Relazione crossy-road
Corso di “Programmazione ad Oggetti”

Giuliano Manzi, Alessio Magnani, Mattia Golinucci, Lorenzo Baldazzi

15 febbraio 2026

Indice

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Crossy Road è un gioco single player endless runner. Un endless runner è un gioco sprovvisto di livelli che termina solo quando il giocatore perde la partita. L'obiettivo del giocatore è quello di avanzare lungo la mappa con il suo personaggio il più possibile senza venire travolto dai vari ostacoli presenti. La mappa è composta da varie sezioni, ognuna con ostacoli differenti. Gli ostacoli sono rappresentati da automobili e treni che attraversano in maniera randomica le rispettive sezioni della mappa. Le sezioni contengono ostacoli passivi che il giocatore deve evitare per proseguire. Disseminate lungo la mappa il giocatore può trovare delle monete che possono essere raccolte e usate per sbloccare nuovi personaggi. Oltre alle monete il giocatore può raccogliere dei power-up con effetti temporanei.

Requisiti funzionali

- Le sezioni della mappa dovranno essere generate in maniera randomica a pari passo con il movimento del personaggio.
- Gli ostacoli andranno generati con densità e velocità randomiche, l'ultima crescente lungo il cammino del personaggio.
- Dovranno essere gestite le collisioni tra il personaggio e gli ostacoli.
- I power-up dovranno modificare lo stato di gioco e avere una durata temporanea.
- Il gioco dovrà avere un menù di avvio per permettere all'utente di iniziare una nuova partita o cambiare personaggio.

- Il gioco dovrà prevedere un sistema di salvataggio per poter recuperare i personaggi sbloccati in precedenza.

Requisiti non funzionali

- L'utilizzo delle risorse deve essere contenuto durante l'esecuzione del gioco.
- Durante l'esecuzione del gioco non devono esserci cali di frame rate.

1.2 Modello del Dominio

La mappa è divisa in sezioni tematiche, ognuna delle quali contiene ostacoli diversi e oggetti raccoglibili. Gli ostacoli come automobili e treni sono fatali e in continuo movimento, mentre ostacoli come rocce e alberi sono innocui e inamovibili. Caso particolare è l'ostacolo acqua che deve essere attraversato usando dei tronchi, altrimenti risulta fatale. Gli oggetti raccoglibili possono essere monete oppure power-up che possono modificare i parametri di gioco, quali velocità di auto, moltiplicatore di monete e invincibilità del giocatore. Il giocatore controlla un personaggio che si muove attraverso le sezioni. Una componente responsabile della partita mantiene le sezioni della mappa in continua generazione casuale e gestisce eventuali collisioni del personaggio con ostacoli e oggetti. Tutto ciò che si trova nella mappa è un oggetto che occupa una posizione nello spazio. Un sistema di salvataggio dei progressi si occupa di salvare informazioni persistenti, che non sono relative alla partite in corso, come le monete raccolte, le skin disponibili, quelle in possesso e la skin attiva. Gli elementi che costituiscono il dominio sono rappresentati in Figura 1.1.

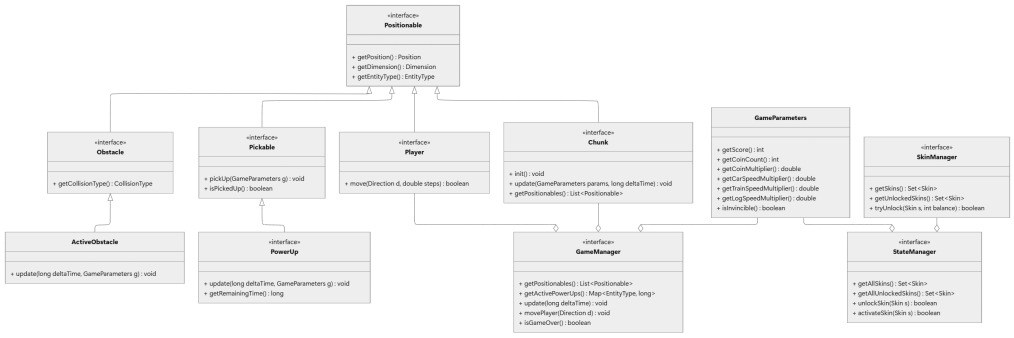

Figura 1.1: Schema UML dell'analisi del dominio, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

Crossy Road adotta il pattern architetturale MVC. Il model è accessibile tramite le interfacce `GameManager` e `StateManager`. `GameManager` si occupa di gestire la logica della partita in corso, aggiornando lo stato delle varie entità, mentre `StateManager` gestisce lo stato globale e persistente dell'applicazione, come il menu e il negozio delle skin. Ciascun entry point del model è associato ad almeno un controller dedicato. In particolare, `GameManager` è gestito dal `GameController`, che si occupa di avviare e fermare il game loop, aggiornare il model e comunicare con la view per la visualizzazione delle posizioni aggiornate delle entità di gioco. Il `MenuController` interagisce con lo `StateController` per caricare e salvare dati persistenti del gioco, mentre lo `ShopController` comunica con lo `StateController` per ricevere informazioni sulle `Skin`, monete e altro riguardante il negozio. È infine presente un controller di livello superiore: `AppController`, il quale coordina l'interazione tra `GameController`, `MenuController` e `ShopController` e gestisce la navigazione tra le diverse sezioni dell'applicazione. Le view dell'applicazione sono tre, ciascuna con un compito specifico. La `MenuView` consente all'utente di navigare tra le diverse opzioni dell'applicazione. La `ShopView` si occupa della visualizzazione delle skin disponibili e di quelle già sbloccate. La `GameView` renderizza le entità di gioco e mostra le informazioni rilevanti per la partita, come le monete raccolte e i power up in corso. Tutte le view si limitano alla rappresentazione grafica e alla raccolta dell'input, delegando ogni logica al controller associato. Con questa architettura è possibile sostituire le view attuali in qualsiasi momento, poiché i controller non mantengono riferimenti a implementazioni concrete e le view non sono vincolate a classi specifiche del model. Allo stesso modo, nessuna entità del model ha riferimenti verso

controller o view, rendendo anche il model facilmente riutilizzabile.
In Figura 2.1 è mostrato il diagramma UML architetturale.

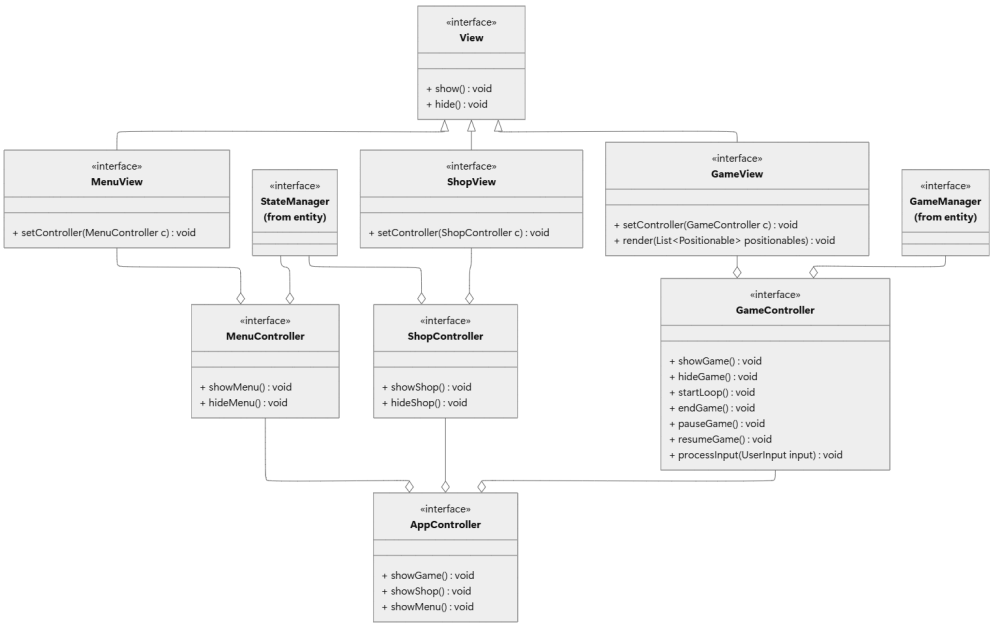

Figura 2.1: Schema UML architetturale di Crossy Road.

