

# ARCHITECT OF MAYHEM

Gioele Foschi

[gioele.foschi@studio.unibo.it](mailto:gioele.foschi@studio.unibo.it)

Nicholas Magi

[nicholas.magi@studio.unibo.it](mailto:nicholas.magi@studio.unibo.it)

Ludovico Maria Spitaleri

[ludovico.spitaleri@studio.unibo.it](mailto:ludovico.spitaleri@studio.unibo.it)

Elettra Ventura

[elettra.ventura@studio.unibo.it](mailto:elettra.ventura@studio.unibo.it)

04 FEBBRAIO 2024

# Indice

<b>1 Analisi</b>	<b>3</b>
1.1 Descrizione e requisiti . . . . .	3
1.2 Modello del Dominio . . . . .	5
<b>2 Design</b>	<b>7</b>
2.1 Architettura . . . . .	7
2.2 Design dettagliato . . . . .	9
2.2.1 Gioele Foschi . . . . .	9
2.2.2 Nicholas Magi . . . . .	12
2.2.3 Ludovico Spitaleri . . . . .	18
2.2.4 Elettra Ventura . . . . .	22
<b>3 Sviluppo</b>	<b>27</b>
3.1 Testing automatizzato . . . . .	27
3.2 Note di sviluppo . . . . .	28
3.2.1 Gioele Foschi . . . . .	28
3.2.2 Nicholas Magi . . . . .	29
3.2.3 Ludovico Spitaleri . . . . .	30
3.2.4 Elettra Ventura . . . . .	31
<b>4 Commenti finali</b>	<b>33</b>
4.1 Autovalutazione e lavori futuri . . . . .	33
4.1.1 Gioele Foschi . . . . .	33
4.1.2 Nicholas Magi . . . . .	33
4.1.3 Ludovico Spitaleri . . . . .	34
4.1.4 Elettra Ventura . . . . .	34
4.2 Difficoltà incontrate e commenti per i docenti . . . . .	35
<b>A Guida utente</b>	<b>36</b>
A.1 Editor . . . . .	37
A.2 Gameplay . . . . .	38

<b>B Esercitazioni di laboratorio</b>	<b>39</b>
B.1 gioele.foschi@studio.unibo.it . . . . .	39
B.2 nicholas.magi@studio.unibo.it . . . . .	39
B.3 ludovico.spitaleri@studio.unibo.it . . . . .	39
B.4 elettra.ventura@studio.unibo.it . . . . .	40

# Capitolo 1

## Analisi

### 1.1 Descrizione e requisiti

Il gioco “Architect of Mayhem” prende spunto dal gioco ”World’s Hardest Game”, che consiste in diversi livelli ”visti dall’alto” in cui una pallina, che parte da un punto A, deve raggiungere un punto B evitando diversi ostacoli.

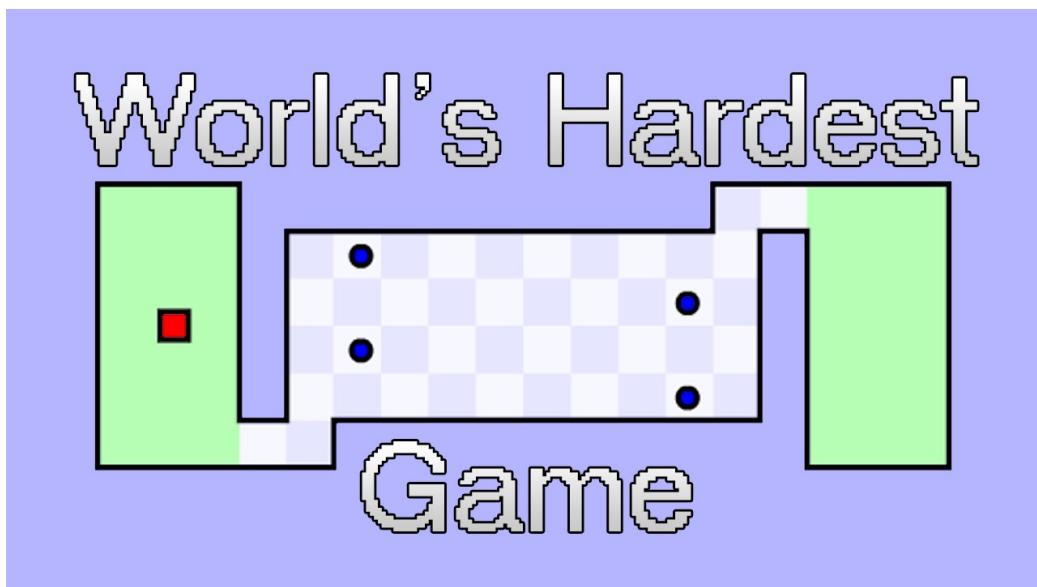


Figura 1.1: Riferimento al gioco 'The World's Hardest Game'

L’applicativo mette a disposizione un editor per la creazione di un livello. Per editor si intende un’interfaccia che permette all’utente di creare, manipolare ed eliminare gli elementi che compongono un livello, selezionati da una collezione di oggetti (a.k.a. *tile*) predefinita. L’utente può creare diversi

livelli; ognuno di essi può essere salvato in qualsiasi momento e, una volta soddisfatto del prodotto, può essere giocato. La collezione di oggetti posseduti dal giocatore potrà essere ampliata tramite un apposito shop. La valuta del gioco è rappresentata dal credito dell'utente, che incrementa in base alle monete collezionate ad ogni partita solo in caso di vittoria.

### **Requisiti funzionali**

- Realizzare un menu di selezione da cui l'utente può scegliere i vari elementi del livello in costruzione.
- Fornire uno spazio di piazzamento “*tile-based*” degli elementi.
- Garantire la possibilità di annullare l'ultima operazione eseguita nell'editor.
- Garantire la possibilità di salvare lo stato di un livello in qualsiasi momento.
- Garantire la possibilità di giocare un livello in fase di creazione in qualsiasi momento.
- Predisporre di un sistema di punteggio minimale, basato su oggetti collezionabili generati all'interno del livello.
- Garantire l'adattamento dell'interfaccia utente alla risoluzione dello schermo.
- Una volta avviato un livello, il giocatore deve essere in grado di muoversi all'interno dello spazio designato dall'utente.
- Fornire la possibilità di acquistare alcuni tile di gioco tramite un apposito shop.
- Garantire il salvataggio dello stato dell'utente ad ogni operazione di modifica.

### **Requisiti non funzionali**

- Il gioco deve essere fluido (30 FPS minimi) e reattivo agli input dell'utente.
- L'applicazione deve essere compatibile con la maggior parte dei sistemi operativi esistenti.
- L'interfaccia grafica dovrà essere esteticamente gradevole e user-friendly.

## 1.2 Modello del Dominio

L’unità fondamentale dell’applicazione è il Game Object, di cui sono previste diverse categorie:

- **Blocchi**: oggetti basilari di costruzione per la definizione dello spazio (muro, pavimento, arrivo, ecc. . . ).
- **Traguardo**: speciale tipo di blocco che indica la fine di un livello.
- **Ostacoli**: oggetti che ostacolano il completamento del livello.
- **Giocatore**: oggetto direttamente comandato da input esterno.
- **Collezionabili**: oggetti per la manipolazione punteggio.

L’applicazione si divide in due parti principali: uno shop e un editor. Entrambe lavorano su una collezione di Game Object. L’editor permette di creare livelli composti da elementi di questa collezione, mentre lo shop permette di ampliarla acquistandone di nuovi. Tramite quest’ultimo viene speso il credito accumulato ad ogni partita dall’utente, ottenuto giocando i suoi stessi livelli.

Questa collezione mantiene uno stato consistente all’interno dei vari contesti dell’applicazione ed è contenuta e gestita da un’entità “Stato dell’utente”. Ogni livello può essere creato in una di due modalità:

- “**sandbox**”: nessuna restrizione sugli oggetti a disposizione dell’utente. L’utente può usare tutti gli oggetti del gioco ed è libero di piazzarli a suo piacimento.
- “**normale**”: gli oggetti a disposizione dell’utente sono limitati. Il piazzamento di alcuni oggetti è vincolato al rispetto di alcuni criteri imposti dalla griglia dell’editor.

Se il livello in costruzione si trova in uno stato consistente, l’utente può avviarlo: alla pressione di un pulsante la possibilità di modifica viene disattivata e tutti gli oggetti “prendono vita”.

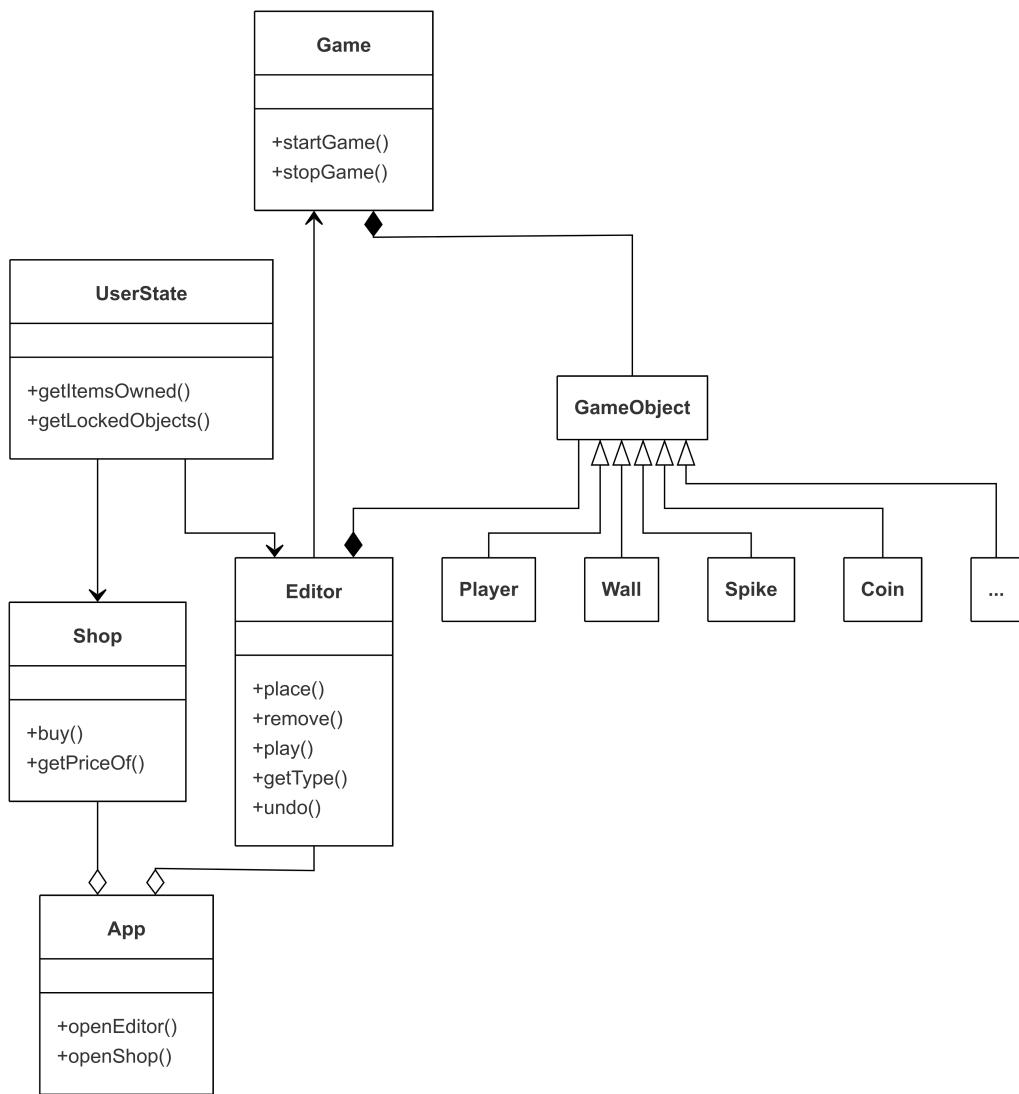


Figura 1.2: Semplice UML del dominio dell'applicazione.

