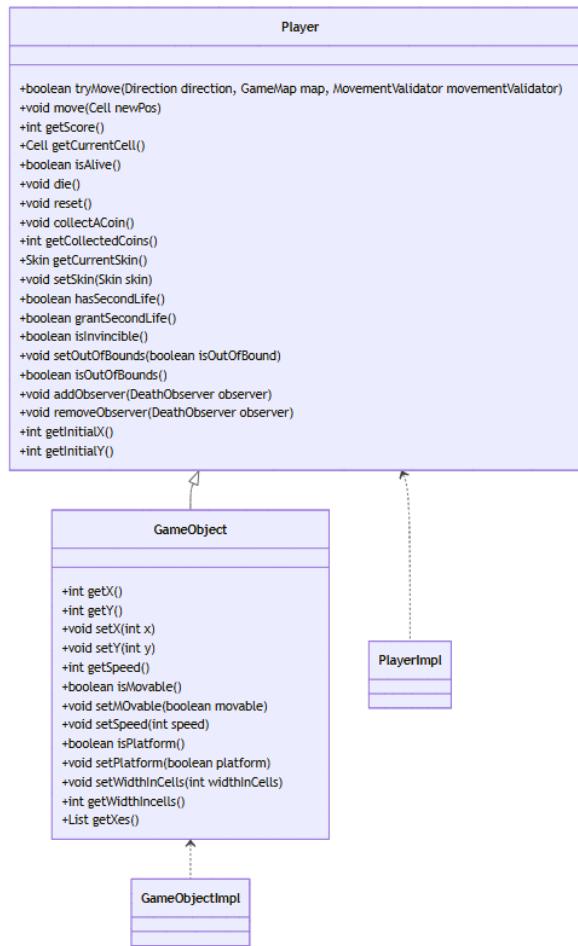
Figura 2.6: Schema UML rappresentante i GameState

2.2.2 Eleonora Bianco

Rappresentazione del Player all'interno del gioco

Problema Definire in modo strutturato come rappresentare il Player all'interno del gioco.

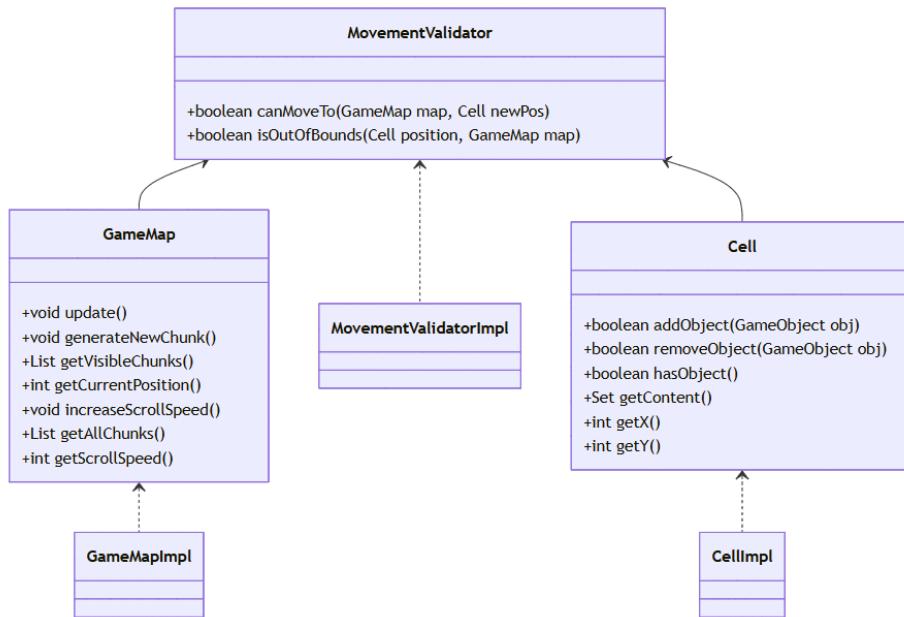
Soluzione Ho implementato una classe Player che estende GameObject, ereditandone i metodi comuni per la rappresentazione nel gioco. Alla classe ho aggiunto tutte le funzionalità specifiche del Player, come la possibilità di morire e di raccogliere i collectible.



Gestione del movimento del player

Problema È necessario verificare che ogni movimento del Player sia valido prima di eseguirlo.

Soluzione Ho utilizzato il pattern Strategy per delegare la logica di validazione del movimento a una componente dedicata. In particolare, ho definito l'interfaccia MovementValidator e la sua implementazione MovementValidatorImpl, incaricate di verificare la validità della cella di destinazione prima che il Player possa spostarsi. Il controllo accerta che la cella non sia occupata da un oggetto non attraversabile (come un albero, TREE) e che rientri nell'area visibile della mappa.

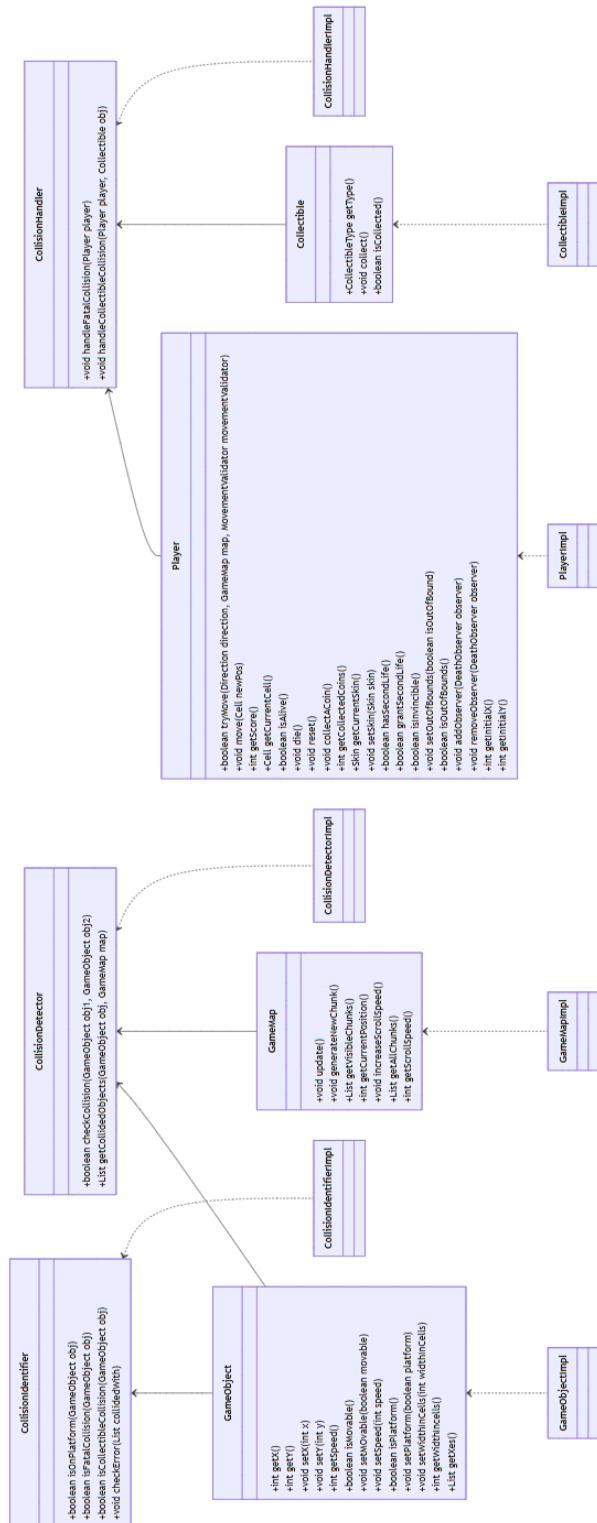


Gestione delle collisioni

Problema Il Player deve poter interagire con vari oggetti presenti nella mappa e reagire correttamente a ogni tipo di collisione.