2.2 Design dettagliato

2.2.1 Giuliano Manzi

Gestione comportamento terreni

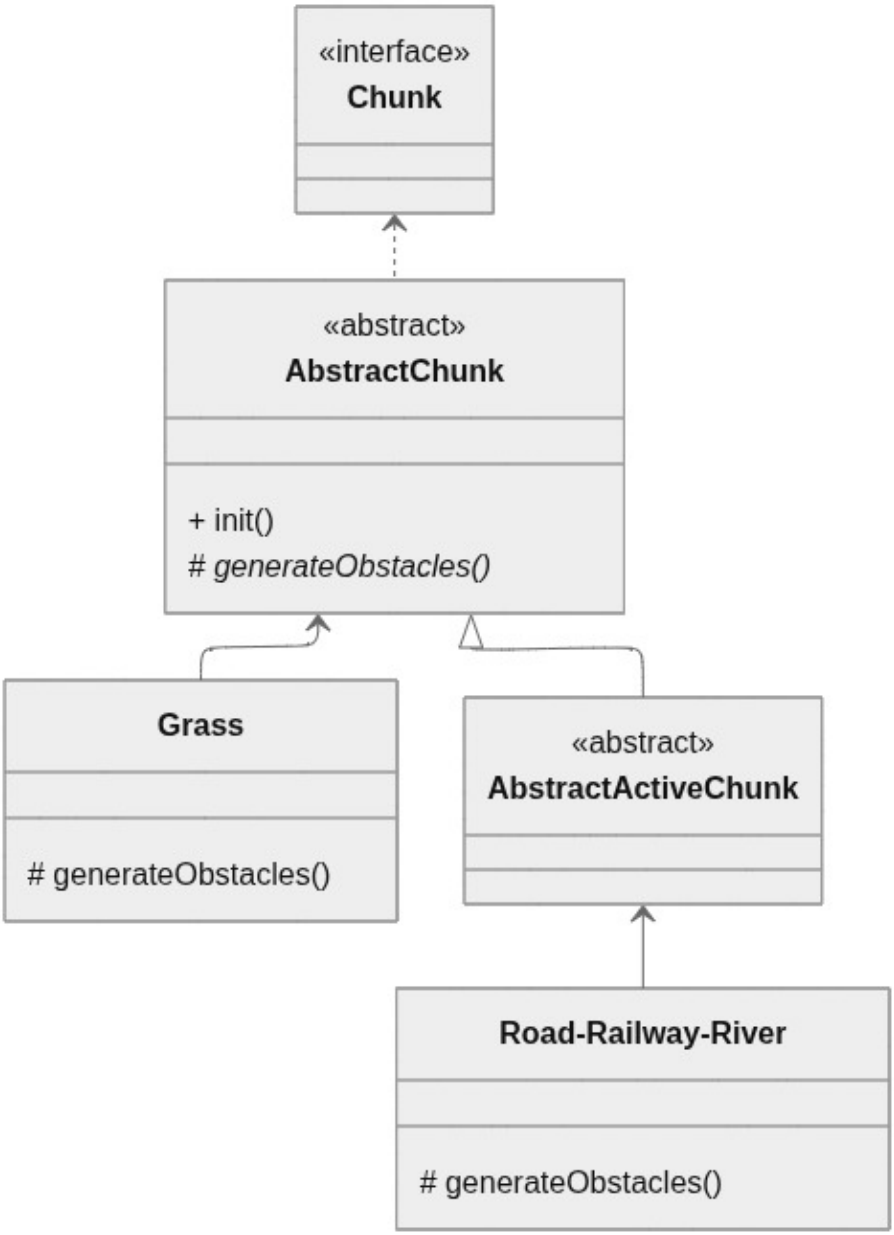

Figura 2.2: Rappresentazione UML dell'applicazione dei pattern Template Method e Strategy alle diverse tipologie di terreno

Problema Ogni `Chunk` genera ostacoli diversi, ma ha lo stesso setup iniziale e lo stesso metodo di aggiornamento. Inoltre il `GameManager` deve poter gestire i terreni indipendentemente dalla loro implementazione e permettere facilmente l'aggiunta di nuovi terreni.

Soluzione Si è scelto quindi di applicare i pattern *Template Method* e *Strategy* come mostrato in Figura 2.2. `Chunk` definisce le strategie comuni a tutti i terreni, `AbstractChunk` e `AbstractActiveChunk` forniscono un primo livello di specializzazione per i metodi comuni oltre al template method `init()`. Questo metodo fornisce una struttura fissa che delega alle classi `Grass`, `Road`, `Railway` e `River` l'implementazione del metodo astratto `generateObstacles()`. Il `GameManager` lavorerà con i `Chunk` senza preoccuparsi dell'implementazione specifica.

Avanzamento temporale degli elementi

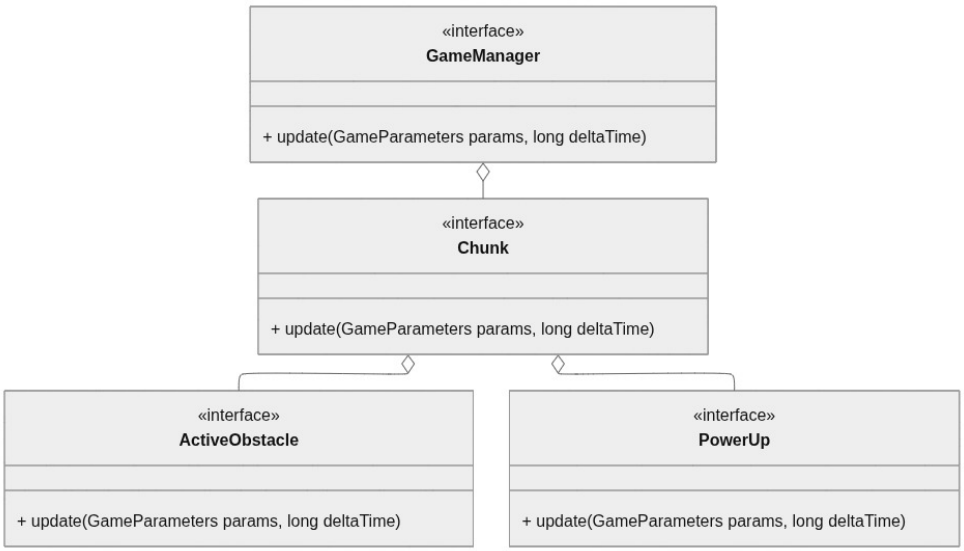

Figura 2.3: Rappresentazione UML dell'applicazione del pattern Composite per l'avanzamento temporale

Problema La mappa di gioco contiene diverse sezioni, ognuna delle quali contiene diversi ostacoli attivi e power-up. Ognuno di questi elementi deve avanzare nel tempo, spostandosi nel caso di un ostacolo, calcolando il tempo di vita rimanente nel caso di un power-up o eliminando elementi non

più necessari nel caso di una sezione. La mappa dovrebbe quindi essere a conoscenza di tutti gli oggetti presenti su di essa e aggiornarli uno per uno.

Soluzione Si è scelto di adottare il principio di *Composite Pattern*. Come si può vedere in fig. 2.3, i vari elementi seguono una gerarchia, ogni elemento espone un metodo `update(...)` con la quale aggiorna se stesso e delega l'aggiornamento agli elementi da esso contenuti. Questa implementazione non prevede un'interfaccia comune, come è standard nel *Composite Pattern*, ma la condivisione del metodo `update(...)` e la propagazione del comportamento permettono comunque di sfruttarne i benefici.

Game Loop per la gestione del gioco

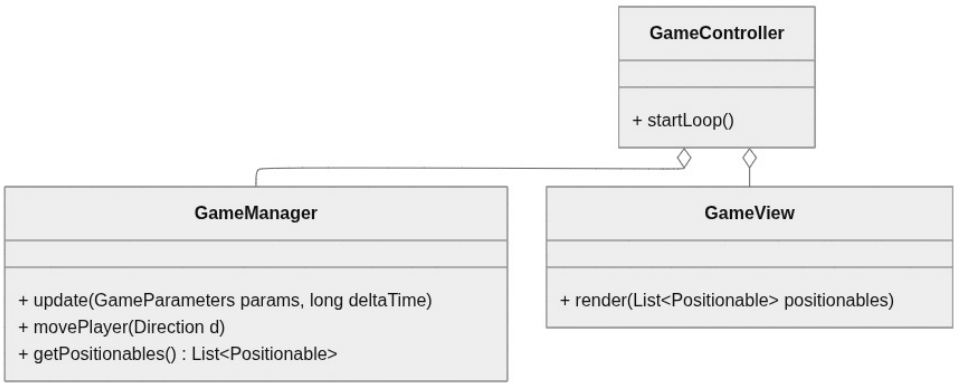

Figura 2.4: Rappresentazione UML dell'applicazione del pattern Game Loop per la sincronizzazione del gioco

Problema È necessario sincronizzare correttamente logica di gioco e rendering, per garantire uno stato visivo coerente con quello logico. Inoltre il gioco deve avanzare nel tempo in maniera continua. Tutto ciò richiede un controllo centralizzato del flusso di esecuzione.

Soluzione La soluzione adottata prevede l'utilizzo del *Game Loop Pattern* che fornisce una struttura centrale per la gestione del gioco. Il *Game Loop* consente di far avanzare il gioco nel tempo, aggiornandone lo stato in base al tempo trascorso e renderizzandolo. In accordo con il *pattern MVC* il `GameController` separa le responsabilità di Model (`GameManager`) e View

(`GameView`) e agisce come intermediario. Per farlo il `GameController` si compone del `GameManager` e della `GameView`, che non comunicano mai direttamente tra loro (fig. 2.4). Ad ogni iterazione del *Game Loop* il `GameController` aggiorna lo stato del gioco tramite il metodo `update(...)` del `GameManager`, poi aggiorna la `GameView` tramite il metodo `render(...)` di quest'ultima.