# Capitolo 2

## Design

### 2.1 Architettura

L'applicazione segue un'architettura MVC. Tutte le view sono create a partire da un relativo controller, e tutti i controller implementati vengono orchestrati da un **MainController**. Ciascun controller ha poi associato un relativo Model che memorizza le diverse informazioni e business logic dei vari contesti. L'entry point di tutto il programma è **MainController**, che fa partire prima di tutti il **MenuController**. Viene costruita la corrispettiva **MenuView**, che viene agganciata al controller e mostrata tramite scambio di messaggi con **MainView**. Ad ogni segnale di cambio finestra, è il controller attivo a sapere quale view creare per poter passare alla schermata successiva (interagendo con il **MainController**), mentre la logica di 'cambio al pannello precedente' è contenuta in **MainController** - poiché comune a tutti i controller.

L'organizzazione e la sequenzialità di tutte le parti viene illustrata nel seguente diagramma.

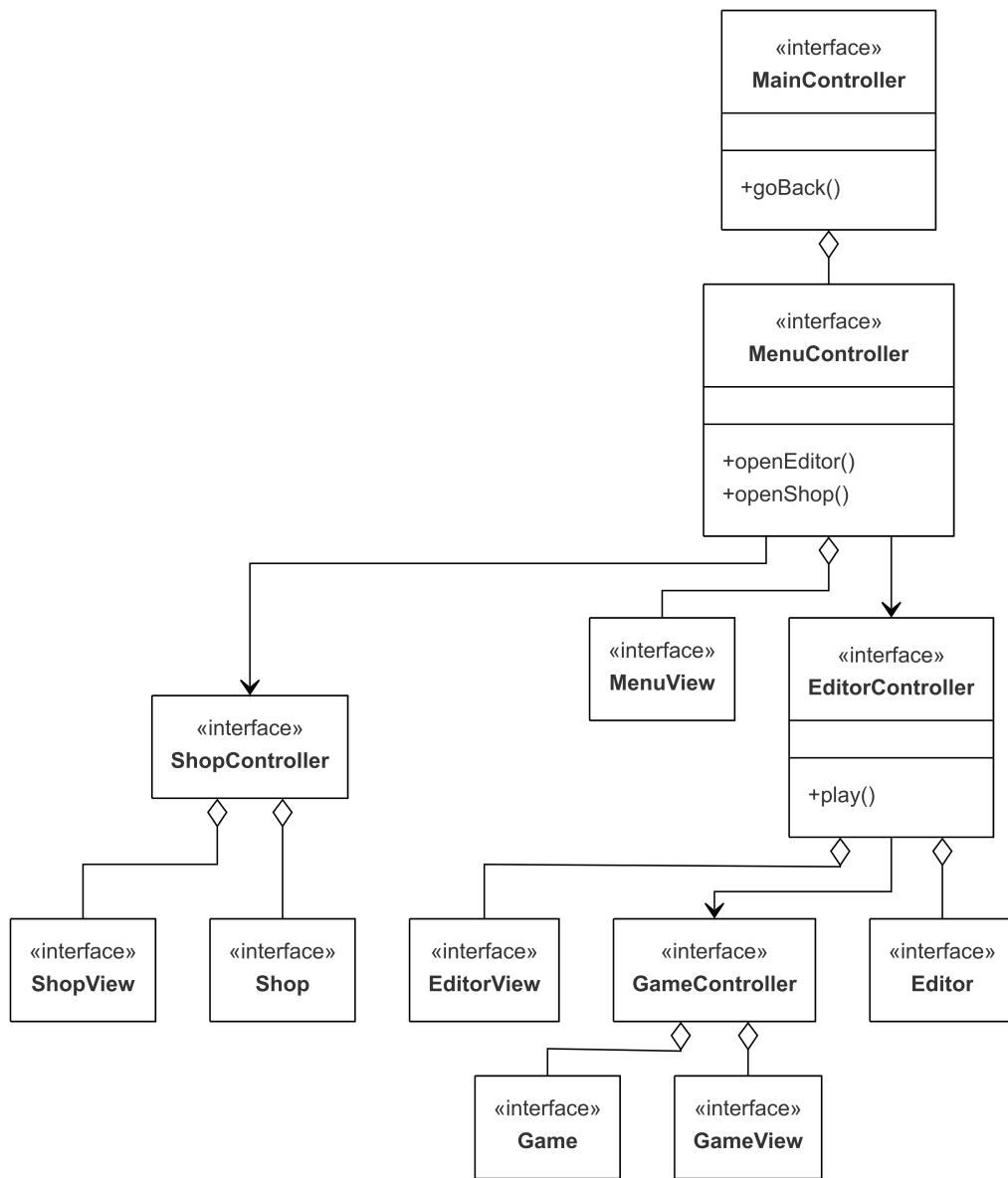


Figura 2.1: Struttura MVC dell'applicazione.

## 2.2 Design dettagliato

### 2.2.1 Gioele Foschi

#### Salvataggio del Livello

**Problema** Si vuole avere la possibilità di salvare un livello creato, in modo da poter riaprirlo in successivo.

**Soluzione** Durante lo sviluppo si è deciso di dividere il salvataggio in due file diversi: Il file dei metadati: contenente dati come nome del livello, dimensioni del livello, tipologia dell'editor e infine l'id con cui verrà salvato il file su disco. Il file contenente il livello effettivo: ottenuto tramite la serializzazione binaria della mappa.

#### Diversi tipi di "Constraints"

**Problema** Non tutti gli oggetti possono essere piazzati liberamente. Alcuni oggetti possono essere piazzati solamente seguendo certe regole: Blocchi adiacenti, Numero massimo di blocchi, numero minimo di blocchi, etc.. Si vuole avere la possibilità di aggiungere nuove regole facilmente.

**Soluzione** Il sistema per la realizzazione delle regole di piazzamento utilizza il *pattern Factory* (`MapConstraintsFactory`) che permette di creare diversi tipi di "*constraints*" (regole) per piazzare un certo tipo di oggetto. Le regole possono essere applicate sia agli oggetti che alle relative categorie.

#### Diverse modalità di editor

**Problema** Alla creazione del nuovo editor di livello, l'applicazione si presenta con una scelta di tipologia di editor. Alla scelta di una tipologia specifica si vogliono impostare le regole predefinite per quella tipologia. Si vuole suddividere le regole in due gruppi:

- Le regole che devono essere applicate al piazzamento dell'oggetto;
- Le regole che devono essere applicate prima di giocare il livello.

**Soluzione** L'aggiunta delle diverse regole verrà delegata ad una classe che, utilizzando il *pattern Factory* (`GridConstraintProvider`), genera una struttura adeguata per l'utilizzo delle diverse regole. Esso in base alla tipologia di editor scelto aggiungerà alla classe apposita (`MapConstraintsContainer`)

le apposite regole da applicare, sia al piazzamento dell'oggetto, sia all'avvio del livello.

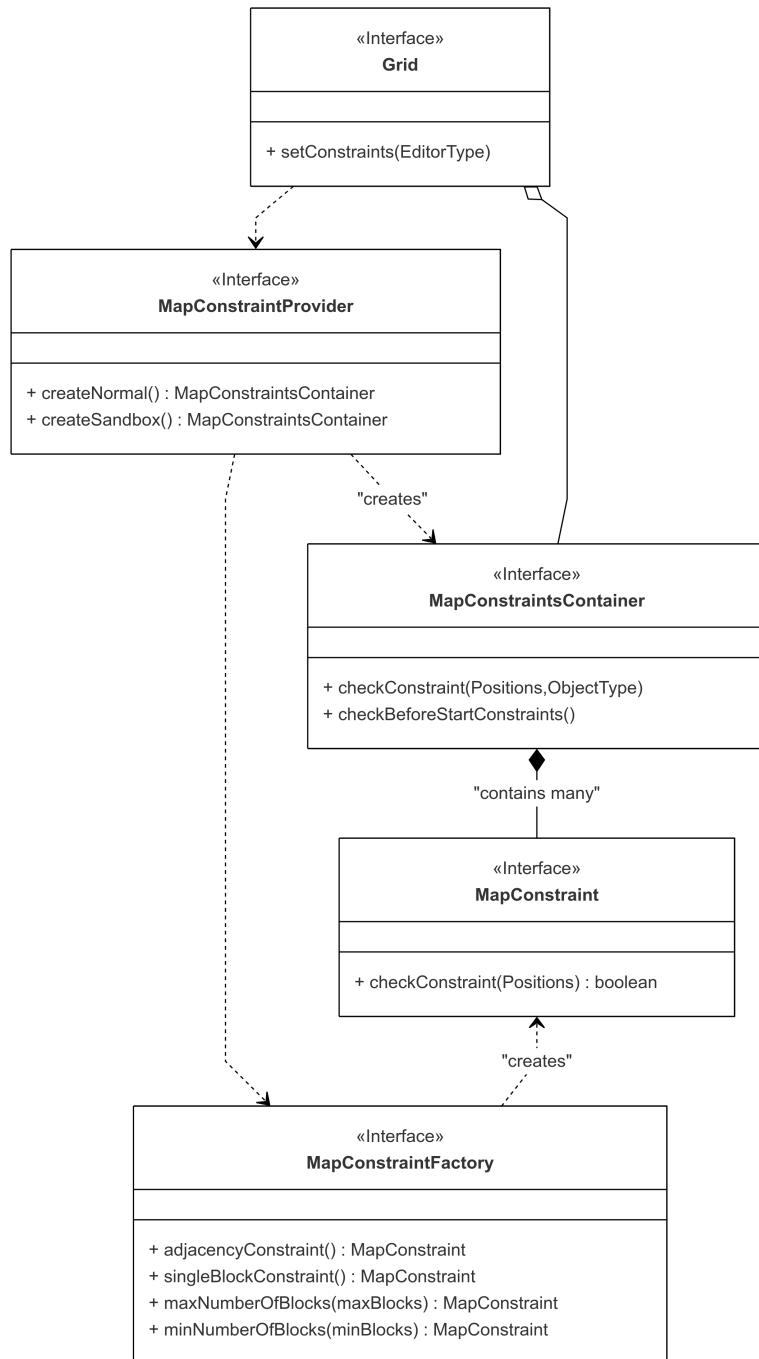


Figura 2.2: Sistema di generazione di regole di piazzamento.