Soluzione Per mantenere il codice organizzato e rispettare il principio di separazione dei compiti, ho suddiviso la gestione delle collisioni in tre componenti principali:

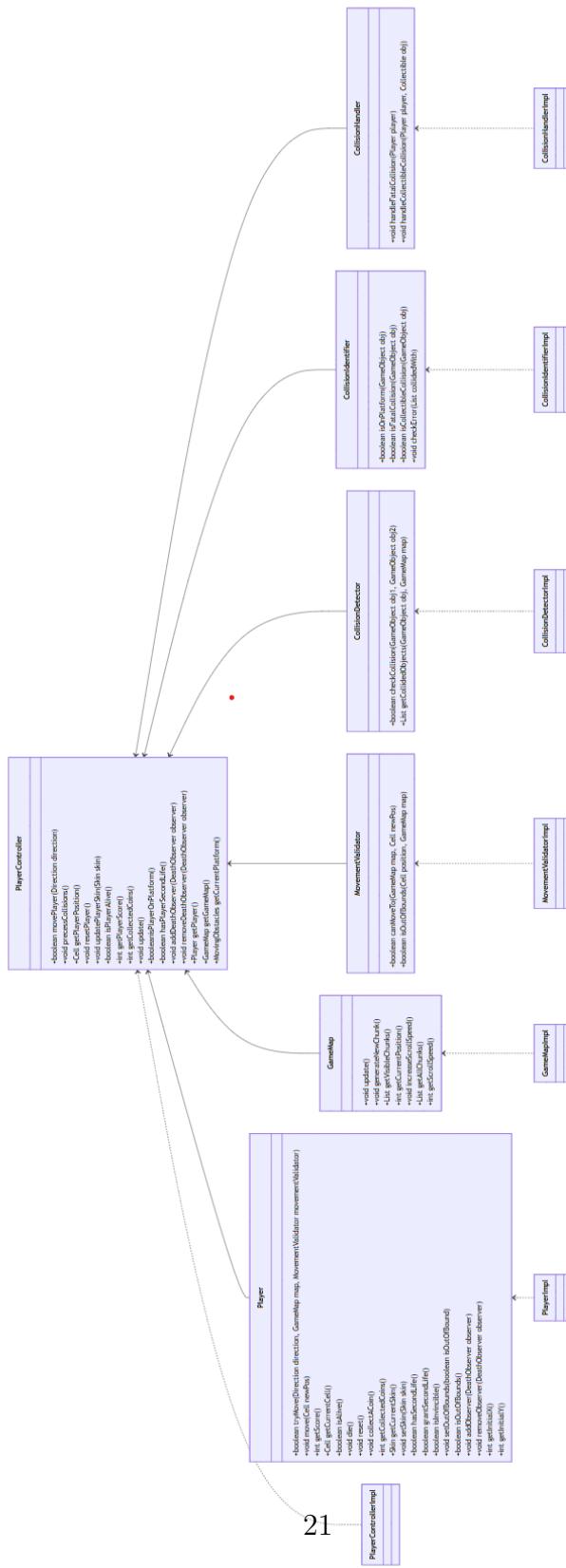
- **CollisionDetector / CollisionDetectorImpl**: rileva se vi è una collisione tra GameObject nella cella corrente e restituisce una lista degli oggetti coinvolti nella collisione.
- **CollisionIdentifier / CollisionIdentifierImpl**: identifica la tipologia di ogni oggetto in collisione, assicurandosi che il Player non interagisca con oggetti con cui non può collidere.
- **CollisionHandler / CollisionHandlerImpl**: gestisce la collisione in base alla sua natura. In caso di collisione fatale, invoca il metodo di morte del Player. Se la collisione avviene con un Collectible, richiama il metodo appropriato nel Player e marca l'oggetto come raccolto.



Gestione effettiva del player all'interno della mappa

Problema Era necessario integrare il comportamento del Player all'interno della mappa, gestendo correttamente le dinamiche di gioco.

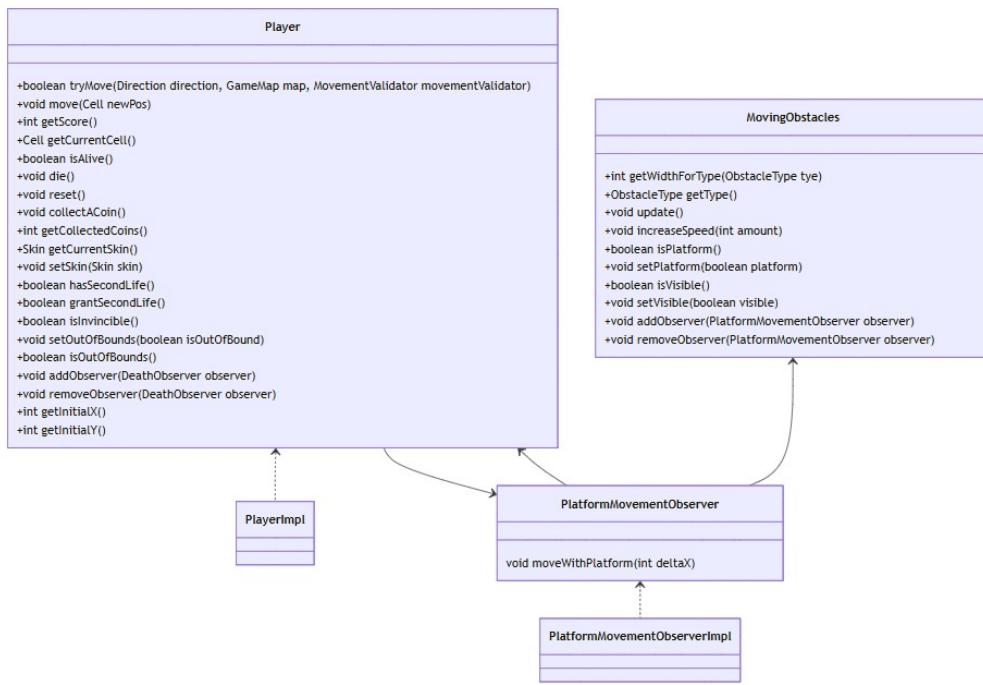
Soluzione Ho implementato l'interfaccia PlayerController e la classe PlayerControllerImpl, responsabili della gestione del Player nella mappa. Questa classe integra le logiche di collisione e si occupa di aggiornare continuamente lo stato del Player. Inoltre, verifica se il Player è fermo in una cella e, se investito o uscito dalla mappa, ne provoca la morte.