2.2.2 Alessio Magnani

Posizionamento delle entità nella mappa

Problema Tutte le entità presenti nel gioco, come il `Player`, gli ostacoli, gli oggetti raccoglibili e i `Chunk`, devono avere una posizione e una dimensione nello spazio 2D. Tutte le entità devono fornire metodi per accedere alla propria posizione e dimensione, senza permettere modifiche dall'esterno.

Soluzione Dato che tutte le entità presenti nella mappa hanno una posizione e una dimensione, è stato scelto di creare un'interfaccia `Positionable`, che rende possibile la lettura di questi ultimi. Poiché tutte le entità devono implementare questa interfaccia, onde evitare ripetizioni di codice, è stata creata una classe astratta che implementa questi metodi comuni. Internamente le sottoclassi di `AbstractPositionable` possono impostare liberamente la propria posizione, ma sono responsabili di esporre all'esterno solo le operazioni di movimento consentite. L'unica operazione di movimento che deve essere esposta pubblicamente, è lo spostamento verso il basso, in quanto tutte le entità devono traslare in quella direzione man mano che il `Player` avanza. Un esempio di entità che estende da `AbstractPositionable` è `PositionablePlayer`, la cui implementazione è mostrata in Figura 2.5.

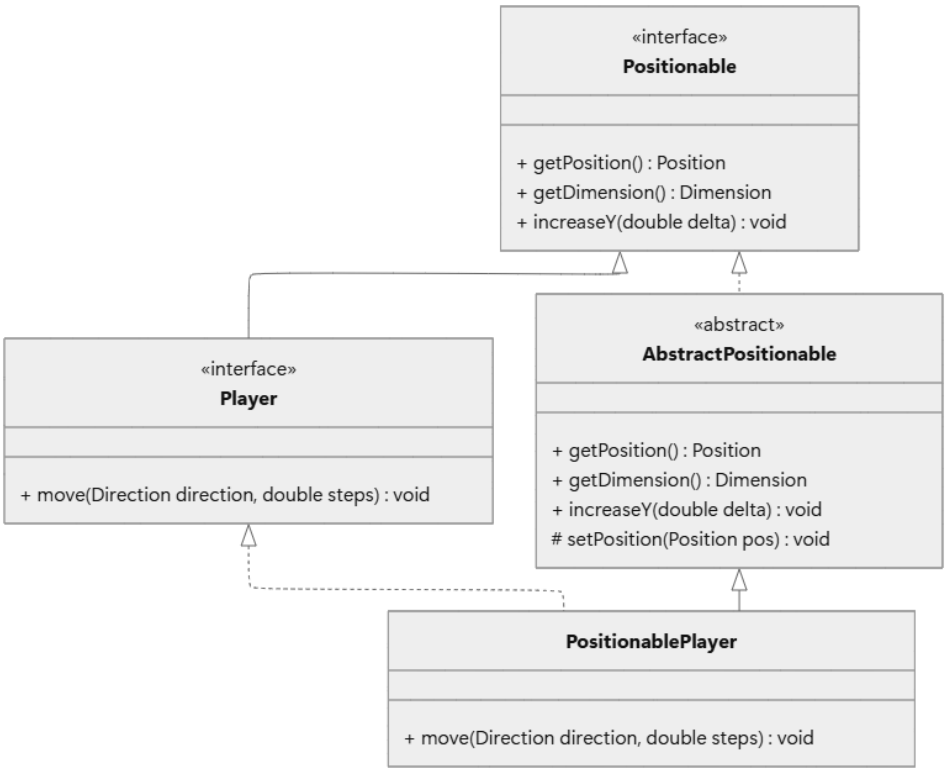

Figura 2.5: Rappresentazione UML della soluzione descritta, in cui sono riportati esclusivamente i metodi pertinenti.

Creazione di `Dimension`

Problema Tutte le entità presenti nella mappa, devono avere una dimensione. È necessario quindi racchiudere in un'unica classe immutabile la larghezza e l'altezza di ogni entità.

Soluzione È stato creato un record `Dimension` che raggruppa larghezza e altezza in un'unica struttura immutabile. All'interno del record è stato applicato lo **Static Factory pattern**, che permette la creazione di dimensioni comuni e ricorrenti, come per esempio un quadrato o una singola unità (1x1). L'implementazione del record `Dimension` è mostrata in Figura 2.6.

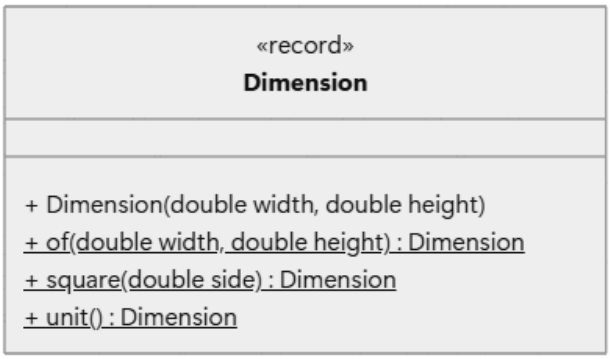

Figura 2.6: Rappresentazione del record `Dimension`.

Generazione di nuovi ostacoli nei `Chunk` attivi

Problema I `Chunk` attivi, che sono una specializzazione dei `Chunk`, si compongono di ostacoli movibili. Il compito dei `Chunk` attivi, oltre a quello di spostare ostacoli ed eliminare quelli non più visibili, è anche quello di **generare nuovi ostacoli** quando necessario. Ogni `Chunk` attivo ha il suo criterio per decidere se è il momento di generare nuovi ostacoli e può decidere cosa e dove generarlo.

Soluzione Per risolvere questo problema, nella classe astratta `AbstractActiveChunk`, il metodo `update(...)`, che si occupa di aggiornare le posizioni degli ostacoli presenti e rimuovere quelli non più visibili, è stato usato come **Template Method** per facilitare la generazione di nuovi ostacoli. Nel metodo `update(...)` viene richiamato un metodo astratto responsabile di determinare se sia necessario generare nuovi ostacoli. Questo metodo sarà implementato successivamente dalle sottoclassi di `AbstractActiveChunk`, in modo che ogni `Chunk` possa avere una **logica differente** di generazione degli ostacoli. In fig. 2.7 è mostrata l'applicazione del Template Method pattern a vari `Chunk` attivi.

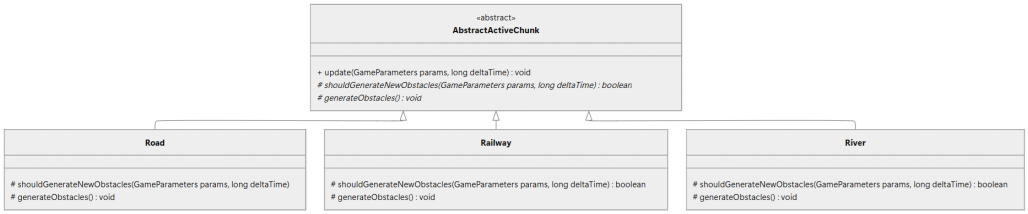

Figura 2.7: Diagramma UML che mostra l'applicazione del Template Method pattern dalle classi `River`, `Road` e `Railway`

Creazione dei Controller

Problema L'`AppController` si compone di tre sotto-controller: `GameController`, `MenuController` e `ShopController`. Ciascun sotto-controller deve avere il riferimento all'`AppController` e l'`AppController` deve avere il riferimento a tutti i sotto-controller.

Soluzione Non è possibile passare direttamente al costruttore di `AppController` le istanze dei tre sotto-controller, poiché ciascuno di essi richiede un riferimento all'istanza di `AppController`, che deve ancora essere creata, provocando così una **dipendenza circolare**. Per risolvere il problema, è stato adottato il **Factory Method pattern**, dove per ogni sotto-controller, nel costruttore di `AppController`, viene richiesto un factory method utile a creare un'istanza di quel sotto-controller. Per esempio, il factory method richiesto per creare il `GameController`, dovrà implementare l'interfaccia creatrice `Function<AppController, GameController>`. Il funzionamento è analogo per gli altri parametri del costruttore. In Figura 2.8 è mostrata la relazione tra i controller, evidenziando i loro costruttori.

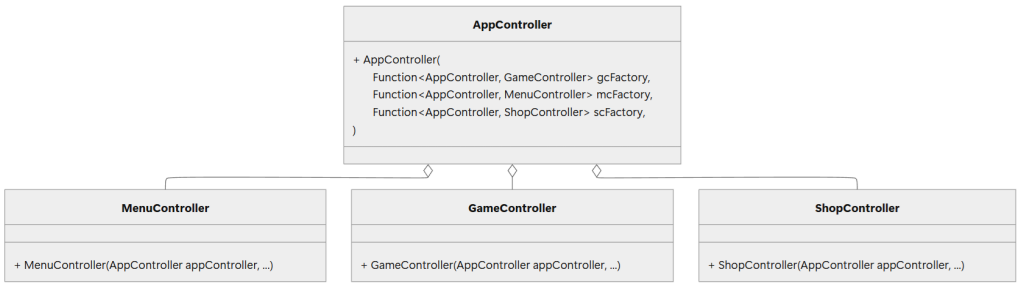

Figura 2.8: Diagramma UML di `AppController` e dei sotto-controller, che evidenzia i loro costruttori.