## Annullamento dell'ultima operazione

**Problema** Si vuole avere la possibilità di annullare l'ultima operazione eseguita sulla griglia, che sia il piazzamento di un oggetto o che sia la rimozione degli oggetti da una area. Si crea la necessità di salvarsi i diversi stati della griglia ad ogni cambiamento, e creare qualche meccanismo che permette di recuperare il vecchio stato.

**Soluzione** Il sistema di recupero dello stato precedente (undo) utilizza il *pattern Memento*.<sup>1</sup> La classe che gestisce la griglia (**GridModel**) avrà il compito di salvare gli stati prima di una modifica, mentre la classe **MementoCaretaker**, consisterà nel tenere traccia degli stati salvati, e dell'ordine in cui sono stati salvati (stack). Solo la classe della griglia (**GridImpl**) sarà in grado di ripristinare uno stato precedente, in modo da non infrangere l'incapsulamento.

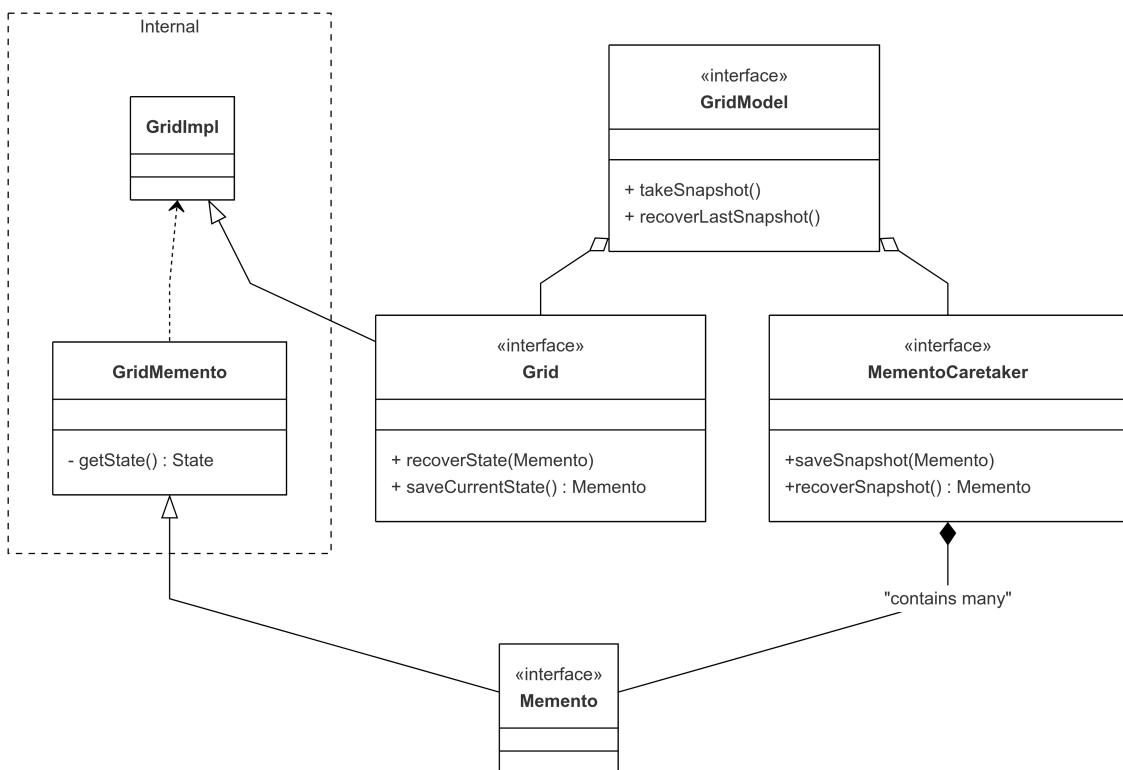


Figura 2.3: Implementazione del pattern *Memento*.

<sup>1</sup><https://java-design-patterns.com/patterns/memento/>

## 2.2.2 Nicholas Magi

**Problema** L'applicazione necessita di qualcosa che permetta di memorizzare delle informazioni relative al giocatore. Queste informazioni devono essere organizzate in modo da mantenere uno stato consistente all'interno dell'applicazione, in qualsiasi contesto vengano utilizzate.

**Soluzione** In prima battuta, la soluzione è rappresentata da un'entità `UserState` che modella quali informazioni dell'utente deve memorizzare e le operazioni necessarie per la manipolazione di esse. Tuttavia, raggruppare tutto quanto in una sola interfaccia espone più elementi di quanti necessari a parti dell'applicazione che si limitano ad una sola lettura di queste informazioni (come nel caso dell'`EditorView`), rischiando di comprometterne l'integrità e violando il SRP (*Single Responsibility Principle*) dell'interfaccia. Ho pertanto ritenuto necessario racchiudere le informazioni e relative operazioni di sola lettura in un record `UserStateInfo`, mentre le operazioni che modificano effettivamente lo stato sono modellate dall'interfaccia `UserStateManager`.

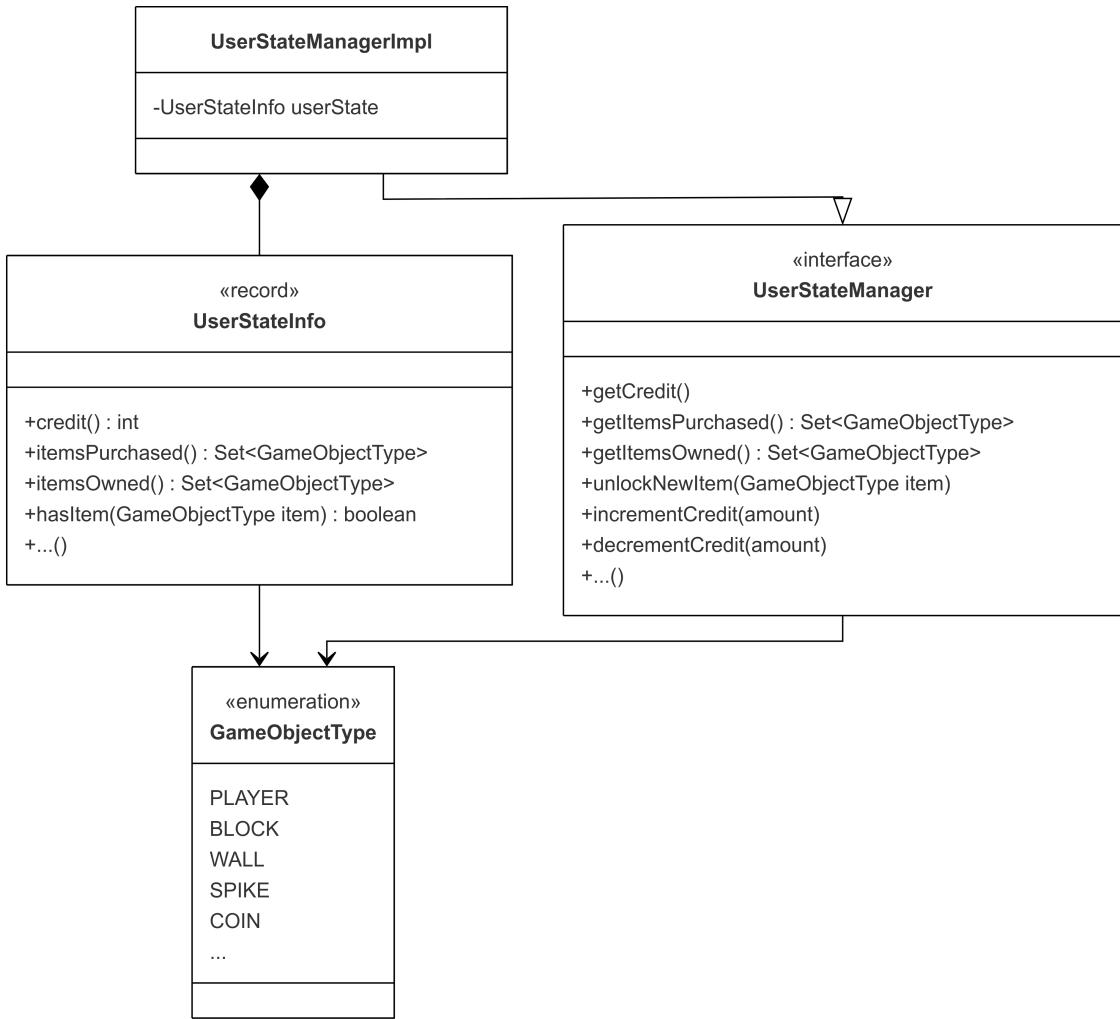


Figura 2.4: Implementazione di `UserState`.

**Problema** Con la precedente soluzione ho individuato un modo per progettare lo `UserState`, ma non per mantenerlo consistente.

**Soluzione** Mi sono venute in mente due ipotesi per raggiungere l'obiettivo:

1. progettare lo `UserState` secondo il *pattern Singleton*, che prevede la creazione di una sola istanza della classe che viene fornita a qualunque parte del programma la richieda.
2. passare attraverso la serializzazione e fare in modo che lo stato effettivo di un utente non sia incarnato in una classe, ma bensì in un file che viene letto e scritto ad ogni operazione.

Ciascuna opzione ha i suoi pro e contro: entrambe richiedono particolare attenzione in ambienti multithreading, specialmente la serializzazione (occorre evitare che corse critiche possano corrompere il file) - anche se il progetto non le prevede. Il pattern Singleton è abbastanza semplice da realizzare, ma viola il *SRP* della classe - poiché la classe si deve preoccupare di aggiornare lo stato e fornirlo all'esterno. Ho quindi deciso di percorrere la strada della serializzazione: uno **UserStateSerializer** che permette di leggere e salvare su un file.