Gestione dell'interazione tra Player e Tronchi

Problema Se il Player si trova su un fiume senza essere su un tronco deve annegare; se invece è su un tronco, deve essere trasportato con esso.

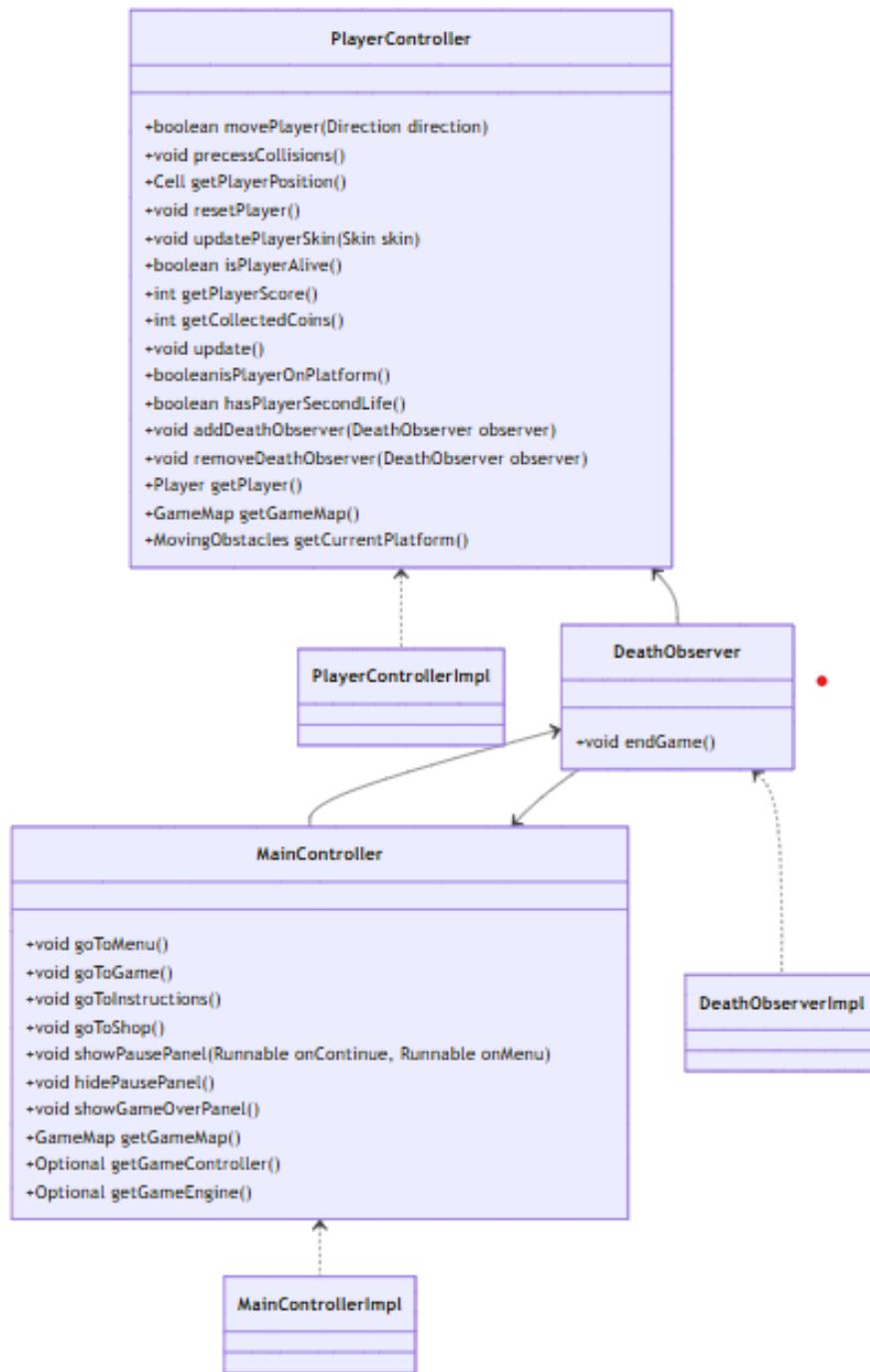
Soluzione In PlayerControllerImpl ho aggiunto un metodo privato che verifica se il Player si trovi in acqua senza supporto, causando la sua morte. Inoltre, ho utilizzato il pattern Observer, creando il PlatformMovementObserver, che viene collegato e scollegato dinamicamente ai tronchi su cui il Player sale. Questo permette di trascinare il Player ogni volta che il tronco si muove.



Apertura della schermata GameOver alla morte del Player

Problema Alla morte del Player, è necessario chiudere la schermata di gioco e mostrare la schermata di GameOver.

Soluzione Anche in questo caso ho applicato il pattern Observer, implementando DeathObserver. Questo osservatore viene collegato al Player attraverso il PlayerController durante l'inizializzazione delle componenti di gioco in MainControllerImpl, e viene notificato quando la variabile isAlive del Player diventa falsa. In risposta, viene invocato il metodo showGameOverPanel().



2.2.3 Giulia Fares

Definizione degli ostacoli mobili

Problema Ogni ostacolo mobile è una singola entità grafica e logica, deve essere in grado di muoversi in autonomia e interagire con il mondo di gioco. È necessario gestire il comportamento dei tronchi affinché possano fungere da piattaforme mobili sulle quali il giocatore può sostare.

Soluzione La classe `MovingObstacles` implementa l'interfaccia `Obstacle` e rappresenta una singola entità mobile. Ogni oggetto contiene informazioni su posizione, tipo, larghezza in celle, velocità e visibilità. Il metodo `update()` permette di aggiornare la posizione dell'ostacolo in base alla sua velocità. Ho aggiunto a `GameObject` il metodo `getWidthInCells()` e che restituisce la larghezza dell'oggetto in celle della griglia di gioco, utile per gestire il movimento. La classe funge da soggetto osservabile (*Observable*) nel pattern `Observer`, è usato per notificare i movimenti dei tronchi. Questa struttura modulare facilita l'interazione con il gestore degli ostacoli e consente un'implementazione flessibile.

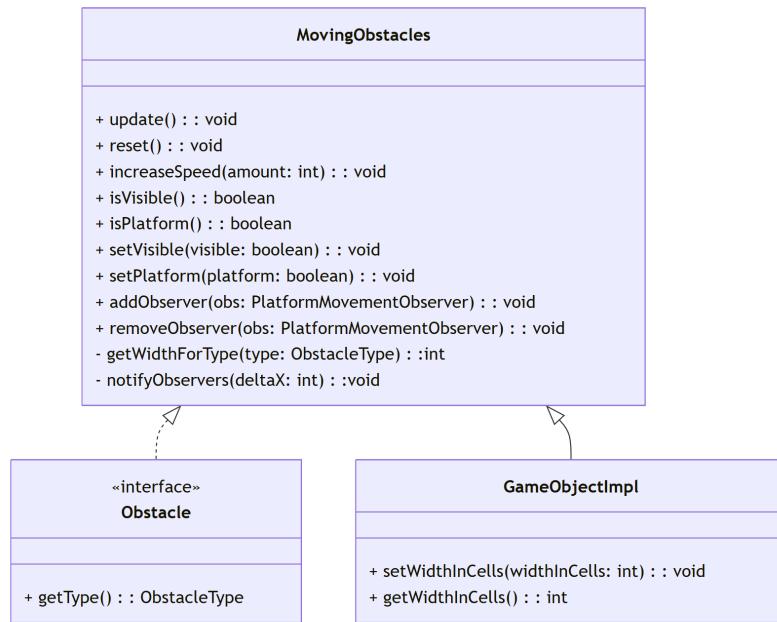


Figura 2.7: Schema UML della rappresentazione di un ostacolo mobile con logica di aggiornamento, visibilità e osservazione

Generazione dinamica degli ostacoli

Problema Popolare dinamicamente il gioco, disporre di un sistema di creazione automatica che possa generare oggetti coerenti per tipo, posizione, direzione e velocità.