Comunicazione tra Controller

Problema L'applicazione è composta da più controller specializzati, ciascuno responsabile di compiti precisi e di interagire con view diverse. Sebbene i controller abbiano compiti differenti, devono comunque coordinarsi per la navigazione tra schermate, che richiede l'attivazione/disattivazione delle view di controller differenti.

Soluzione Il coordinamento dei controller è gestito dall'`AppController`, che si occupa di comunicare con gli altri controller senza farli interagire direttamente tra loro. In questo contesto, l'`AppController` adotta il **Mediator pattern**, assumendo il ruolo da mediatore tra i sotto-controller. Così facendo ci assicuriamo che non vengano mai mostrate contemporaneamente due view differenti, poiché ogni sotto-controller può mostrare o nascondere esclusivamente la propria view e, nel caso in cui debba visualizzare view gestite da altri controller, deve inoltrare una richiesta all'`AppController`. Quando l'`AppController` riceve una richiesta di visualizzazione, si occupa anche di nascondere tutte le altre view, evitando così qualsiasi possibile conflitto. In Figura 2.9 è mostrata la relazione tra l'`AppController` e gli altri controller.

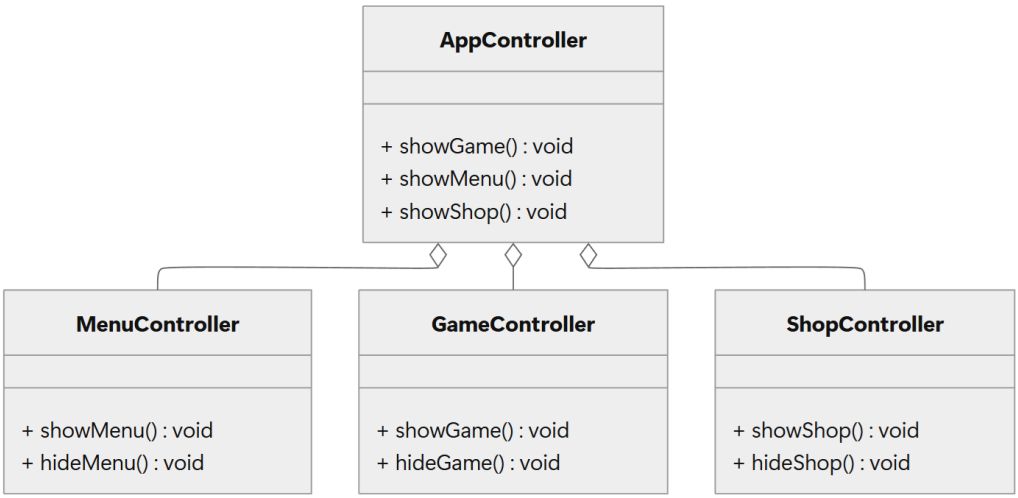

Figura 2.9: Diagramma UML dell'interazione tra `AppController` e gli altri controller.

2.2.3 Golinucci Mattia

Gestione del comportamento degli Ostacoli Attivi

Problema Il gioco presenta due categorie di ostacoli: ostacoli statici (come `Rock` e `Tree`) che rimangono fermi e ostacoli dinamici (come `Car`, `Train` e `WoodLog`) che si muovono nel tempo. Questi ultimi condividono la stessa logica di aggiornamento della posizione basata sul deltaTime e parametri di gioco, ma ciascuno deve utilizzare un moltiplicatore di velocità specifico. Implementare questa logica in ogni classe concreta porterebbe a duplicazione del codice e difficoltà di manutenzione.

Soluzione Si è applicato il *Template Method Pattern* come mostrato in Figura 2.10. La classe astratta `AbstractActiveObstacle` implementa `ActiveObstacle` e definisce il template method `update(...)` che fornisce la struttura fissa dell'algoritmo di aggiornamento. Qui dentro viene richiamato il metodo astratto `getSpeedMultiplier(...)` il quale viene delegato alle classi concrete. `Car` restituisce `gameParameters.getCarSpeedMultiplier()`, `Train` restituisce `gameParameters.getTrainSpeedMultiplier()` e `WoodLog` restituisce gameParameters.getLogSpeedMultiplier(). Questo permette di aggiungere nuovi tipi di ostacoli attivi semplicemente estendendo `AbstractActiveObstacle` e specificando il loro moltiplicatore senza modificare la logica di movimento.

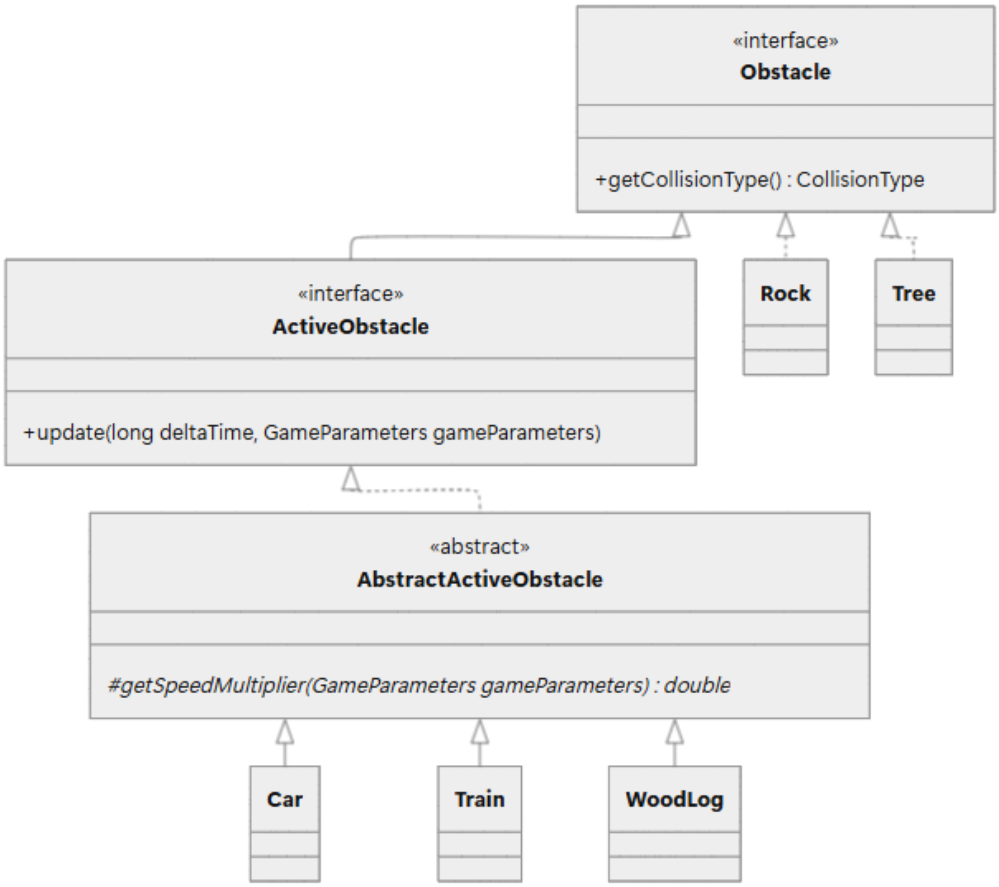

Figura 2.10: Rappresentazione UML della gerarchia degli ostacoli: l'interfaccia `Obstacle` definisce il contratto base, `ActiveObstacle` aggiunge il comportamento di aggiornamento temporale, `AbstractActiveObstacle` implementa il template method, e le classi concrete specializzano il comportamento.

Determinazione del Tipo di Collisione

Problema Ogni ostacolo ha un comportamento di collisione specifico (alcuni sono mortali, altri sono semplicemente bloccanti e alcuni possono essere attraversabili). Il sistema di collisione deve poter determinare l'effetto di una collisione senza conoscere il tipo specifico di ostacolo.

Soluzione Si è applicato lo *Strategy Pattern* attraverso l'enumerazione CollisionType. Ogni ostacolo, sia attivo che passivo, implementa `getCollisionType()` restituendo la propria strategia di collisione (`DEADLY` per `Car` e `Train`, `SOLID`

per `Tree` e `Rock` e `TRANSPORT` per `WoodLog`). Il collision detector può così ottenere il tipo di collisione tramite polimorfismo e applicare l'effetto appropriato (game over, blocco del movimento, trasporto) senza dipendere dalle classi concrete degli ostacoli. Si veda Figura 2.10.

Costruzione flessibile dei Parametri di Gioco

Problema La classe `GameParametersImpl` richiede 7 parametri nel costruttore (`coinMultiplier, carSpeedMultiplier, trainSpeedMultiplier, invincibility, logSpeedMultiplier, coinCount e initScore`). Creare istanze con costruttori di questo genere può diventare complesso e poco leggibile, specialmente quando si vogliono impostare solo alcuni parametri mantenendo i valori di default per gli altri. Inoltre, l'ordine dei parametri può generare errori se non si conosce bene la struttura del costruttore.