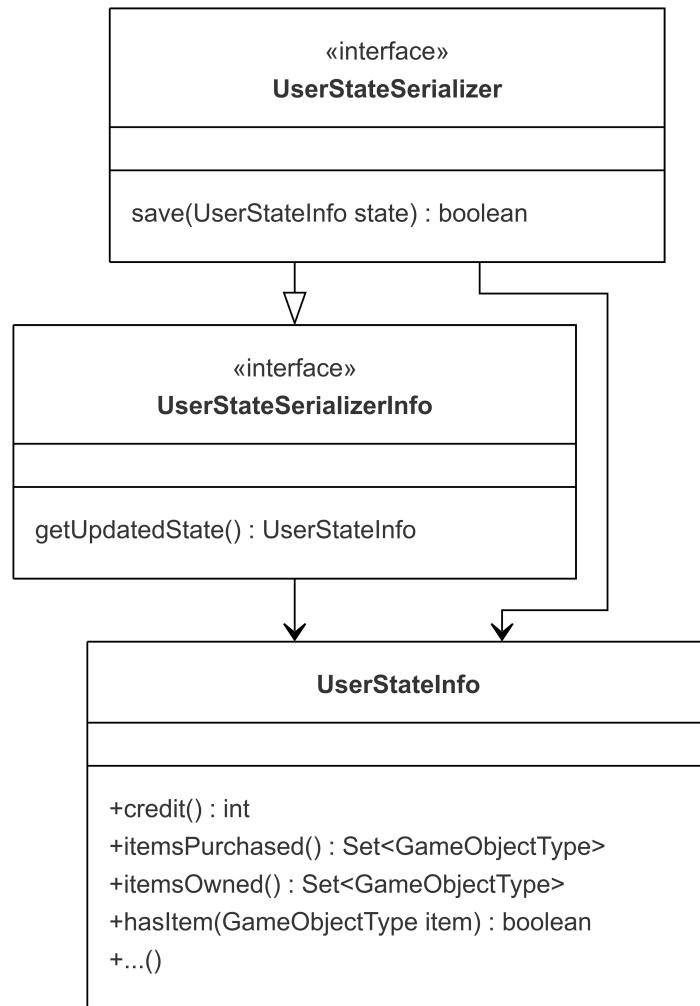


Figura 2.5: Primo UML dello UserStateSerializer.

**Problema** La serializzazione si compone di diversi metodi e controlli “di basso livello” che non servono sempre e che un programmatore esterno non è tenuto a conoscere.

**Soluzione** Le operazioni esposte sono infatti solamente 2: `save()` e `getUpdatedState()`. L’interfaccia `UserStateSerializer` rappresenta, secondo il *pattern Facade*, una semplificazione all’accesso e utilizzo di funzioni di serializzazione. Le due operazioni sono inoltre divise su due interfacce: un’interfaccia `UserStateSerializerInfo` per l’operazione di sola lettura, e un’interfaccia `UserStateSerializer` che aggiunge la possibilità di scrittura. Questo sempre per il motivo precedentemente spiegato: prendendo come riferimento le schermate di Editor e Shop, la prima legge la collezione di oggetti dell’utente e la mette a disposizione nel menu laterale e nella griglia; la seconda deve poter leggere e aggiornare credito e collezione di oggetti in caso di acquisto. Ho ritenuto corretto poter fornire loro diversi livelli di accesso alla serializzazione. Da notare inoltre come le operazioni previste nelle interfacce non dipendono dalla logica di serializzazione implementata: nel caso di questo progetto ho optato per serializzare in JSON (`UserStateSerializerJSON`), ma qualora decidessi di fare diversamente mi basterebbe creare una nuova classe che implementa `UserStateSerializer` e riscrivere i due metodi con una logica diversa - nel pieno rispetto dell’*Open-Closed Principle*.

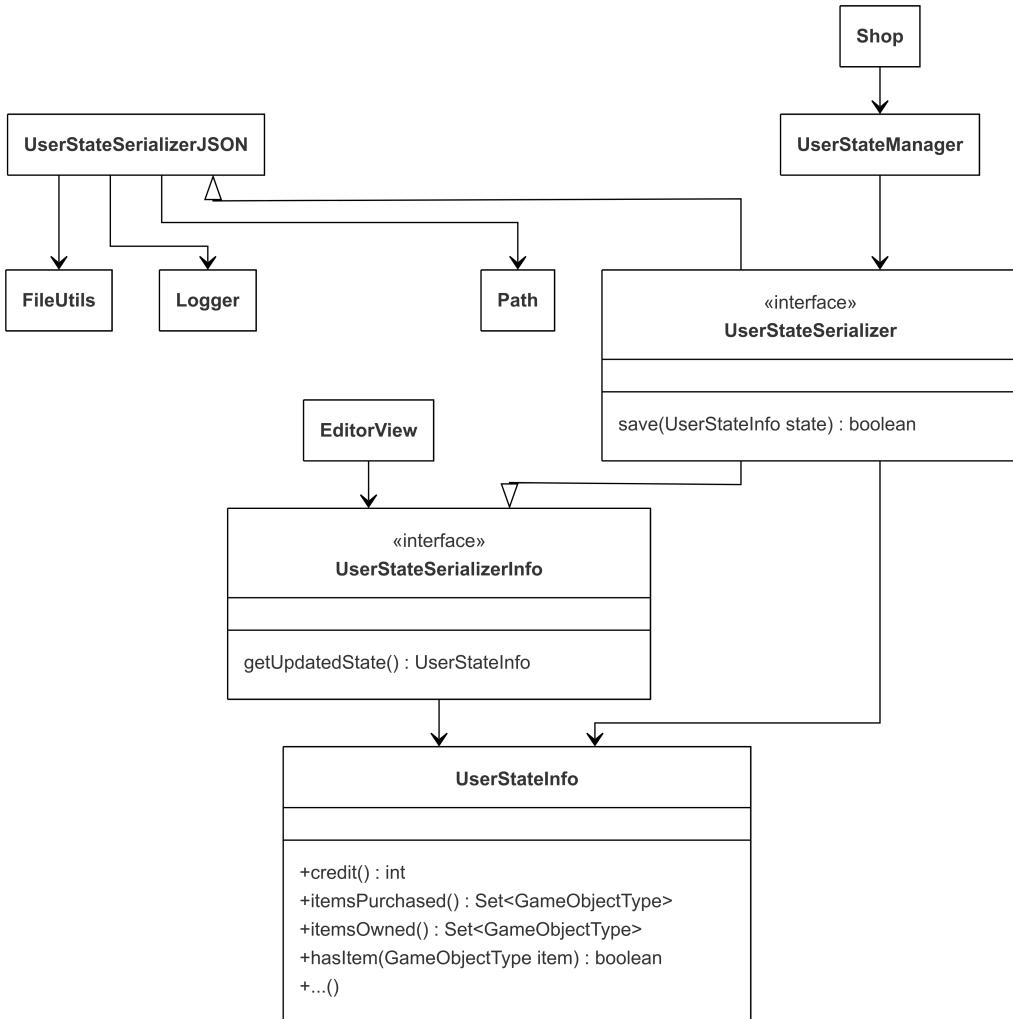


Figura 2.6: UML più completo dello UserStateSerializer.

## Main Controller

**Problema** L'applicazione si compone di diverse schermate (Home, Shop, Editor e Game). Trovare un sistema centralizzato per passare da una schermata all'altra secondo un ordine prestabilito.

**Soluzione** Ogni **View** dell'applicazione viene creata da un relativo **Controller**. Questi ultimi sono raccolti e gestiti in un **MainController**, che li memorizza in uno stack - in cui l'ultimo elemento è il controller corrispondente alla view che si sta visualizzando nella finestra **MainView**. Il **MainController** viene direttamente invocato ogni qualvolta si cerca di tornare indietro ad una

schermata precedente, mentre è responsabilità dei singoli controller chiamare uno ‘switcher’ che permette di navigare alla schermata successiva. Il metodo ‘switcher’ è modellato dall’interfaccia funzionale **ControllerSwitcher** e riceve un controller e una funzione che crea una nuova view a partire dal controller ricevuto. Internamente succedono 2 cose:

- il controller viene aggiunto allo stack
- viene agganciato al controller la view ottenuta dalla funzione

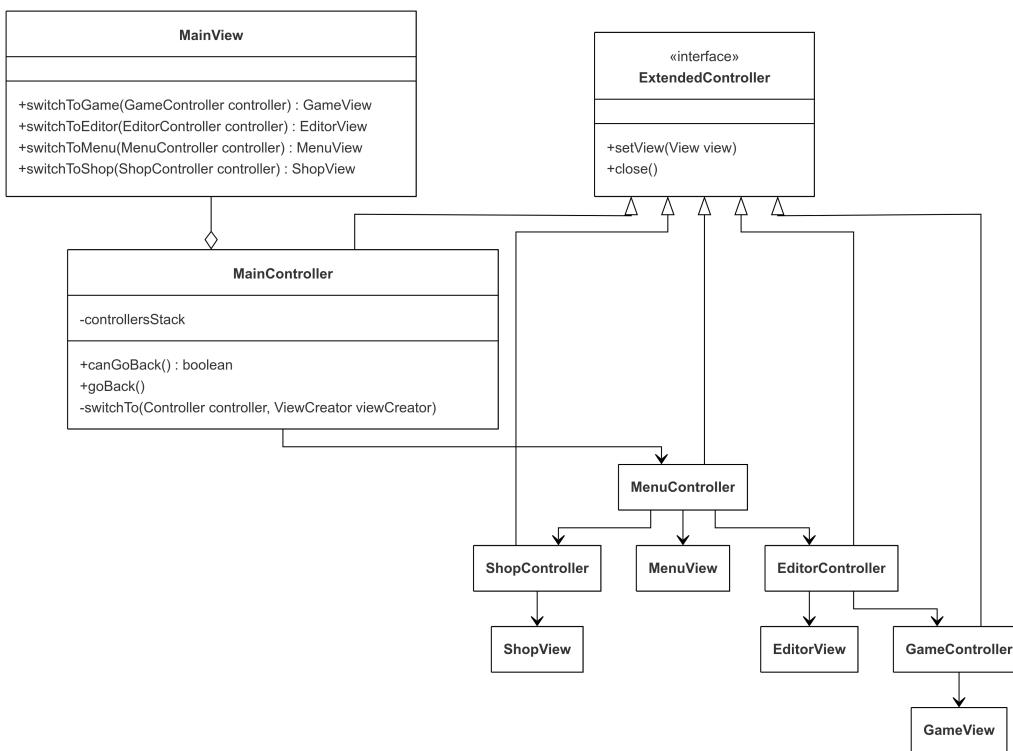


Figura 2.7: UML del Main Controller e della relazione tra i vari controller-view dell’applicazione.