Soluzione È stato adottato il Factory Method Pattern tramite l’interfaccia `MovingObstacleFactory`, che dichiara metodi per creare istanze di `MovingObstacles`. L’implementazione `MovingObstacleFactoryImpl` fornisce un metodo specifico per ogni tipo di ostacolo e un metodo generico `createObstacleByType()` per dispatch dinamico. Si appoggia a `SpeedConfig` per generare casualmente la velocità di ciascun ostacolo. Espone `getRandomSpeed(type)` per recuperare la velocità configurata in base al tipo, e `increaseSpeedLimits(amount)` per innalzare il tetto massimo di velocità di tutte le categorie.

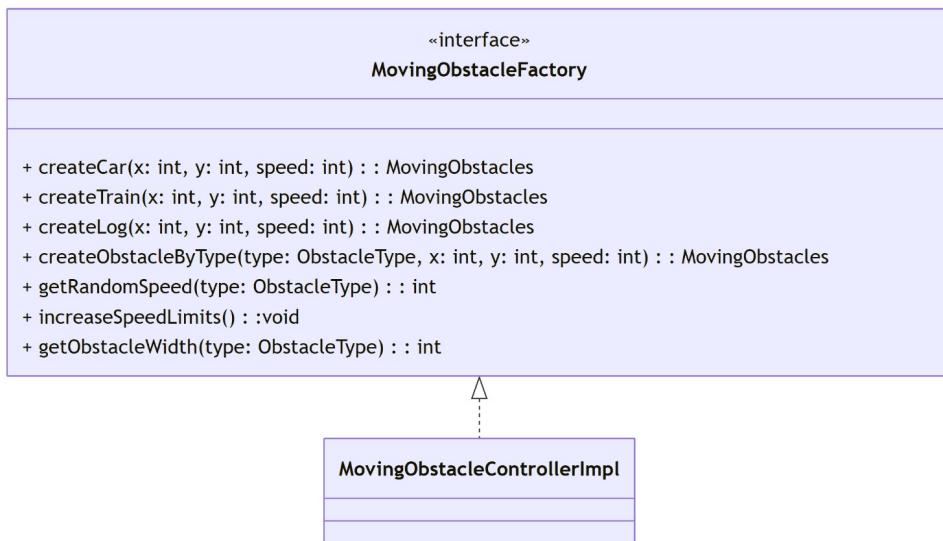


Figura 2.8: Schema UML dell’implementazione del Factory Method Pattern per la generazione di ostacoli mobili

Gestione degli ostacoli mobili

Problema Gestire dinamicamente un insieme di ostacoli mobili, ognuno dei quali si muove in una direzione e con una velocità prestabilita, mantenere aggiornate le posizioni e rimuovere gli ostacoli usciti dall'area visibile.

Soluzione La classe `MovingObstacleManagerImpl` gestisce l'aggiunta degli ostacoli tramite `addObstacle()` che controlla che la posizione non si sovrapponga a quelli esistenti. Il metodo `updateAll()` consente di aggiornare ogni ostacolo a ogni tick di gioco. La classe gestisce anche la rimozione degli oggetti non più presenti nell'area visibile tramite `cleanupOffscreenObstacles()` e consente l'aumento di velocità degli ostacoli per incrementare la difficoltà. L'approccio adottato consente una gestione modulare ed efficiente della dinamica degli ostacoli, semplifica l'integrazione con il ciclo di gioco principale.

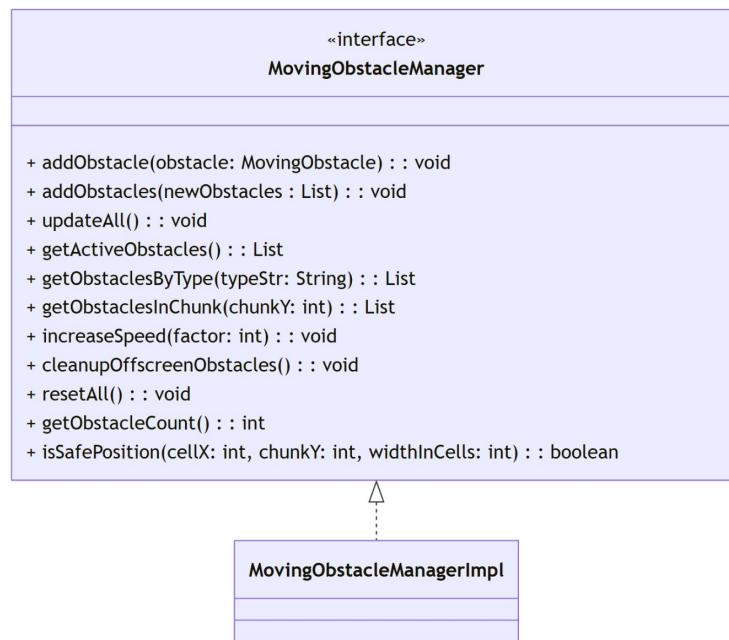


Figura 2.9: Schema UML della classe responsabile della gestione e aggiornamento centralizzato degli ostacoli mobili

Controllo della generazione e aggiornamento degli ostacoli

Problema Il sistema deve generare automaticamente ostacoli mobili nei chunk visibili della mappa, in base al tipo di terreno e mantenendo coerenza all'interno di ciascun chunk.

Soluzione La classe `MovingObstacleControllerImpl`, coordina la generazione e l'aggiornamento degli ostacoli mobili. Interagisce con la factory per creare gli ostacoli in base al tipo di chunk, con una certa velocità e direzione e con il manager per aggiungerli alla gestione centrale. Il metodo `update()` effettua un controllo dei chunk attualmente visibili e valuta se è necessario generare nuovi ostacoli in base alla spaziatura tra questi. Gestisce il ripristino di direzioni, velocità e lista degli ostacoli quando inizia il gioco. Il metodo `increaseAllObstaclesSpeed(amount)` aumenta la velocità degli oggetti già presenti sul terreno di gioco. Questo approccio permette una generazione controllata e un aggiornamento continuo mantenendo coerenza e modularità nell'integrazione con il gioco.