Soluzione Si è applicato il *Builder Pattern* attraverso la classe `GameParametersBuilder` come mostrato in Figura 2.11. Il builder permette di costruire un oggetto `GameParametersimpl` specificando anche solo alcuni dei parametri tramite setter. Alla fine di tutti i `set` vi è il `build` il quale restituisce l'istanza finale passando tutti i parametri al costruttore di `GameParametersImpl`. I parametri non specificati mantengono i valori di default del gioco. Questo approccio rende il codice più leggibile:

```
GameParameters params = new GameParametersBuilder()
.setCarSpeedMultiplier(1.5)
.setInvincibility(true)
.setCoinCount(8)
.build();
```

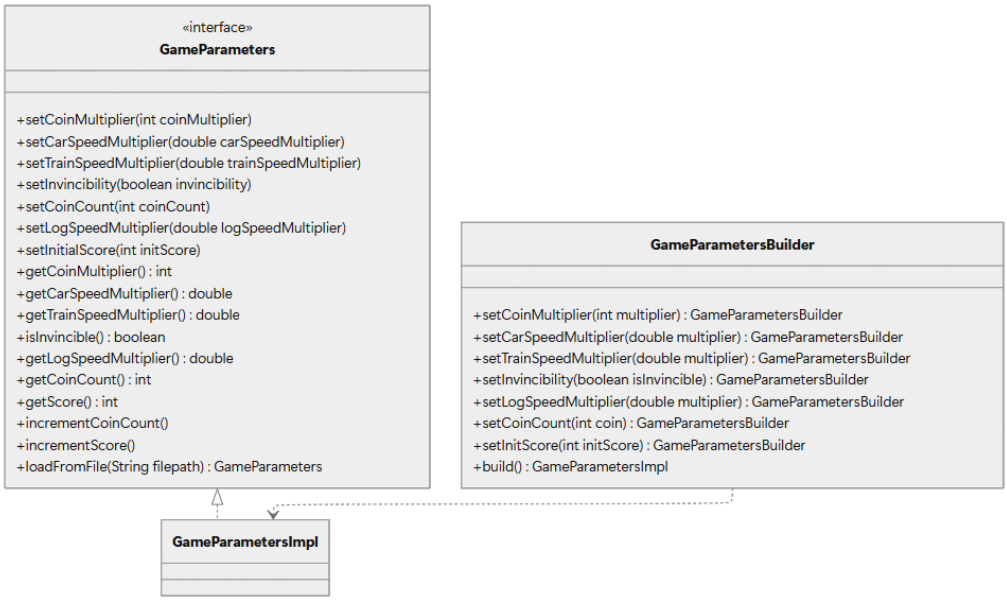

Figura 2.11: Rappresentazione UML di `GameParameters` con applicazione del Builder Pattern per la costruzione flessibile dei parametri di gioco.

2.2.4 Lorenzo Baldazzi

Gestione Effetti `Pickable`

Problema Nella mappa di gioco sono presenti diversi oggetti raccoglibili (`Coin` e `PowerUp`). Ognuno di essi applica effetti diversi sui parametri di gioco (`GameParameters`) quando vengono raccolti.Tutti gli oggetti raccoglibili (`Pickable`) hanno la stessa logica di raccolta, ma l'effetto specifico varia per ciascun tipo.

Soluzione Si è scelto di applicare il **Template Method Pattern** in `AbstractPickable`. Più precisamente nel metodo `pickUp` nel caso in cui l'oggetto `Pickable` non sia ancora stato raccolto, viene richiamato il metodo astratto `applyEffect` che viene implementato dalle singole sottoclassi per definire l'effetto specifico.

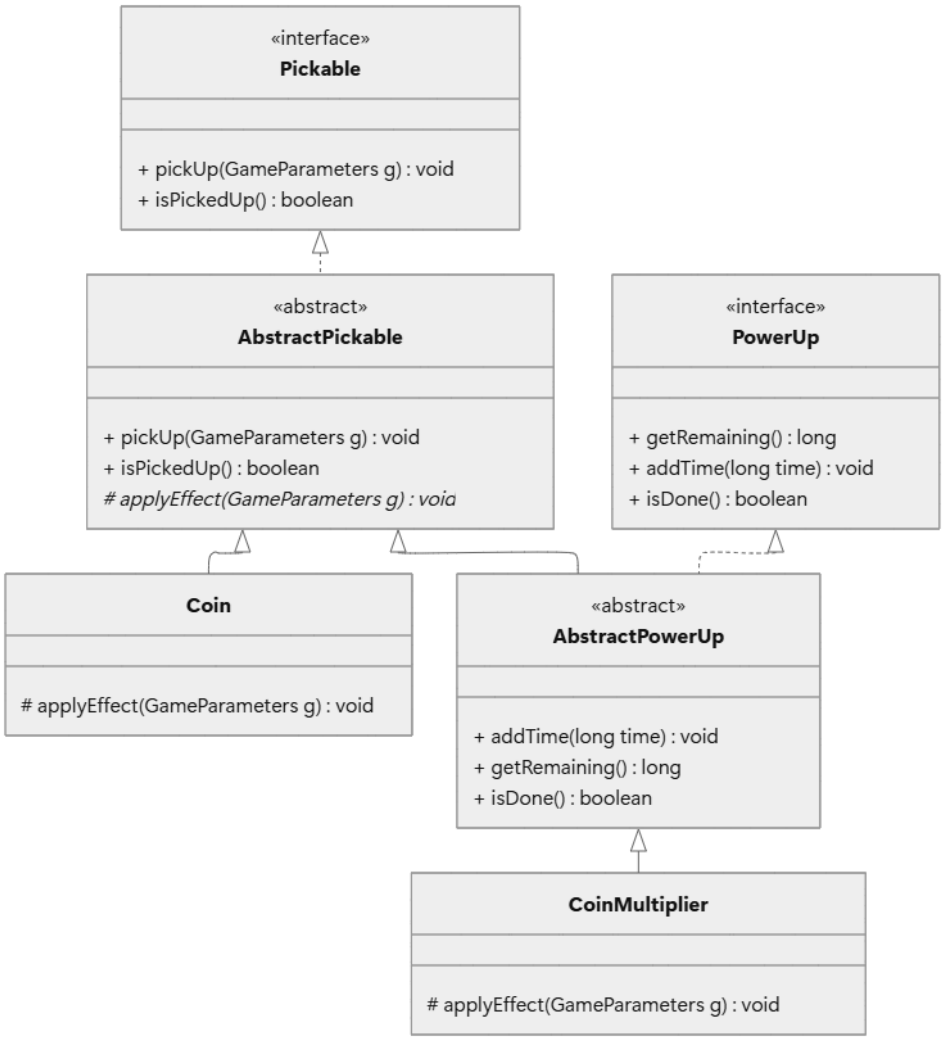

Figura 2.12: Rappresentazione UML di `Pickable` e `CoinMultiplier` come esempio di implementazione del Template Method Pattern per i `PowerUp`.

Gestione Durata `PowerUp`

Problema I vari tipi di `PowerUp` (`CoinMultiplier`, `Invincibility`, `SlowCars`) quando raccolti attivano il proprio effetto e lo mantengono per un certo quantitativo di tempo. La logica generale di aggiornamento temporale è in comune con tutti i `PowerUp` ma ciascun tipo deve attivare e disattivare effetti diversi sui parametri di gioco (`GameParameters`).

Soluzione Si è scelto di applicare il **Template Method Pattern** in `AbstractPowerUp`. In particolare il metodo `update` tiene aggiornato il tempo rimanente fino alla fine dell'effetto e quando scade richiama il metodo astratto `deactivate` che viene implementato dalle singole sottoclassi per ripristinare i parametri di gioco (`GameParameters`) ai valori precedenti all'attivazione.