### 2.2.3 Ludovico Spitaleri

**Problema** Gestione del loop di gioco.

#### Soluzione

*”Decouple the progression of game time from user input and processor speed.”<sup>2</sup>*

Il gioco si basa su un game loop che viene eseguito su un thread parallelo a quello principale così da garantire un'esecuzione non bloccante per il resto dell'applicazione. Ogni ciclo verrà calcolato il `deltaTime`, ovvero il tempo in millisecondi tra ogni frame, così da permettere a tutti gli oggetti di potersi aggiornare se stessi in maniera relativa a questo valore e di conseguenza funzionare in maniera indipendente dalle prestazioni della macchina: se trascorre un intervallo di tempo elevato tra ogni ciclo a causa di prestazioni basse, il `deltaTime` sarà elevato e di conseguenza gli aggiornamenti dipendenti da questa velocità (esempio: movimento) saranno meno ma con valori maggiori, viceversa in caso di un frame rate più elevato.

**Problema** Gestione delle interazioni tra le varie parti del gioco.

**Soluzione** Le interazioni oggetti-oggetti vengono gestite tramite una lieve rivisitazione dell'*Observer Pattern*. Il gioco, per ogni tipo di evento (di input o di gioco), presenta un gestore principale il quale fornirà la possibilità di registrare delle operazioni da eseguire all'avvenire di un evento e di lanciare un evento. Questo gestore dovrà essere diviso in interfacce diverse così da rendere visibili solo le operazioni necessarie in ogni momento: nella prima fase di setup verrà fornito a tutti sotto forma di “*subscriber*” con solo la possibilità di registrare delle reazioni ai vari eventi, mentre durante il loop verrà fornito sotto forma di “*scheduler*” che permette di registrare l'avvenimento di un certo evento. L'utente verrà provvisto prima dell'avvio di uno scheduler per gli eventi di input. Questo sistema non permette di avere eventi con uno stato interno, di conseguenza gli eventi di costruzione, aggiornamento e distruzione degli oggetti all'interno della scena sono notificati “*manualmente*” all'utente.

---

<sup>2</sup><https://gameprogrammingpatterns.com/game-loop.html#intent>

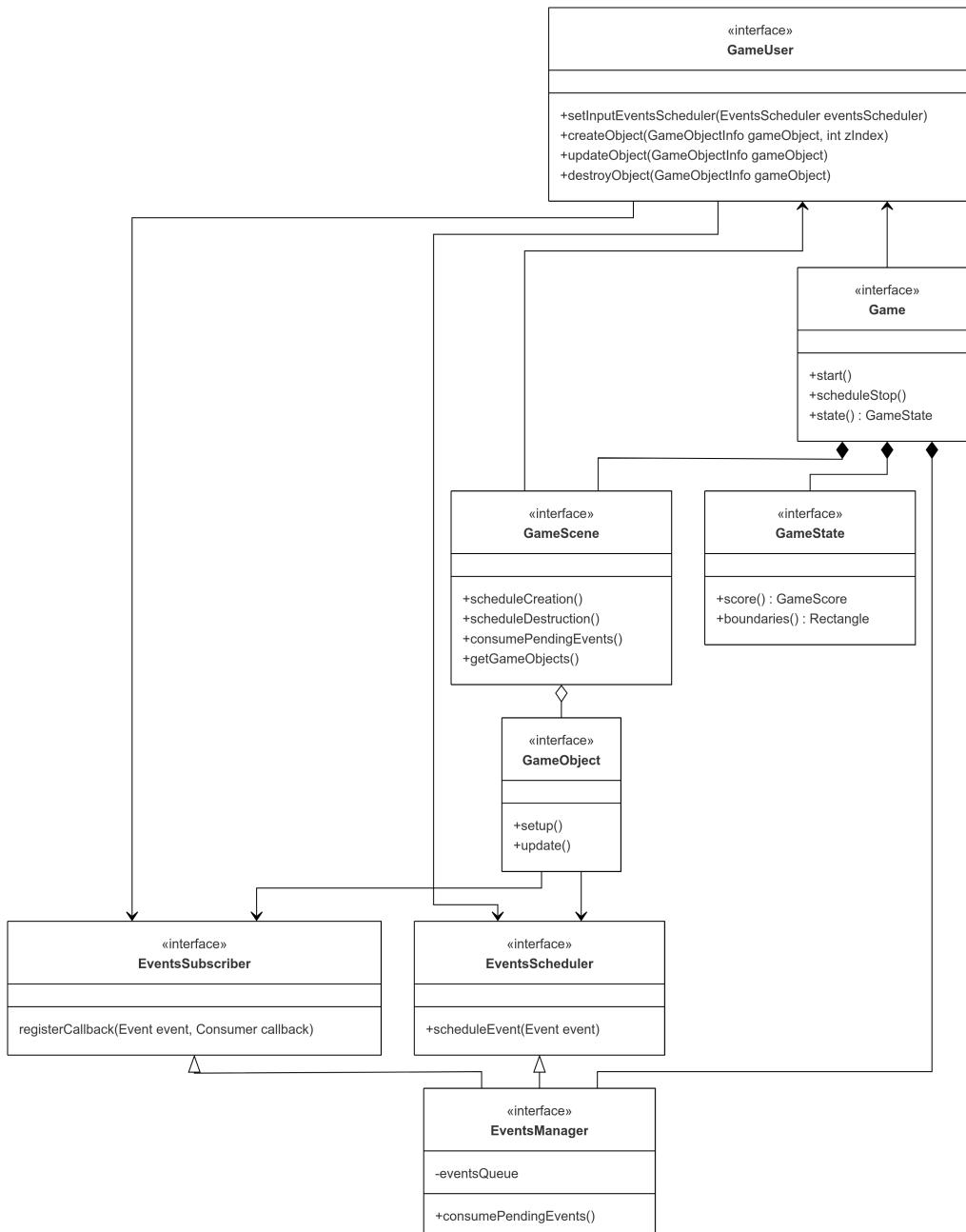


Figura 2.8: UML del game loop e della gestione delle interazioni.

**Problema** Fornire un sistema di creazione del gioco utilizzabile dall'editor.

**Soluzione** Utilizzo del *Builder Pattern*: sarà presente un `GameBuilder` che fornisce la possibilità di aggiungere gli oggetti prima della creazione effettiva del gioco, così da poterlo riempire in base agli elementi presenti sulla griglia. Tutti questi elementi da aggiungere sono in realtà registrati come eventi di creazione su una scena e processati solo al momento della chiamata al metodo `build`, alla quale verranno forniti anche i bordi totali dell'intero livello. Per evitare di ripetere logica che potrebbe essere comune a diverse implementazioni, utilizzo il *Template Method Pattern*: sarà presente un `AbstractGameBuilder` trasforma i due metodi di `GameBuilder` in due template method: l'implementazione effettiva dovrà solo decidere che tipo di scena utilizzare per l'aggiunta di oggetti e come creare il gioco senza preoccuparsi di tutte le operazioni comuni di setup.

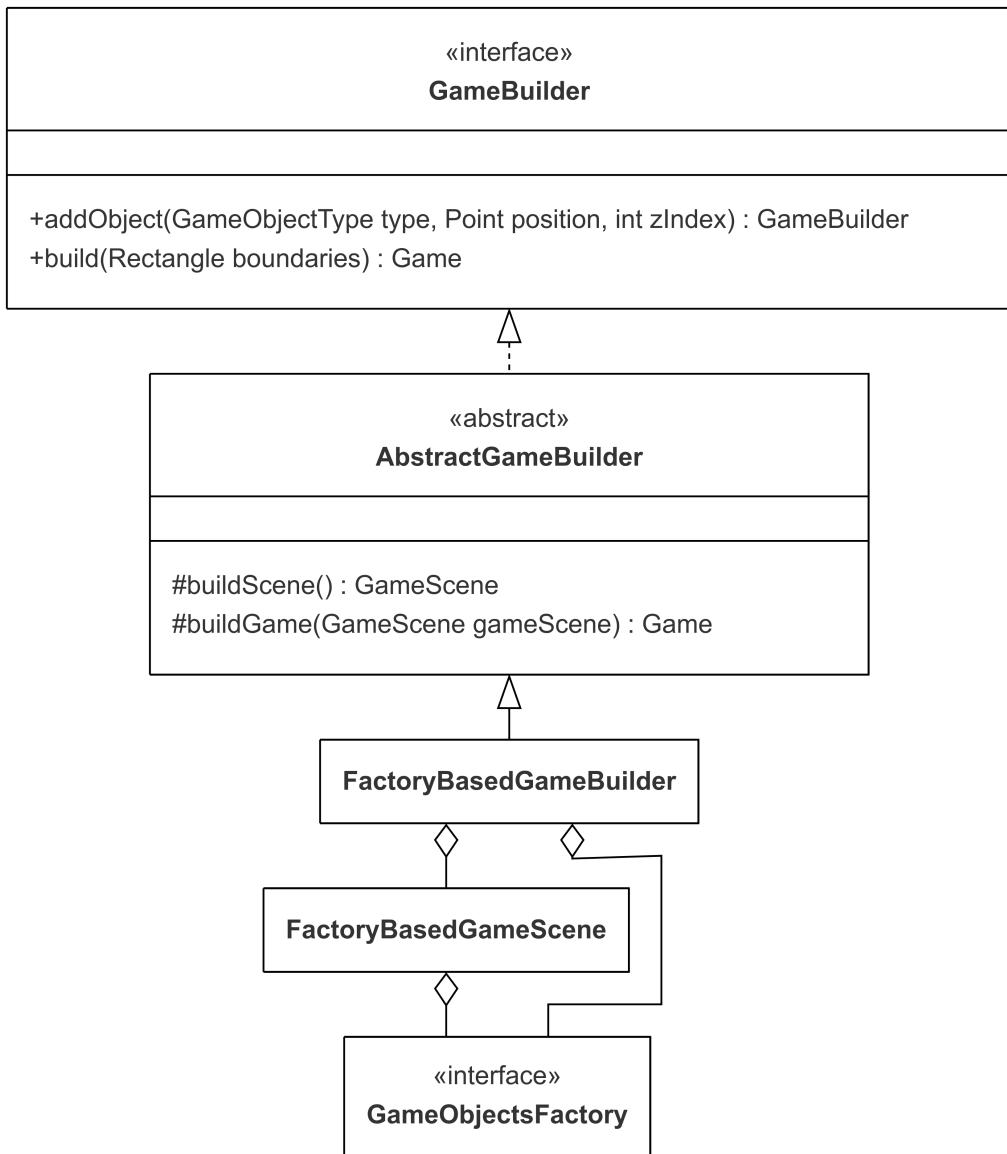


Figura 2.9: UML del game builder.

## 2.2.4 Elettra Ventura

**Creazione degli oggetti di gioco.**

**Problema** Se si seguisse l’approccio dell’ereditarietà tra classi la creazione dei game objects nella sezione giocabile dell’applicazione potrebbe portare a un’implementazione ripetitiva e pesante, con un’eccessiva duplicazione del codice. Questo poiché molti oggetti condividono comportamenti simili con lievi differenze, rendendo l’ereditarietà inefficiente. Tutto ciò compromette la scalabilità dell’applicazione e complica l’eventuale introduzione di nuovi oggetti (per esempio, nuovi tipi di nemici, o particolari elementi che decrementano il punteggio...).