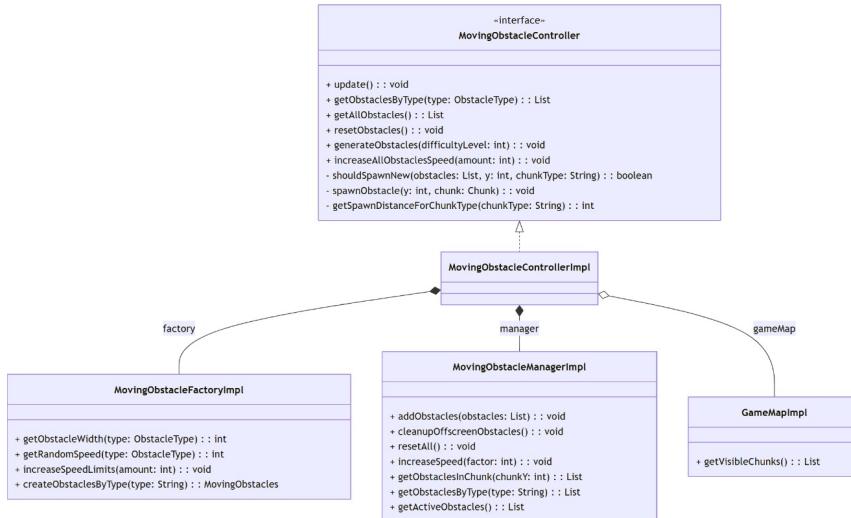


Figura 2.10: Schema UML del `MovingObstacleControllerImpl`, che coordina la generazione e l'aggiornamento degli ostacoli mobili sulla base della mappa

2.2.4 Luca Varale Rolla

Sistema di persistenza dei dati anche alla chiusura dell'app

Problema I dati del negozio (skin possedute, skin selezionata, monete) vengono resettati ad ogni chiusura dell'applicazione ma devono essere salvati tra le sessioni di gioco.

Soluzione Ho implementato un sistema di persistenza utilizzando file Properties Java, che permette di salvare i dati su file esterno che ho inserito nella cartella resources, e caricare automaticamente lo stato del negozio.

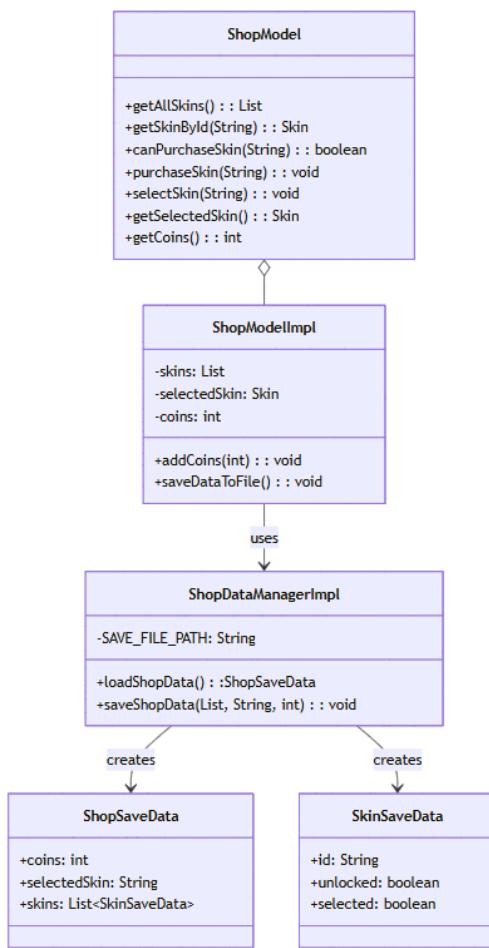


Figura 2.11: Schema UML della gestione del salvataggio dei dati di gioco su file.

Gestione del negozio delle Skin

Problema Il gioco necessita di un sistema di negozio dove i giocatori possono acquistare e selezionare diverse skin per il personaggio utilizzando le monete raccolte durante il gioco, quindi il negozio deve poter comunicare con la classe Skin.

Soluzione Ho implementato un sistema completo di shop utilizzando il pattern Observer per la comunicazione tra View e Controller.

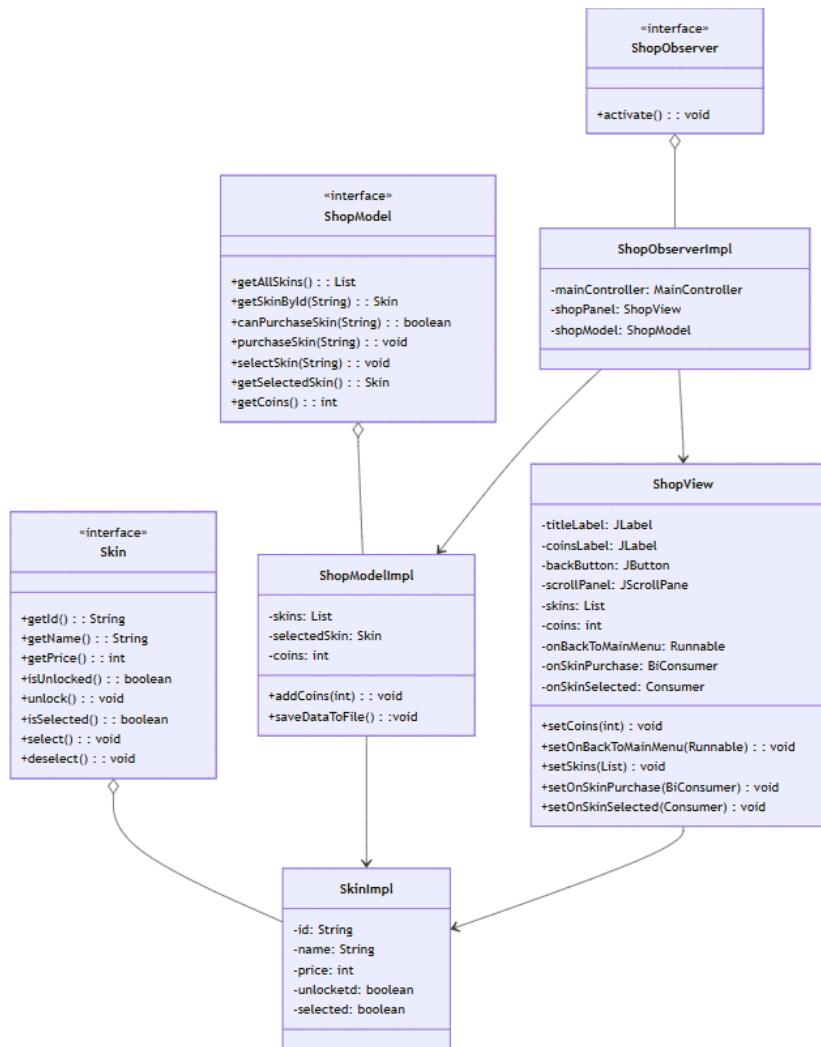


Figura 2.12: Schema UML che rappresenta la struttura dello Shop.

Gestione del Menu Principale

Problema Il gioco necessita di un menù principale che comunichi con negozio, istruzioni e tasto di avvio del gioco

Soluzione Appoggiandosi al GameFrame, il menù dispone di tre bottoni per poter entrare nella schermata dello shop, per poter leggere le istruzioni o per avviare una partita. All'interno della partita si può accedere al menù di Pausa tramite il quale continuare o tornare al menù principale. Alla morte del Player compare la schermata di Game Over dove vediamo il punteggio e le monete raccolte, e da cui possiamo tornare al menu principale.