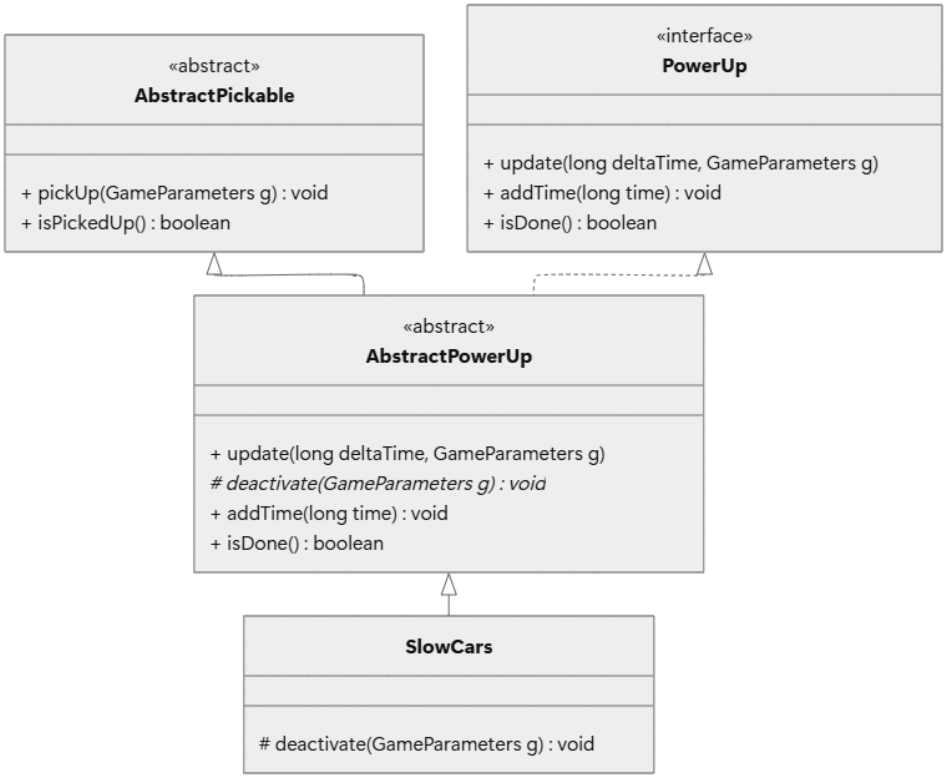

Figura 2.13: Rappresentazione UML del Template Method Pattern applicato ai soli `PowerUp`. `SlowCars` è un esempio di implementazione, analogo è il funzionamento per gli altri.

Costruzione Flessibile delle `Skin`

Problema La classe `SkinImpl` utilizza un costruttore con (`name, id, price, overheadImagePath, frontImagePath`). Con numerosi parametri si rende l'istanziazione di nuovi oggetti meno intuitiva e leggibile considerando anche il fatto che i parametri devono essere passati in un ordine specifico si aumenta il rischio di errore.

Soluzione Si è scelto di applicare il **Builder Pattern** attraverso la inner-class `SkinImpl.Builder`. Il builder consente di creare l'oggetto `Skin` impostando i parametri attraverso i metodi : `name()`, `id()`, `price()`, `overheadImagePath()`, `frontImagePath()` migliorando cosi la leggibilità del codice.

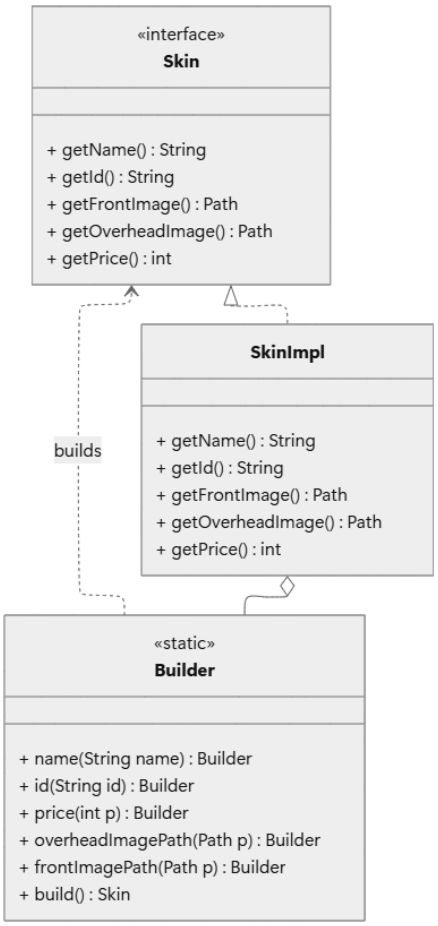

Figura 2.14: Rappresentazione UML del Builder Pattern applicato per la creazione di `Skin`.

Creazione `Skin` Da Sorgenti Diversi

Problema Le `Skin` possono essere create da file (json) o con parametri espliciti è quindi necessario garantire che tutte le skin vengano create in modo consistente indipendetemente dalla sorgente dei dati.

Soluzione Si è scelto di applicare lo **Static Factory Pattern** in `SkinFactory`. Nello specifico, il metodo statico `loadFromJson()` gestisce la validazione del json, mentre l'altro metodo statico `create()` permette la creazione a partire dai parametri forniti.

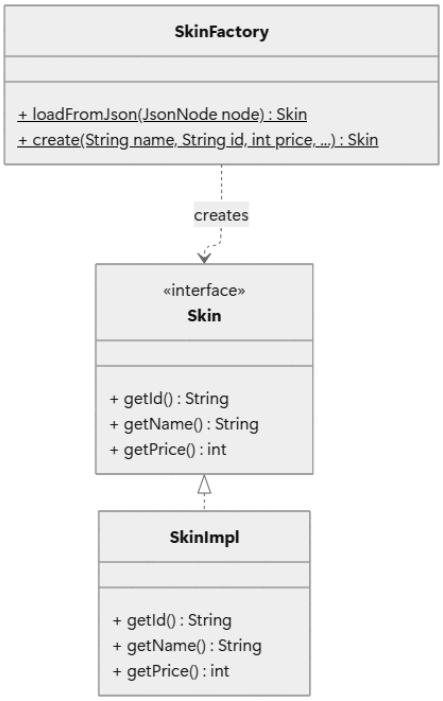

Figura 2.15: Rappresentazione UML del Static Factory Pattern usato per la creazione di `Skin`.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Tutti i test sono stati realizzati con JUnit ed eseguiti tramite il build system Gradle. Per ogni classe presente nel model sono stati creati opportuni test per verificare il corretto funzionamento di esse e per prevenire la regressione durante lo sviluppo. I test sviluppati riguardanti `GameManager` e `Chunk` sono:

- `TestGameManager`: viene testata la corretta inizializzazione del terreno di gioco.
- `TestGrass`: viene testata la generazione randomica di ostacoli passivi e oggetti raccoglibili sul terreno.
- `TestRiver`: viene controllato che il costruttore effettui le validazioni necessarie e che dopo l'inizializzazione, siano presenti gli ostacoli che ci aspettavamo, ovvero `Water` e `WoodLog`.
- `TestRoad`: viene testata la generazione randomica di macchine.
- `TestRailway`: viene testata la randomica generazione di treni.

I test sviluppati riguardanti `Obstacle` e le sue specializzazioni sono:

- `TestPassiveObstacles` viene testata la corretta inizializzazione degli ostacoli passivi.
- `TestWater`: vengono eseguiti controlli base, per verificare che vengano impostati correttamente i parametri passati nel costruttore.
- `TestWoodLog`: viene testato il suo spostamento in varie condizioni, simulando lo scorrimento del tempo e controllando che si trovi nella posizione in cui ci aspettavamo.

- `TestCar`: viene testata la corretta inizializzazione della macchina e l'aggiornamento di posizione di essa.
- `TestTrain`: viene testata la corretta inizializzazione del treno e l'aggiornamento di posizione di esso.

I test sviluppati riguardanti il `Player` sono:

- `TestPositionablePlayer`: viene verificato il corretto spostamento nelle varie direzioni, controllando che il `Player` si trovi nella posizione in cui ci aspettavamo.

I test sviluppati riguardanti il `GameParameters` sono:

- `TestGameParameters`: viene testato la corretta inizializzazione dei parametri di gioco, anche tramite il pattern implementato, e il carimcamento di essi da file.

I test sviluppati riguardanti il `Coin` sono:

- `TestCoin`: viene testata la raccolta delle monete, che la moneta possa essere raccolta una sola volta, che la raccolta incrementi correttamente il contatore delle monete in base al moltiplicatore attivo.

I test sviluppati riguardanti i `PowerUp` sono:

- `AbstractTestPowerUp`: vengono testati i comportamenti comuni dei `PowerUp`. Si testa il raccoglimento dei vari `PowerUp`, che l'effetto specifico venga applicato una sola volta e che tale effetto si disattivi al termine della sua durata.
- `TestCoinMultiplier`: viene testata la corretta applicazione dell'effetto del `CoinMultiplier` e il suo corretto ripristino al termine della sua durata.
- `TestInvincibility`: viene testata la corretta applicazione dell'effetto di `Invincibility` e il suo corretto ripristino al termine della sua durata.
- `TestSlowCars`: viene testata la corretta applicazione dell'effetto di `SlowCars` e il suo corretto ripristino al termine della sua durata.

I test sviluppati riguardanti le `Skin` sono:

- `TestSkin`: viene testata la costruzione della skin tramite `SkinImpl.Builder`.

I test sviluppati riguardanti lo `SkinManager` sono:

- `TestSkinManager`: viene testato il caricamento delle `Skin` dalle risorse, lo sblocco delle `Skin` e la gestione di quelle gia sbloccate.

I test sviluppati riguardanti lo `StateManager` sono:

- `TestStateManager`: viene testata la corretta inizializzazione con la `Skin` di default, il reset del gioco, lo sblocco e attivazione delle `Skin`, il salvataggio e caricamento del file json.