**Soluzione** Abbiamo scelto un approccio a componenti di gioco, che migliora l’efficienza del processo e permette di assegnare comportamenti specifici agli oggetti senza dover creare una nuova classe per ogni possibile tipo (rappresentati tramite un `enum`).

Si è scelto di implementare una `ComponentsBasedGameObjectsFactory`, che estende una più generica `GameObjectsFactory`. La factory espone il metodo `ofType`, che accetta come argomento un tipo di `GameObjectType`. Questo metodo si occupa di chiamare la creazione di un oggetto con la dimensione corretta (adattando i suoi bordi alla grandezza di una casella della griglia) e di assegnargli i componenti appropriati, in base all’argomento della funzione. Con l’eventuale aggiunta di nuovi tipi di oggetti, la scalabilità del processo di creazione risulta semplice e flessibile, in quanto è sufficiente aggiungere al tipo di oggetto desiderato nella factory la giusta combinazione di componenti per definirne le logiche e i comportamenti desiderati.

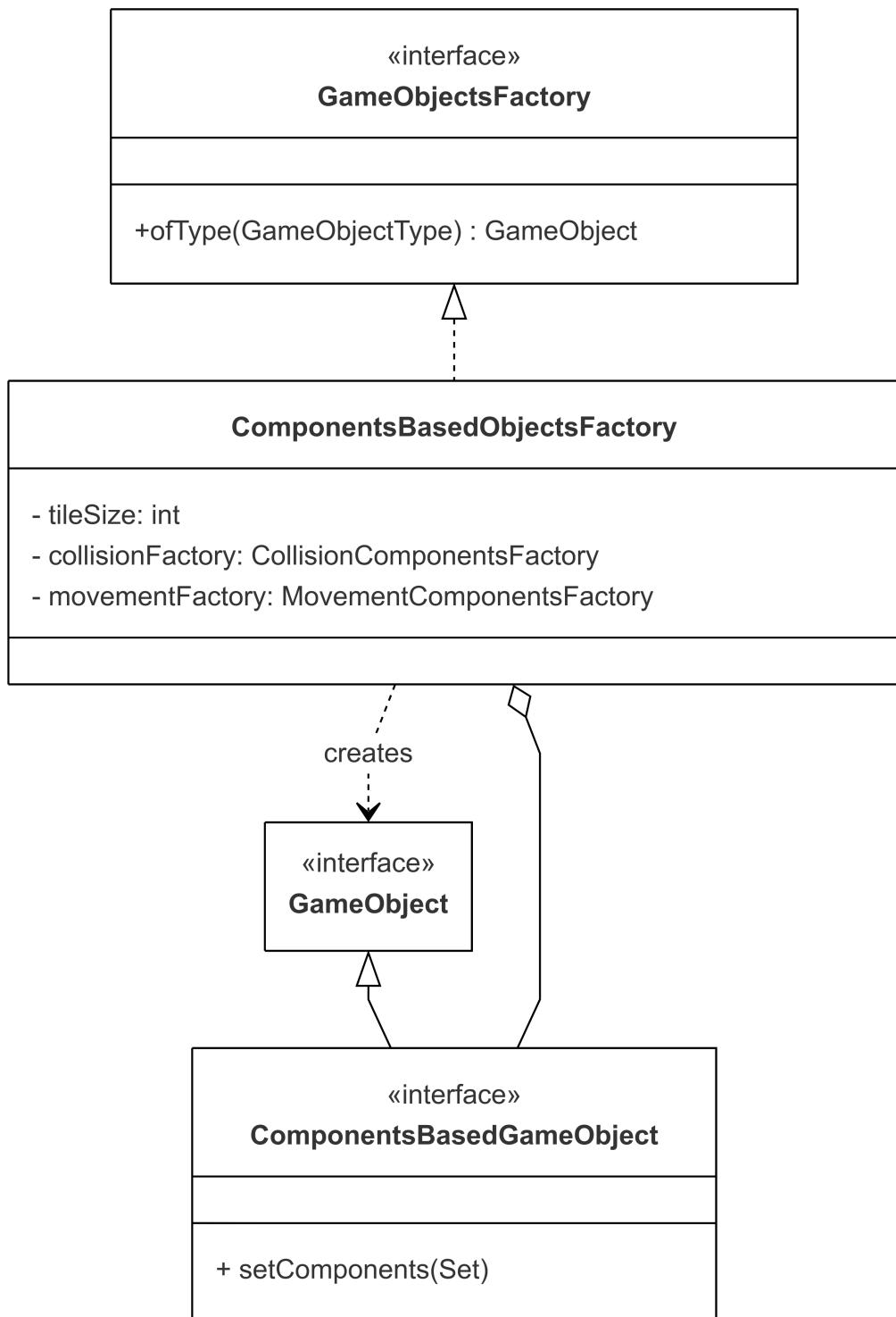


Figura 2.10: UML del *ComponentsBasedGameObjectFactory*.

## Generazione dei componenti di gioco

**Problema** La factory ha il compito di assegnare agli oggetti diversi tipi di comportamenti, anche relativi a diverse logiche di gioco, come per esempio quelli riguardanti il movimento e la gestione delle collisioni. Volevamo che questi comportamenti fossero implementati in modo facilmente estendibile, evitando ripetizioni nel codice.

**Soluzione** Un primo approccio al problema potrebbe essere la creazione di un'unica `GameComponentsFactory`, responsabile della creazione di tutti i componenti e della loro logica. Sebbene questa soluzione risolva il problema della scalabilità del progetto, avrebbe però portato a un'implementazione difficilmente mantenibile ed eccessivamente complessa. Con la prospettiva di un'espansione futura del gioco, la factory sarebbe diventata sempre più pesante. Per suddividere in maniera efficiente il compito e seguire al meglio il *Single Responsibility Principle*, l'idea della factory unica è stata rimpiazzata da due factory più piccole: La `MovementComponentsFactory`, che gestisce il movimento degli oggetti e assicura che tutti rimangano entro i confini dell'area di gioco (negando il movimento contro muri e bordi della griglia), e la `CollisionComponentsFactory`, che segnala gli eventi generati dalle interazioni tra gli oggetti. Inoltre, l'implementazione del componente di movimento che riceve input esterni è stato separato dalla factory, in modo da comunicare correttamente con gli eventi di input del gioco.

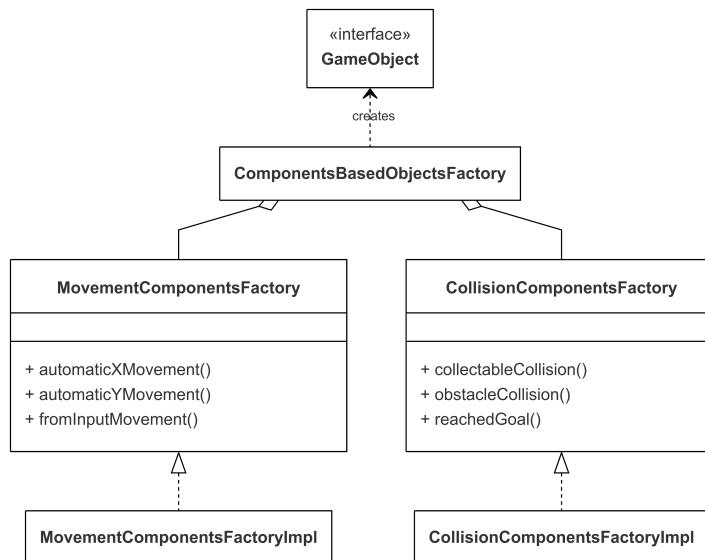


Figura 2.11: UML del `ComponentsBasedGameObjectFactory`.

**Problema** La creazione dei componenti di gioco nelle factory dedicate, deve essere abbastanza flessibile da gestire diversi tipi di comportamento senza rendere il codice rigido o difficile da estendere. Molti componenti, come per esempio quelli di movimento automatico in una sola direzione, hanno molte caratteristiche in comune e si differenziano per pochi elementi implementativi.

**Soluzione** Viene utilizzato il *pattern Strategy* all'interno delle factory per l'implementazione dei loro stessi metodi:

per esempio, nella `CollisionComponentsFactory` viene utilizzata l'interfaccia funzionale `CollisionConsumer`, su cui si appoggiano dei metodi privati più generali. I componenti utilizzano diverse strategie per reagire all'impatto con altri oggetti, che possono ora essere specificate in modo dinamico. Così facendo, nuove strategie per una futura estensione dell'applicazione potranno essere facilmente implementate senza fare modifiche pesanti al codice della factory. Infatti, basterà aggiungere un nuovo metodo che specifichi il tipo di game object che scaturisce la reazione, e la reazione stessa, sotto forma di consumer. All'interno del meccanismo di gioco, le reazioni dei componenti andranno a notificare, se necessario, degli eventi del game, come la sconfitta (se un nemico collide con il giocatore) e la vittoria (se l'area di traguardo collide con il giocatore).

### Renderizzazione della scena di gioco

**Problema** Data la quantità elevata di elementi di gioco, di cui molti potenzialmente in movimento, è necessario un modo efficiente per aggiornare la `GameView`, dato che la semplice gestione di ciascun oggetto come `JComponent` separato rallenterebbe il rendering e le prestazioni.

**Soluzione** Si utilizza il metodo `paintComponent()` all'interno di un `JPanel` personalizzato (`GamePanel`), permettendo un rendering grafico diretto degli oggetti di gioco in un'unica operazione. In questo modo, invece di aggiornare tutti i componenti Swing con `setBounds()`, il gioco disegna direttamente tutti gli oggetti sulla mappa, in modo più efficiente.

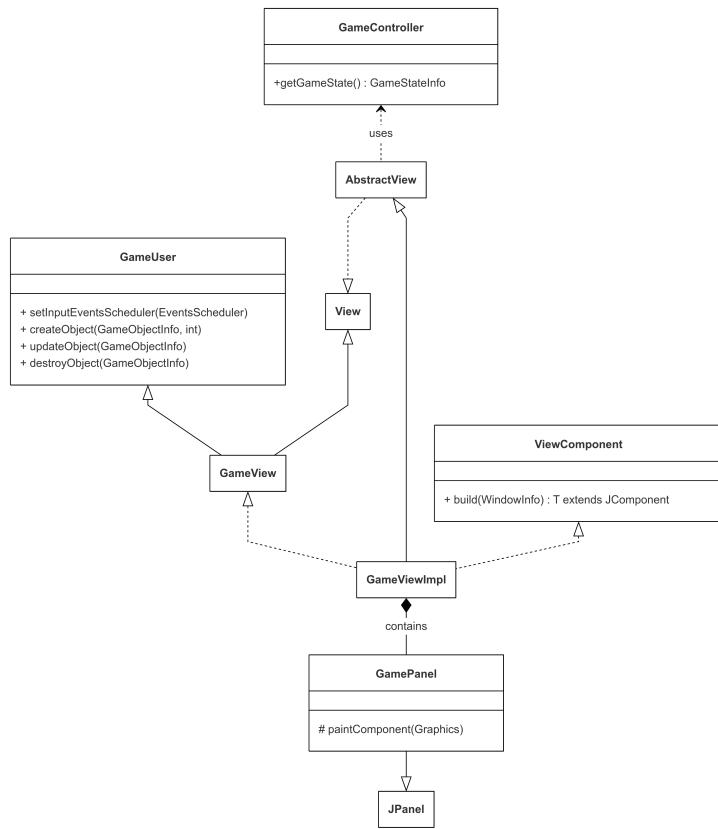


Figura 2.12: UML della *Game View*.