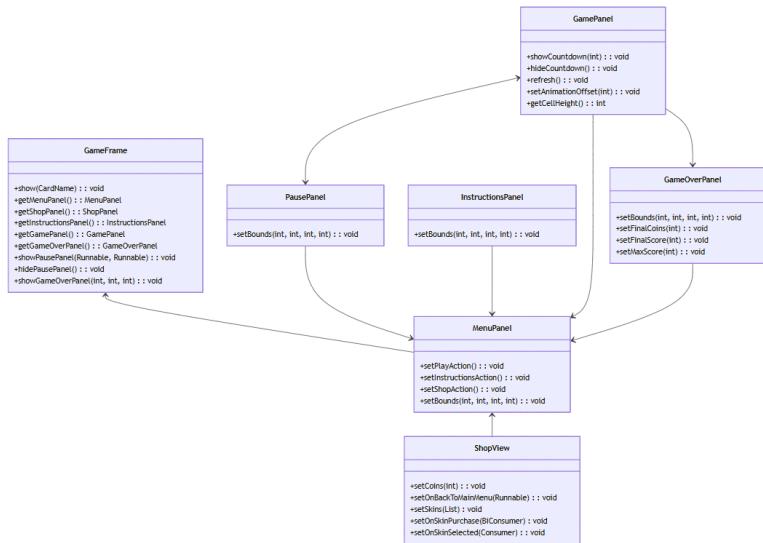


Figura 2.13: Schema UML che rappresenta la struttura del Menu.

Capitolo 3

Sviluppo

3.1 Testing Automatizzato

Per la realizzazione dei seguenti test abbiamo utilizzato JUnit 5. Di seguito le classi di test implementate:

- GameMapTest: verifica il funzionamento del model della mappa.
- ChunkTest: controlla la corretta inizializzazione, l'aggiunta di oggetti e i calcoli di visibilità.
- ChunkFactoryTest: testa la creazione dei diversi chunks.
- CollectibleTest: verifica il corretto posizionamento e raccoglimento dei Collectible.
- ObstacleTest: testa l'inizializzazione e il posizionamento degli ostacoli.
- CellTest: controlla gli inserimenti e le rimozioni di oggetti nelle celle e la creazione.
- ObjectPlacerTest: verifica l'esistenza di un percorso sempre attraversabile nei chunk di erba e il corretto posizionamento degli ostacoli fissi e dei collezionabili.
- CollisionDetectorTest : testa che la classe riconosca correttamente l'avvenuta collisione tra due GameObject
- CollisionIdentifierTest : testa il giusto riconoscimento del tipo di collisione e che se assata una lista di oggetti non coerenti lanci una IllegalArgumentException

- CollisionHandlerTest : testa che nel caso di una avvenuta collisione tra il player e un GameObject la classe la gestisca correttamente
- MovementValidator : testa che la classe sia in grado di riconoscere in modo corretto se sia possibile spostarsi nella Cell fornita in input
- PlayerTest : testa che tutte le funzionalità del player funzionino correttamente

3.2 Note di Sviluppo

3.2.1 Lorenzo Antonioli

Utilizzo di Stream e lambda expressions

Utilizzati in vari punti. Un esempio è: <https://github.com/lolloantonioli/OOP24-road-hop/blob/d3431eb005bdb207df5c242df718ba3b18a0f73d/src/main/java/it/unibo/model/Map/impl/GameMapImpl.java#L77-L80>

Utilizzo di ImmutableList e ImmutableSet della libreria Google Guava

Utilizzata per vera immutabilità ed efficienza. Un esempio è: <https://github.com/lolloantonioli/OOP24-road-hop/blob/d3431eb005bdb207df5c242df718ba3b18a0f73d/src/main/java/it/unibo/model/Map/impl/GameMapImpl.java#L109> e <https://github.com/lolloantonioli/OOP24-road-hop/blob/a7d0e64d3a0e2ec2ff7d3f6cf11f8df/src/main/java/it/unibo/model/Map/impl/CellImpl.java#L60>

Utilizzo di Preconditions della libreria Google Guava

Utilizzata per maggiore chiarezza e facilità d'uso in vari punti del progetto. Un esempio è: <https://github.com/lolloantonioli/OOP24-road-hop/blob/fceb9567cd724384d9e32318b78278d5d61885ea/src/main/java/it/unibo/model/Map/impl/ChunkImpl.java#L40-L45>

3.2.2 Eleonora Bianco

Utilizzo di Stream e lambda expressions

Utilizzato in vari punti, questo è un esempio: Un esempio è: <https://github.com/lolloantonioli/OOP24-road-hop/blob/main/src/main/java/it/unibo/model/Player/impl/MovementValidatorImpl.java#L23-L29>

Utilizzo di Preconditions della libreria Google Guava

Utilizzato in più metodi, questo è un esempio: Un esempio è: <https://github.com/lolloantonioli/OOP24-road-hop/blob/main/src/main/java/it/unibo/controller/Player/impl/PlayerControllerImpl.java#L52-L53>

3.2.3 Giulia Fares

Utilizzo di Stream e lambda expressions

Utilizzate per semplificare la manipolazione delle collezioni di ostacoli, sostituendo loop esplicativi con operazioni funzionali. Un esempio è: <https://github.com/lolloantonioli/OOP24-road-hop/blob/main/src/main/java/it/unibo/model/Obstacles/impl/MovingObstacleManagerImpl.java#L44-L46>

Utilizzo di Switch Expressions

Per mappare tipi di ostacoli in costanti in modo più conciso e leggibile. Un esempio è: <https://github.com/lolloantonioli/OOP24-road-hop/blob/main/src/main/java/it/unibo/model/Obstacles/impl/MovingObstacles.java#L70-L73>

Uso di computeIfAbsent per il caching di configurazioni di chunk

Per evitare inizializzazioni ridondanti. Un esempio è: <https://github.com/lolloantonioli/OOP24-road-hop/blob/main/src/main/java/it/unibo/controller/Obstacles/impl/MovingObstacleControllerImpl.java#L102-L103>.

3.2.4 Luca Varale Rolla

Utilizzo di Properties per la persistenza dei dati

Implementato un sistema di salvataggio robusto utilizzando la classe Properties di Java per gestire il salvataggio e caricamento dello stato del shop. Un esempio è: <src/main/java/it/unibo/model/shop/impl/ShopDataManagerImpl.java#L38-L62>.

Utilizzo di Stream e lambda expressions

Utilizzate per semplificare la gestione delle skin e la ricerca nel modello shop. Un esempio è: <src/main/java/it/unibo/model/Shop/impl/shopModelImpl.java#L54> per la selezione della skin corrente, e <src/main/java/it/unibo/>

`model/Shop/impl/shopModelImpl.java#L60` per il recupero delle skin per ID.