3.2 Note di sviluppo

3.2.1 Giuliano Manzi

Utilizzo di `Range` dalla libreria Google Guava

Permalink: `https://github.com/TheRealGtx/OOP25-crossy-road/blob/5df9c97a5c6fc9cf7d696d8a84d8b6ae0ff6ad8a/src/main/java/it/unibo/crossyroad/model/api/AbstractPositionable.java#L79-L93`

Utilizzo di `Stream` e lambda expressions

Usate spesso nel progetto, un esempio è il seguente. Permalink: `https://github.com/TheRealGtx/OOP25-crossy-road/blob/5df9c97a5c6fc9cf7d696d8a84d8b6ae0ff/src/main/java/it/unibo/crossyroad/model/impl/managers/GameManagerImpl.java#L113-L117`

Utilizzo di `JavaFX`

Sviluppo della GameView. Permalink: `https://github.com/TheRealGtx/OOP25-crossy-road/blob/5df9c97a5c6fc9cf7d696d8a84d8b6ae0ff6ad8a/src/main/java/it/unibo/crossyroad/view/impl/GameViewImpl.java#L79-L164`

Implementazione dell'algoritmo `BFS`

Usato per controllare la validità delle generazioni di ostacoli/pickable. Permalink: `https://github.com/TheRealGtx/OOP25-crossy-road/blob/5df9c97a5c6fc9cf7d69/src/main/java/it/unibo/crossyroad/model/impl/managers/GameManagerImpl.java#L220-L255`

Utilizzo di `Optional`

Permalink: `https://github.com/TheRealGtx/OOP25-crossy-road/blob/5df9c97a5c6fc9cf7d696d8a84d8b6ae0ff6ad8a/src/main/java/it/unibo/crossyroad/model/impl/managers/GameManagerImpl.java#L230-L238`

3.2.2 Alessio Magnani

Utilizzo di `JavaFX`

Permalink: `https://github.com/TheRealGtx/OOP25-crossy-road/blob/5df9c97a5c6fc9cf7d696d8a84d8b6ae0ff6ad8a/src/main/java/it/unibo/crossyroad/view/impl/MenuViewImpl.java`

Utilizzo di lambda expressions

Permalink: `https://github.com/TheRealGtx/OOP25-crossy-road/blob/5df9c97a5c6fc9cf7d696d8a84d8b6ae0ff6ad8a/src/main/java/it/unibo/crossyroad/view/impl/MenuViewImpl.java#L155-L186`

Utilizzo di `Stream` e `Optional`

Permalink: `https://github.com/TheRealGtx/OOP25-crossy-road/blob/5df9c97a5c6fc9cf7d696d8a84d8b6ae0ff6ad8a/src/main/java/it/unibo/crossyroad/model/impl/managers/GameManagerImpl.java#L288-L294`

Utilizzo di `Function`

Permalink: `https://github.com/TheRealGtx/OOP25-crossy-road/blob/5df9c97a5c6fc9cf7d696d8a84d8b6ae0ff6ad8a/src/main/java/it/unibo/crossyroad/controller/impl/AppControllerImpl.java#L28-L35`

Utilizzo di `Consumer`

Permalink: `https://github.com/TheRealGtx/OOP25-crossy-road/blob/5df9c97a5c6fc9cf7d696d8a84d8b6ae0ff6ad8a/src/main/java/it/unibo/crossyroad/model/impl/managers/GameManagerImpl.java#L331-L340`

Utilizzo di `Range` dalla libreria Google Guava

Permalink: `https://github.com/TheRealGtx/OOP25-crossy-road/blob/5df9c97a5c6fc9cf7d696d8a84d8b6ae0ff6ad8a/src/main/java/it/unibo/crossyroad/model/api/AbstractPositionable.java#L68-L74`

Utilizzo di `Optional`

Permalink: `https://github.com/TheRealGtx/OOP25-crossy-road/blob/5df9c97a5c6fc9cf7d696d8a84d8b6ae0ff6ad8a/src/main/java/it/unibo/crossyroad/model/impl/managers/GameManagerImpl.java#L124-L125`
`https://github.com/TheRealGtx/OOP25-crossy-road/blob/5df9c97a5c6fc9cf7d696d8a84d8b6ae0ff6ad8a/src/main/java/it/unibo/crossyroad/view/impl/MenuViewImpl.java#L174`

3.2.3 Golinucci Mattia

Utilizzo della libreria esterna `Jackson`

Permalink: `https://github.com/TheRealGtx/OOP25-crossy-road/blob/5df9c97a5c6fc9cf7d696d8a84d8b6ae0ff6ad8a/src/main/java/it/unibo/crossyroad/model/impl/GameParametersImpl.java#L215-L231`

Utilizzo di `Range` della libreria `Guava` di Google

Permalink: `https://github.com/TheRealGtx/OOP25-crossy-road/blob/5df9c97a5c6fc9cf7d696d8a84d8b6ae0ff6ad8a/src/main/java/it/unibo/crossyroad/model/api/chunks/AbstractActiveChunk.java#L69-L75`

Utilizzo di `JavaFX`

Permalink: `https://github.com/TheRealGtx/OOP25-crossy-road/blob/5df9c97a5c6fc9cf7d696d8a84d8b6ae0ff6ad8a/src/main/java/it/unibo/crossyroad/view/impl/GameViewImpl.java#L79-L164`

Utilizzo di `Stream` e `Lambda expression`

Permalink: `https://github.com/TheRealGtx/OOP25-crossy-road/blob/5df9c97a5c6fc9cf7d696d8a84d8b6ae0ff6ad8a/src/main/java/it/unibo/crossyroad/model/impl/managers/GameManagerImpl.java#L480-L490`

Utilizzo di `Optional`

Permalink: `https://github.com/TheRealGtx/OOP25-crossy-road/blob/5df9c97a5c6fc9cf7d696d8a84d8b6ae0ff6ad8a/src/main/java/it/unibo/crossyroad/model/impl/managers/GameManagerImpl.java#L528-L532`

3.2.4 Lorenzo Baldazzi

Utilizzo della libreria esterna `Jackson`

Permalink:
Link: loadSkinsFromStream()
Link: hasRequiredFields()
Link: save()
Link: load()

Utilizzo di `JavaFX`

Permalink:
Link: ShopViewImpl

Utilizzo di `Stream` e `Lambda expression`

Usate spesso nel progetto, i seguenti sono degli esempi:
PermaLink:
Link: tryUnlock() e unlockDefaultSkin()
Link: StateManagerImpl()
Link: activateSkin()

Utilizzo di `Optional`

Permalink:
Link: getTextValue()

Utilizzo di `Generics`

Permalink:
Link: AbstractTestPowerUp

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Giuliano Manzi

Durante la fase di progettazione iniziale ritengo di aver contribuito molto al progetto, avendo proposto molte delle soluzioni che alla fine sono state implementate. Ho immaginato il gioco da zero senza cercare di forzare alcun pattern specifico, solo ciò che ritenevo fosse ideale per il gioco, alla fine i pattern sono apparsi da soli. Per quanto riguarda la scrittura di codice ho cercato l'approccio più elegante e improntato al riuso possibile, spero di essermi quantomeno avvicinato alla soluzione migliore. La divisione del lavoro è stata efficace, ognuno di noi ha avuto una porzione bilanciata di lavoro, che è stata rispettata. Ho avuto delle difficoltà a capire alcuni aspetti di progettazione/programmazione riguardo alle parti dei miei colleghi, alla quale ho dovuto chiedere aiuto per interfacciarmi con le loro sezioni. Questo mi ha fatto capire che fatico a guardare i problemi (e le possibili soluzioni) da un punto di vista diverso. Sono soddisfatto del prodotto finale e sopratutto del team, che si è dimostrato all'altezza della sfida che ci siamo posti a inizio progetto.

4.1.2 Alessio Magnani

Personalmente sono molto soddisfatto del risultato ottenuto e dal lavoro svolto da tutto il gruppo. Lavorare in gruppo non è mai troppo facile dall'aspetto dell'organizzazione e suddivisione dei compiti, poiché è difficile essere tutti d'accordo su una determinata implementazione o risoluzione a un problema, ma nonostante ciò siamo riusciti a trovare sempre un punto d'incontro che andasse bene a tutti. In particolare sono contento di come è stata svol-

ta la progettazione dell'intero gioco, perché ci siamo soffermati svariate ore su quale fosse la struttura migliore, proponendo varie idee, e ho apprezzato che molte delle mie idee siano state approvate dai miei colleghi. Una volta accordati sulla struttura delle classi e interfacce, lo sviluppo del progetto è proceduto in maniera fluida e senza particolari criticità, perché ognuno aveva un compito preciso da fare che non interferisse con il lavoro degli altri. Nello specifico mi è piaciuto come abbiamo operato su Git, ognuno con propri branch e opportuni merge quando una feature era pronta: pratica nuova che prima d'ora in progetti passati non avevo mai messo in pratica. Per quanto riguarda il gioco, sono riuscito a portare a termine ciò che mi ero imposto dall'inizio, sviluppando anche parti opzionali. In futuro è sicuramente possibile aggiungere nuove funzionalità al gioco, come per esempio la selezione del livello di difficoltà che si vuole affrontare, o migliorare quelle esistenti, creando per esempio nuovi `Power Up`, `Skin` e `Chunk`.