Il controller che si occupa della gestione della **GameView** (**GameController**) espone lo stato del gioco per rappresentare al meglio lo score dell'utente e notificare la dimensione dei limiti del gioco. In questo modo la view può assicurarsi che la dimensione della griglia di gioco (che è variabile e dipendente dalle preferenze dell'utente) sia sempre centrata e della giusta dimensione.

# Capitolo 3

## Sviluppo

### 3.1 Testing automatizzato

Per quanto riguarda l'**editor**:

- **TestConstraints**: vengono testati i limiti che l'editor pone sulla creazione del livello, come l'obbligo di adiacenza tra i blocchi piazzati, e i tetti minimi e massimi su quanti oggetti posso apparire all'interno del livello.
- **TestGrid**: viene testato il piazzamento e la rimozione di oggetti e la possibilità di salvataggio all'interno delle modalità di editing (“sandbox” e “normal”).
- **TestGrid**: viene testato il piazzamento e la rimozione di oggetti e la possibilità di salvataggio all'interno delle modalità di editing (“sandbox” e “normal”).
- **TestMetadataManager**: verifica il corretto salvataggio dei metadati.

Per quanto riguarda il gioco (nello specifico gli eventi di gioco):

- **TestEventsManager**: viene testato il corretto funzionamento dell'event manager e della corretta ricezione degli eventi in base al loro grado di priorità (che ne determina una posizione diversa all'interno della event queue).

Per quanto riguarda lo **shop**:

- **TestShop**: vengono testati i metodi dello shop che verificano che l'acquisto sia possibile e che effettuano effettivamente la transazione, facendo controlli sulle strutture dati che tengono traccia degli oggetti già posseduti e di quelli ancora bloccati.

Per quanto riguarda i **dati dell'utente**:

- **TestUserState**: viene verificato l'aggiornamento dei dati dell'utente come il credito posseduto e gli oggetti a propria disposizione e come questi vengono modificati.

Per quanto riguarda il package **Common**:

- **TestCartesianEntity**: viene testato che due entità cartesiane poste nella stessa posizione siano equivalenti. Vengono inoltre verificate le operazioni basiliari di somma, sottrazione e moltiplicazione tra entità e il metodo per invertire le coordinate
- **TestRectangle**: viene testata la creazione del rettangolo data la disposizione di tutti i suoi vertici, la creazione di un rettangolo centrato e il metodo che stabilisce se un altro rettangolo è contenuto o meno all'interno di se stesso.
- **TestOptionals**: viene testato il corretto funzionamento della funzione che lancia `IllegalStateException` se l'optional è vuoto.

## 3.2 Note di sviluppo

### 3.2.1 Gioele Foschi

#### Utilizzo di librerie di terze parti (Gson)

Permalink: <https://github.com/4realgames/OOP24-arcaym/blob/60b3ce9be9c0b7d094786e34319b0fb26cc1f7c7/src/main/java/arcaym/common/utils/file/FileUtils.java#L134>

#### Implementazione dell'algoritmo [HK76] tramite librerie di terze parti (Guava)

Utilizzata in diversi casi, utilizzo più avanzato: <https://github.com/4realgames/OOP24-arcaym/blob/60b3ce9be9c0b7d094786e34319b0fb26cc1f7c7/src/main/java/arcaym/model/editor/constraints/MapConstraintFactoryImpl.java#L22>

#### Utilizzo di Stream

Usati in numerosi punti: Permalink: <https://github.com/4realgames/OOP24-arcaym/blob/60b3ce9be9c0b7d094786e34319b0fb26cc1f7c7/src/main/java/arcaym/model/editor/grid/GridImpl.java#L229>

## **Utilizzo di Lambda, e Method reference**

Usati in vari punti: Permalink: <https://github.com/4realgames/OOP24-arcaym/blob/60b3ce9be9c0b7d094786e34319b0fb26cc1f7c7/src/main/java/arcaym/model/editor/grid/GridImpl.java#L164>

## **Lettura di più file all'interno di una cartella**

Permalink al progetto: <https://github.com/4realgames/OOP24-arcaym/blob/180b1b3bb9af5b833e3607d7f414da018d877d6b/src/main/java/arcaym/controller/editor/saves/MetadataManagerImpl.java#L44>

Preso da: <https://github.com/unibo-oop-projects/OOP23-PietroPasiniPietroPasini4st-Giacomoarenti-DavideFiocchi-GiacomoBoschi-PietroPasini-Davide-TD/blob/55bd37ec2520f80833f3ddd5e7e9ce177219be6f/src/main/java/it/unibo/towerdefense/model/saves/SavesImpl.java#L59>

### **3.2.2 Nicholas Magi**

#### **Utilizzo di librerie esterne - Guava**

Permalink: <https://github.com/4realgames/OOP24-arcaym/blob/180b1b3bb9af5b833e3607d7f414da018d877d6b/src/main/java/arcaym/model/user/UserStateInfo.java#L38>

#### **Utilizzo di librerie esterne - Gson**

Permalink: <https://github.com/4realgames/OOP24-arcaym/blob/180b1b3bb9af5b833e3607d7f414da018d877d6b/src/main/java/arcaym/controller/user/UserStateSerializerJSON.java#L36>

#### **Utilizzo di librerie esterne - SLF4J**

Permalink: <https://github.com/4realgames/OOP24-arcaym/blob/180b1b3bb9af5b833e3607d7f414da018d877d6b/src/main/java/arcaym/controller/user/UserStateSerializerJSON.java#L24>

#### **Utilizzo di stream con lambda expressions e method reference**

Utilizzati in qualche occasione. Di seguito un esempio.

Permalink: <https://github.com/4realgames/OOP24-arcaym/blob/180b1>

b3bb9af5b833e3607d7f414da018d877d6b/src/main/java/arcaym/controller/app/MainControllerImpl.java#L38

### Utilizzo di Optional

Utilizzati in qualche occasione, specialmente in `UserStateSerializerJSON`. Di seguito un esempio.

Permalink: <https://github.com/4realgames/OOP24-arcaym/blob/25e8ea02e91a1e2dd0a26c3aeebdf76eea33fdbb/src/main/java/arcaym/controller/user/UserStateSerializerJSON.java#L43>

### 3.2.3 Ludovico Spitaleri

#### Uso di reflection

Permalink: <https://github.com/4realgames/OOP24-arcaym/blob/1683c9de008220b463d95ab05748ad0a11f10ecb/src/main/java/arcaym/common/utils/representation/StringRepresentation.java#L48>

#### Uso di Optional

Usati in vari punti. Di seguito riportato un esempio.

Permalink: <https://github.com/4realgames/OOP24-arcaym/blob/1683c9de008220b463d95ab05748ad0a11f10ecb/src/main/java/arcaym/common/utils/Optionals.java#L23>

#### Uso di wildcard

Permalink: <https://github.com/4realgames/OOP24-arcaym/blob/1683c9de008220b463d95ab05748ad0a11f10ecb/src/main/java/arcaym/view/core/ParentComponent.java#L54>

#### Uso di generici

Usati in vari punti. Di seguito riportato un esempio.

Permalink: <https://github.com/4realgames/OOP24-arcaym/blob/1683c9de008220b463d95ab05748ad0a11f10ecb/src/main/java/arcaym/model/game/core/events/EventsManager.java#L8>

## **Uso del Thread static builder**

Permalink: <https://github.com/4realgames/OOP24-arcaym/blob/1683c9de008220b463d95ab05748ad0a11f10ecb/src/main/java/arcaym/model/game/core/engine/SingleThreadedGame.java#L40>

## **Uso di lambda tramite method reference**

Usati in vari punti. Di seguito riportato un esempio.

Permalink: <https://github.com/4realgames/OOP24-arcaym/blob/1683c9de008220b463d95ab05748ad0a11f10ecb/src/main/java/arcaym/model/game/core/engine/AbstractGameBuilder.java#L65>

### **3.2.4 Elettra Ventura**

#### **Uso di Stream**

Usate pervasivamente. Di seguito un esempio.

<https://github.com/4realgames/OOP24-arcaym/blob/180b1b3bb9af5b833e3607d7f414da018d877d6b/src/main/java/arcaym/model/game/components/CollisionUtils.java#L36>

#### **Uso di Optionals**

<https://github.com/4realgames/OOP24-arcaym/blob/1683c9de008220b463d95ab05748ad0a11f10ecb/src/main/java/arcaym/view/game/GameViewImpl.java#L53C1-L54C1>

#### **Uso di lambda expressions**

Usate pervasivamente. Di seguito un esempio.

Permalink: <https://github.com/4realgames/OOP24-arcaym/blob/180b1b3bb9af5b833e3607d7f414da018d877d6b/src/main/java/arcaym/model/game/components/InputMovementComponent.java#L56>

#### **Utilizzo di risorse esterne**

Permalink: <https://github.com/4realgames/OOP24-arcaym/blob/180b1b3bb9af5b833e3607d7f414da018d877d6b/src/main/java/arcaym/common/geometry/Rectangle.java#L101>

Sorgente: AABB (AXIS ALIGNED BOUNDING BOX) algortihm for 2D collisions - [https://developer.mozilla.org/en-US/docs/Games/Techniques/2D\\_collision\\_detection](https://developer.mozilla.org/en-US/docs/Games/Techniques/2D_collision_detection)

# Capitolo 4

## Commenti finali

### 4.1 Autovalutazione e lavori futuri

#### 4.1.1 Gioele Foschi

Durante lo sviluppo della mia sezione, ritengo di aver prodotto codice di qualità medio-alta. Quando una soluzione inizialmente non mi soddisfaceva, riuscivo a riprogettarla in modo efficace, migliorando la qualità finale del lavoro.

Un aspetto in cui ho incontrato difficoltà è stato la coordinazione con il team, in particolare nella divisione del progetto in parti ben definite. Questo problema è stato in parte causato da una valutazione iniziale non ottimale del progetto, che ha portato alla definizione di alcuni dettagli implementativi che, durante lo sviluppo dell'applicativo, si sono rivelati incoerenti.