Gestione delle eccezioni con fallback ai valori di default

Implementata una strategia di gestione robusta degli errori con fallback automatico ai dati di default in caso di problemi nel caricamento del file di salvataggio. Un esempio è: `src/main/java/it/unibo/model/shop/impl/ShopDataManagerImpl.java#L113-L121`.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Lorenzo Antonioli

Durante lo sviluppo di questo progetto sento di aver avuto un ruolo centrale all'interno del gruppo, sia in termini organizzativi che decisionali. Mi sono impegnato attivamente nel coordinare le attività, nel indirizzare i compagni nelle scelte tecniche e nell'assicurarmi che il lavoro procedesse in modo coerente rispetto agli obiettivi prefissati. Il progetto mi ha messo alla prova sotto diversi aspetti, sia tecnici che gestionali. Alcune fasi sono state particolarmente complesse, ma grazie all'impegno e a un lavoro costante sono riuscito a dare un contributo concreto e a portare avanti l'intero sviluppo in modo efficace, arrivando a un risultato che reputo discreto. Occupandomi della mappa del gioco sono riuscito a sperimentare anche gli aspetti grafici di un applicativo anche se l'interfaccia utente risulta essenziale e potrebbe essere migliorata.

4.1.2 Eleonora Bianco

Nello sviluppo di questo progetto riconosco di aver gestito male i tempi, rimandando l'inizio del lavoro. Nonostante ciò, credo di aver dato un contributo significativo al gruppo, in particolare nel mantenere coerenza nell'approccio adottato per la gestione del gioco. Il progetto ha rappresentato una sfida su diversi fronti, soprattutto per quanto riguarda l'adattamento delle mie idee al lavoro già svolto dai miei compagni. Ritengo comunque di aver raggiunto un buon risultato complessivo, anche se dal punto di vista visivo ci sono ancora margini di miglioramento.

4.1.3 Giulia Fares

Durante lo sviluppo di questo progetto ho avuto alcuni problemi a gestire i tempi, a capire come iniziare, quando iniziare ma anche cosa serviva fare e come dovevo farlo; soprattutto essendo il primo progetto di una certa complessità posso dire di aver imparato molto. Alla fine sono riuscita a organizzarmi anche se non al meglio, ma su questo potrò sicuramente lavorare in futuro. Da questo progetto ho imparato anche che per lavorare in gruppo devo adattare il mio lavoro a quello degli altri, è stato difficile all'inizio accettare di dover cambiare qualcosa che io ritenevo concluso perchè, per esempio, nell'insieme non funzionava bene. Un aspetto tecnico, su cui ho lavorato io, che non mi soddisfa pienamente è la parte grafica degli ostacoli mobili che poteva sicuramente venire meglio, essere più fluida e meno "meccanica". Sono comunque abbastanza soddisfatta di come è venuto il lavoro complessivo alla fine, anche se organizzandoci meglio sicuramente avremmo potuto fare qualcosa in più, siamo tutti riusciti ad adattarci e fare del nostro meglio.

4.1.4 Luca Varale Rolla

All'inizio del progetto ho avuto difficoltà a organizzarmi e a capire esattamente "cosa" e "come" fare, e nel cercare di lavorare in modo coordinato coi compagni. Mi sono inizialmente affidato ai miei compagni, in particolare a lorenzo, poi col tempo sono migliorato e ho aumentato la mia capacità di gestione dei problemi e autonomio. Nel complesso sono soddisfatto del risultato finale: superate le difficoltà iniziali, ho affinato le mie competenze tecniche e organizzative, consegnando un prodotto coerente con gli obiettivi del gruppo. Nella lavorazione allo Shop ho potuto gestire in prima persona alcune componenti grafiche e di fluidità ad esempio nell'ingrandimento della finestra e alla fine posso ritenermi abbastanza contento.

Appendice A

Guida utente

A.1 Menù Iniziale

Quando si apre l'applicazione appare il menù principale. Sono presenti 3 pulsanti:

- Play: avvia una partita.
- Instructions: apre il menù delle istruzioni.
- Shop: rimanda allo shop dove è possibile acquistare le skin.

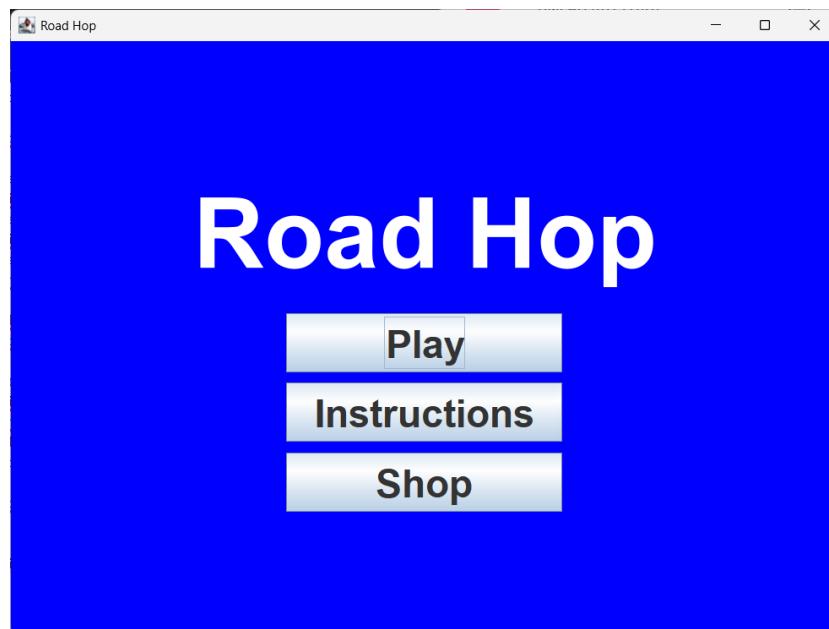


Figura A.1: Schermata iniziale del videogioco

A.2 Shop

Nello shop sono presenti diverse skin da acquistare, già acquistate ma non selezionate e selezionate. In alto a destra ci sono il numero di monete a disposizione dell'utente. In basso a sinistra, cliccando "back", si torna al menù principale.