4.1.3 Golinucci Mattia

Durante la realizzazione del progetto ho compreso quanto una buona progettazione iniziale sia fondamentale. Definire una struttura solida fin dall'inizio ci ha permesso di introdurre nuove funzionalità e modificare parti del gioco senza dover rivedere l'architettura generale, facilitando anche l'integrazione tra i moduli sviluppati da me e dai miei colleghi. Nel corso dello sviluppo ci siamo confrontati spesso per discutere le scelte progettuali e risolvere i punti più critici. Le idee non erano sempre allineate, ma il confronto è stato costruttivo e ha portato a soluzioni condivise. Personalmente ho contribuito proponendo diverse opzioni e piccole funzionalità aggiuntive che hanno migliorato l'esperienza complessiva del gioco. La fase di integrazione delle componenti è risultata fluida grazie al lavoro svolto in progettazione: nella maggior parte dei casi è stato sufficiente adattare alcune interfacce o introdurre nuovi metodi senza modifiche strutturali rilevanti. Dal punto di vista personale ho migliorato la mia capacità di lavorare su codice condiviso con uso di GitHub e di adattarmi alle scelte architetturali del gruppo, comprendendo meglio l'importanza della comunicazione. Il progetto è stato pensato per essere estendibile nel tempo. Tra i possibili sviluppi futuri vedo l'aggiunta di nuovi power-up, ostacoli attivi e passivi e l'introduzione di nuovi ambienti o varianti dei chunk esistenti, mantenendo la struttura modulare attuale.

4.1.4 Lorenzo Baldazzi

Arrivati alla fine della realizzazione del progetto posso dire di essere molto soddisfatto del risultato ottenuto. Non credevo che lavorare in team potesse rivelarsi cosi divertente e stimolante. Partendo dalla fase progettuale a quella implementativa, il team si è dimostrato molto unito e collaborativo, ciascun membro ha sempre avuto modo di esporre le proprie idee e alla fine si è sempre scelta quella ritenuta più valida. Questo progetto mi ha permesso di comprendere quanto sia fondamentale definire una struttura ben organizzata fin dall'inizio per non incorrere in problemi durante lo sviluppo. Personalmente ho contribuito principalmente nella parte di `Skin` e `PowerUp` e sono contento che molte delle mie idee abbiano trovato riscontro positivo tra i membri del team. Ho cercato di scrivere del codice il più leggibile e "clean" possibile, cercando di evitare ripetizione di codice inutile. L'unica parte in cui ho riscontrato delle difficoltà è stato l'utilizzo di `GitHub` con il quale non avevo molta confidenza, tuttavia durante lo sviluppo del progetto sento di aver acquisito maggior confidenza e mi sento decisamente più sicuro nel suo utilizzo.
Fin dall'inizio abbiamo organizzato la struttura del progetto in modo tale che in futuro sia meno impegnativo implementare nuove funzionalità, in particolare per quanto riguarda la parte su cui mi sono concentrato, sarà possibile aggiungere nuovi `PowerUp` o `Skin` in un modo facile e intuitivo.

4.2 Difficoltà incontrate e commenti per i docenti

Complessivamente siamo rimasti soddisfatti delle lezioni tenute dai docenti. Tutti gli argomenti sono stati spiegati ampiamente e in modo chiaro. Tuttavia ci sarebbe piaciuto avere qualche lezione in più riguardo a costruzione di progetti di scala un po' più grande, circa come il nostro. Questo probabilmente ci avrebbe semplificato il lavoro iniziale di creazione delle classi, sapendo meglio dove si deve limitare la responsabilità di ciascuna di esse, cosa ciascuna di esse deve delegare e quali dipendenze deve avere. Comprendiamo allo stesso tempo che gli argomenti da spiegare durante il corso siano molti, quindi risulta complicato aggiungere altre ore per mostrare esempi di progetti di larga scala.

Appendice A

Guida utente

All'avvio del gioco ci si troverà davanti a un menù con 4 opzioni.

- `Play`: inizia la partita.
- `Shop`: accede al negozio.
- `Reset`: reimposta il gioco, eliminando i salvataggi dell'utente.
- `Exit`: salva i cambiamenti dell'utente e esce dal gioco.

Una volta in partita è possibile muoversi nello spazio usando i comandi `W`, `A`, `S`, `D` o in alternativa le frecce ↑, ←, ↓, →. Premendo il tasto `Esc` è possibile tornare al menù. Per raccogliere monete e power-up è sufficiente posizionare il player sopra di essi.
Una volta nello shop il giocatore può acquistare nuove skin o selezionare le skin già in suo possesso. Tutte le modifiche vengono salvate automaticamente e il giocatore ritroverà le sue skin sbloccate/selezionate una volta rientrato nel gioco.

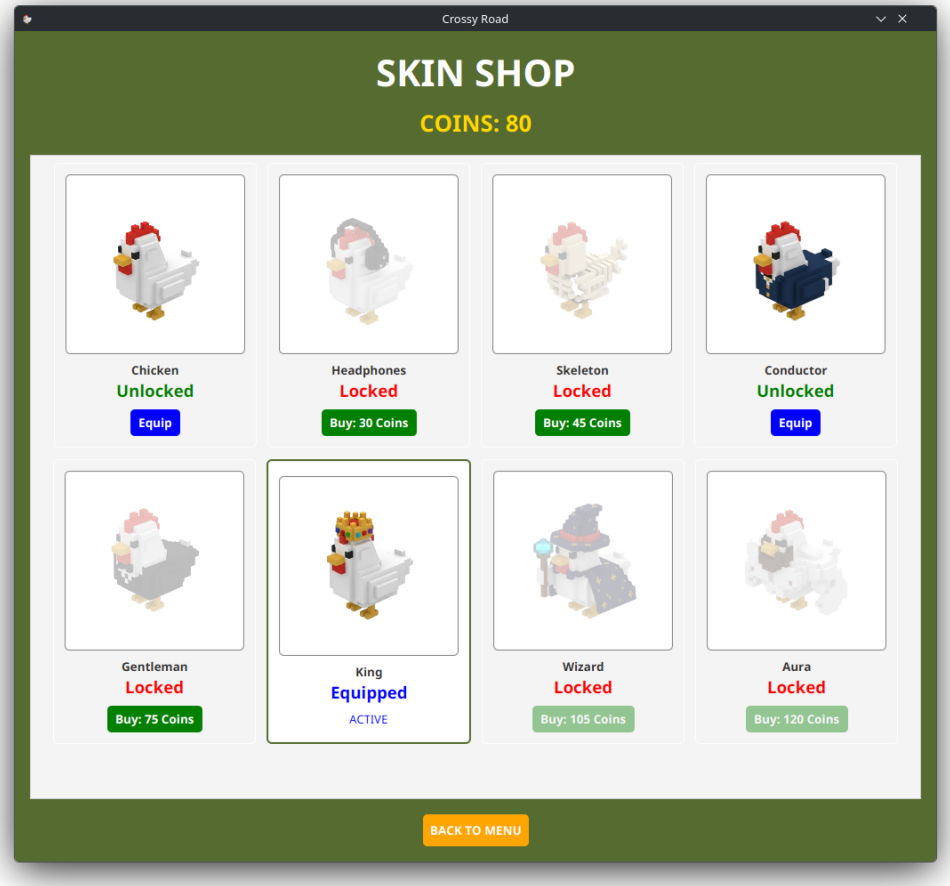

Figura A.1: Cliccando su buy si può acquistare la skin, cliccando su equip la si può selezionare.

Appendice B

Esercitazioni di laboratorio

B.1 giuliano.manzi@studio.unibo.it

- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=207193#p284906
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=207921#p285569
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=208718#p286583
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=209589#p288168
- Laboratorio 11: https://virtuale.unibo.it/mod/forum/discuss.php?d=210617#p289400
- Laboratorio 12: https://virtuale.unibo.it/mod/forum/discuss.php?d=211539#p290204

B.2 alessio.magnani@studio.unibo.it

- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=206731#p284005
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=207193#p284531
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=207921#p285437

- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=208718#p286541
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=209589#p288482
- Laboratorio 11: https://virtuale.unibo.it/mod/forum/discuss.php?d=210617#p289504
- Laboratorio 12: https://virtuale.unibo.it/mod/forum/discuss.php?d=211539#p290255

B.3 mattia.golinucci2@studio.unibo.it

- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=206731#p284102
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=207193#p284852
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=207921#p285848
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=208718#p286490
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=209589#p288410
- Laboratorio 11: https://virtuale.unibo.it/mod/forum/discuss.php?d=210617#p289568
- Laboratorio 12: https://virtuale.unibo.it/mod/forum/discuss.php?d=211539#p290266

B.4 lorenzo.baldazzi2@studio.unibo.it

- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=207193#p285072
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=207921#p286202

- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=208718#p287257
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=209589#p288516
- Laboratorio 11: https://virtuale.unibo.it/mod/forum/discuss.php?d=210617#p289691
- Laboratorio 12: https://virtuale.unibo.it/mod/forum/discuss.php?d=211539#p290720