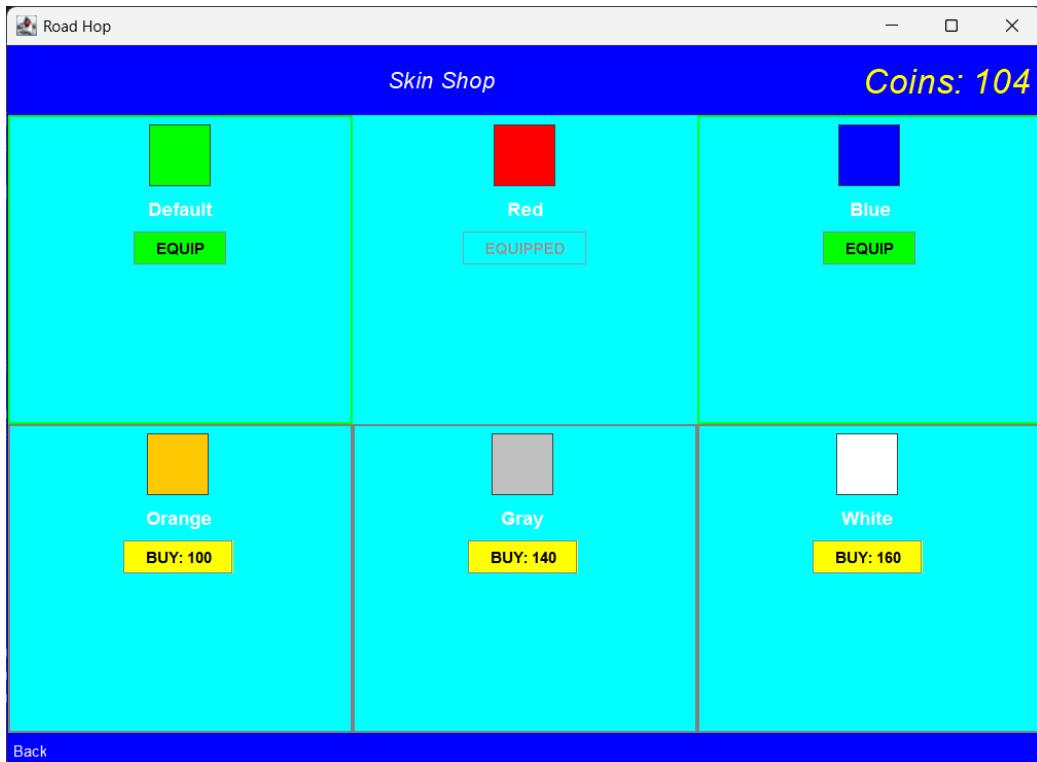


Figura A.2: Shop

A.3 Play

Cliccando il tasto play apparirà un countdown che, arrivato a 0, farà partire lo scorrimento della mappa e il personaggio (cerchio grande colorato in base alla skin) potrà muoversi. Per muoversi, l'utente potrà utilizzare:

- W, freccia su: avanti
- A, freccia sinistra: sinistra
- S, freccia giù: indietro

- D, freccia dx: destra

Inoltre, il giocatore può mettere in pausa il gioco cliccando il tasto P. I chunk della mappa sono rappresentati con:

- Verde: chunk di tipo grass
- Blu: chunk di tipo river
- Nero: chunk di tipo road
- Ferrovia: chunk di tipo railway

Gli oggetti sono:

- Alberi: quadrati neri nei grass chunk
- Tronchi: rettangoli marroni nei river chunk
- Auto: rettangoli rossi nei road chunk
- Treni: rettangoli lunghi grigi nei railway chunk
- Monete: cerchi gialli
- Seconda Vita: cerchi magenta

Nel caso in cui il player collezionasse una seconda vita, apparirà in alto a sinistra un cerchio magenta, che sparirà quando verrà utilizzata. In alto a destra è presente un numero bianco che indica il numero di passi in avanti effettuati e un numero giallo che indica il numero di monete raccolte.

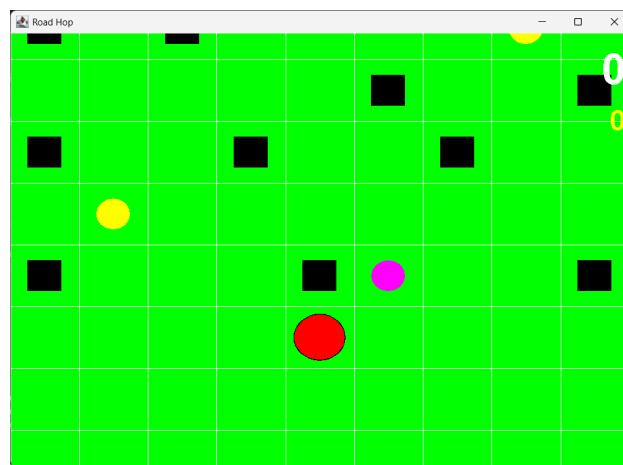


Figura A.3: Interfaccia di gioco

A.4 Gameover

Una volta che il giocatore è morto, apparirà una schermata di game over con il numero di passi effettuati, il numero di monete raccolte e il record delle partite precedenti.

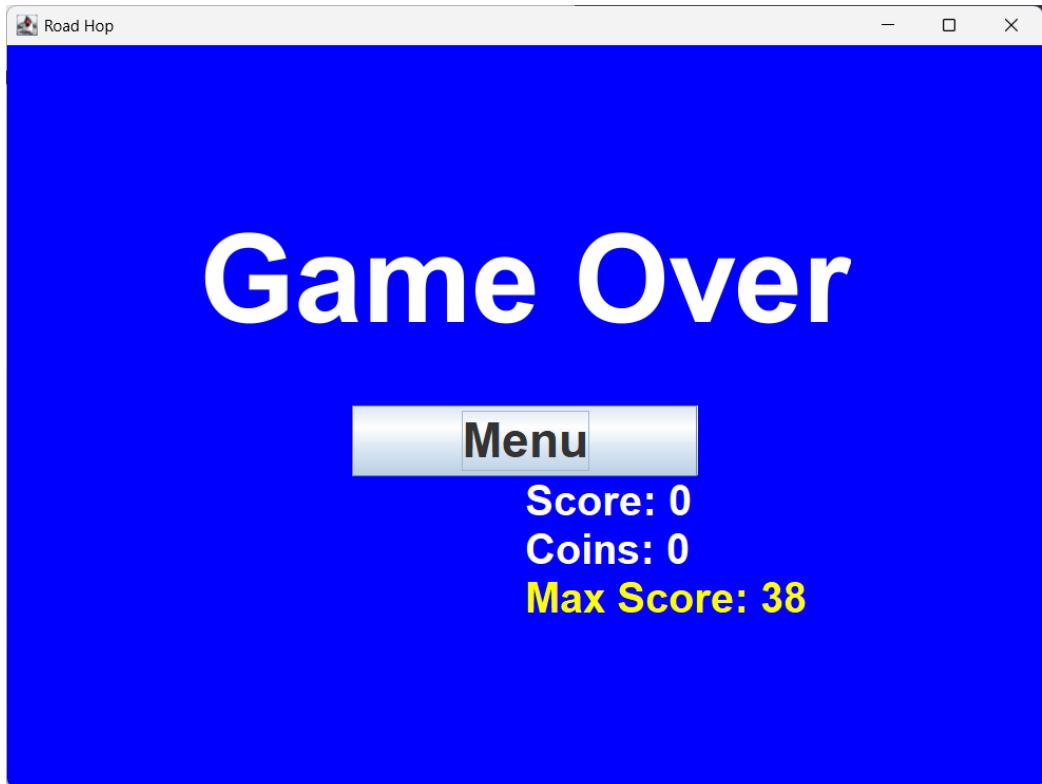


Figura A.4: Gameover

Appendice B

Esercitazioni di Laboratorio

B.0.1 lorenzo.antonioli2@studio.unibo.it

- Laboratorio 07: <https://virtuale.unibo.it/mod/forum/discuss.php?d=177162#p245948>
- Laboratorio 08: <https://virtuale.unibo.it/mod/forum/discuss.php?d=178723#p247377>
- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=179154#p248300>
- Laboratorio 10: <https://virtuale.unibo.it/mod/forum/discuss.php?d=180101#p249465>
- Laboratorio 11: <https://virtuale.unibo.it/mod/forum/discuss.php?d=181206#p250643>