È stato dunque difficile trovare soluzioni adeguate quando le scelte iniziali si sono rivelate problematiche.

Se dovessi continuare a lavorare su questo progetto in futuro, mi concentrerei sull'unificazione della gestione dei file di gioco, sia testuali che binari, all'interno di un unico sistema. Inoltre, rivedrei la struttura del model per cercare di semplificare il salvataggio dei dati, centralizzandolo completamente lato controller. Sebbene la struttura attuale lo permetta, il processo risulta poco intuitivo e complesso da gestire.

In generale, sono molto soddisfatto del lavoro svolto.

#### 4.1.2 Nicholas Magi

Sento che questo progetto è stato un grande stimolo per migliorare le mie abilità di progettazione e di implementazione del software. Sono abbastanza soddisfatto del mio lavoro che, personalmente, vedo come una valida aggiunta

al prodotto finale. Il mio codice è stato pensato e ripensato molteplici volte, cercando sempre di rispettare il più possibile i principi di buona progettazione di un programma. La classe di cui vado più fiero è `UserState`, per la quale ho dovuto rivedere la progettazione e il funzionamento in diverse occasioni. Ha raggiunto una fase in effettuare un'aggiunta di informazioni dovrebbe essere relativamente semplice e lineare, e non dovrebbe rompere in alcun modo le altre parti di programma. Pensando ad una manutenzione o aggiunta futura, dedicherei volentieri più tempo a migliorare l'esperienza di gioco dell'utente, introducendo - per esempio - un meccanismo di *Achievement* per modellare uno stato di completamento del gioco. Cercando di fare un'analisi critica su me stesso e sull'intero progetto, osservo che la progettazione iniziale è forse stata affrettata o non subito ben pensata. La mia porzione di lavoro è mutata in corso d'opera, poiché quello che inizialmente pensavo fosse un compito di dimensioni simili a quelle dei compiti dei miei compagni non si è rivelato tale. Tutto sommato, un'esperienza impegnativa ma soddisfacente.

#### 4.1.3 Ludovico Spitaleri

Le performance del gioco sono molto buone e la struttura permette una facile espandibilità. La difficoltà più grande è stata quella di dividere le varie interfacce per fornire solo il minimo indispensabile ad ogni parte del gioco. Un futuro aggiornamento potrebbe includere una ristrutturazione del sistema di eventi per permettere un utilizzo maggiore anche in altri contesti, come quello della scena.

#### 4.1.4 Elettra Ventura

Nel progetto, il mio ruolo principale è stato lo sviluppo dei componenti di gioco relativi alla fisica degli oggetti, in particolare la gestione delle collisioni e del movimento. Non avendo molta esperienza con progetti di programmazione di questa complessità, ho inizialmente incontrato alcune difficoltà nel trovare le strategie più efficaci per la mia parte e per la comunicazione con le altre. Tuttavia, grazie all'ambiente stimolante trovato nel gruppo, posso dire di avere acquisito nuove tecniche e approcci che mi hanno spinto a migliorarmi costantemente.

Sono complessivamente molto soddisfatta del lavoro svolto e dei progressi fatti, ma sono tuttavia consapevole del fatto che il prodotto del mio lavoro non è perfetto, e ciò mi ha portata più volte a rivedere e correggere frammenti del mio lavoro per migliorarne le prestazioni e l'organizzazione. Detto ciò, posso dire che questa esperienza è stata estremamente formativa e mi ha dato l'opportunità di mettermi alla prova.

Se il progetto dovesse essere portato avanti, mi piacerebbe testare l'effettiva scalabilità dell'applicazione con l'introduzione di nuovi oggetti di gioco e nuovi componenti, come ad esempio nemici con diversi sistemi di movimento, per verificare la solidità dell'architettura. Inoltre, con più esperienza, potrei rivedere alcune scelte di progettazione per ottimizzare ulteriormente il codice e apportare rifinimenti sulle logiche implementate, con una maggiore attenzione alla corretta posizione dei game objects in prossimità di un oggetto invalicabile (muri e bordi del gioco).

## **4.2 Difficoltà incontrate e commenti per i do-centi**

...

# Appendice A

## Guida utente

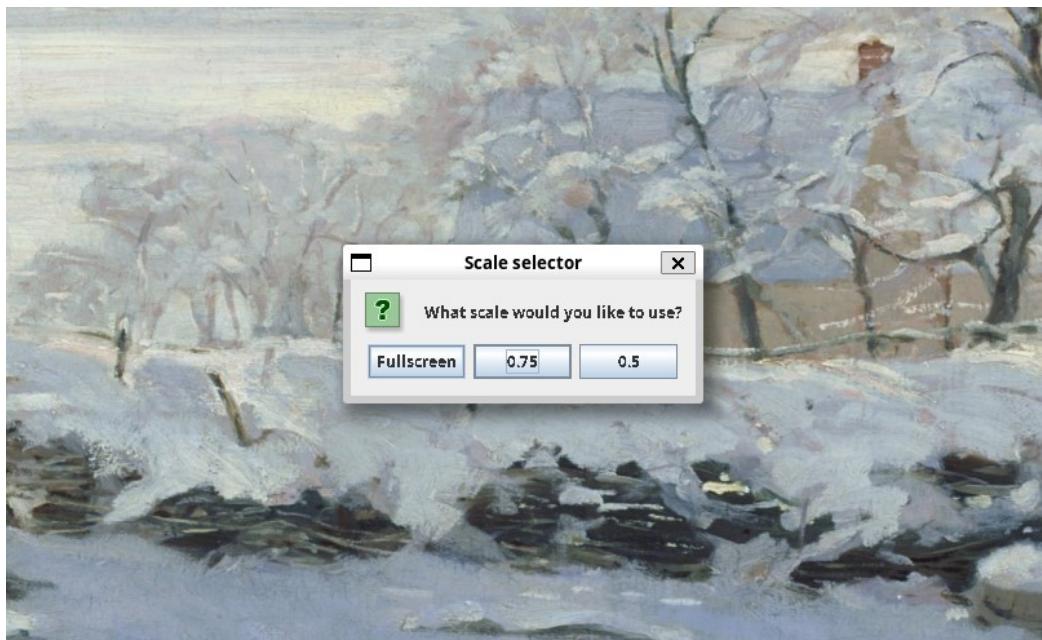


Figura A.1: Selezione della dimensione dello schermo

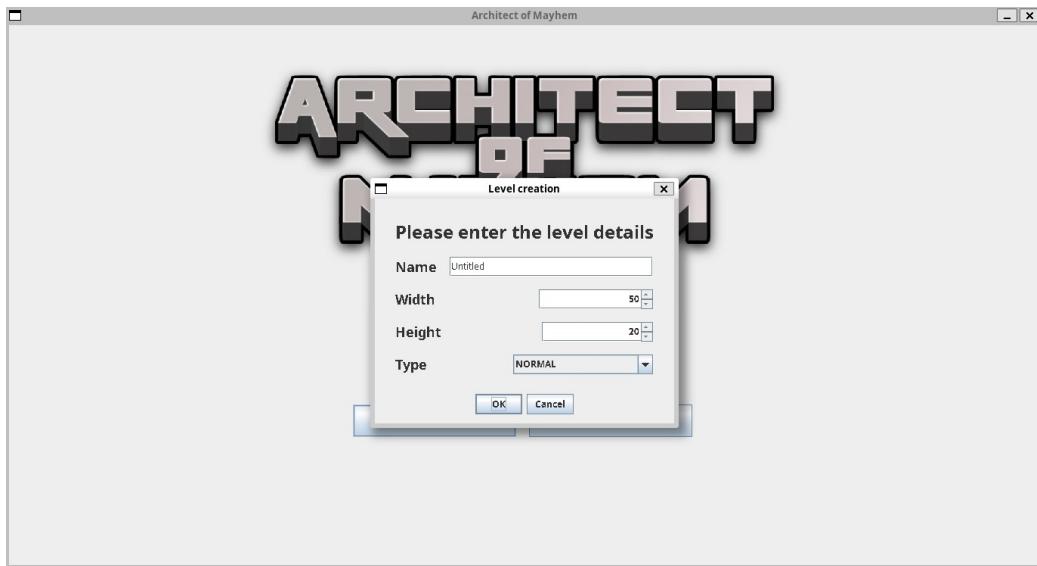


Figura A.2: Creazione del livello

## A.1 Editor

All'interno dell'editor è possibile selezionare un oggetto dal menu laterale e piazzarlo a proprio piacimento sulla griglia a disposizione. Nella parte bassa dell'applicazione vengono segnalati eventuali problemi con lo stato di creazione del livello.

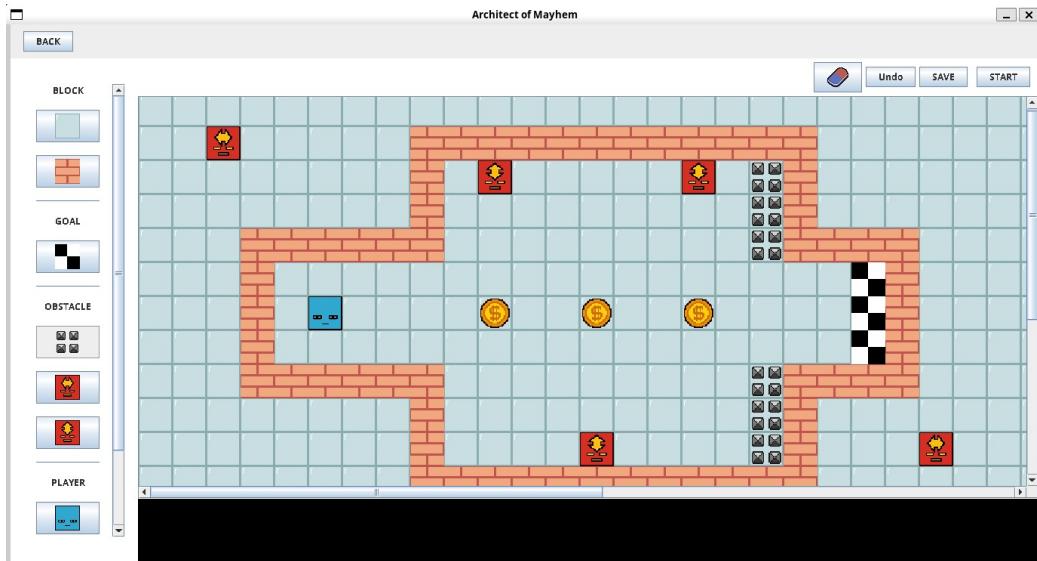


Figura A.3: Schermata di editor

## A.2 Gameplay

Una volta avviato il gameplay, è possibile controllare i movimenti del player con i tasti WASD:

- W: UP
- A: LEFT
- S: DOWN
- D: RIGHT

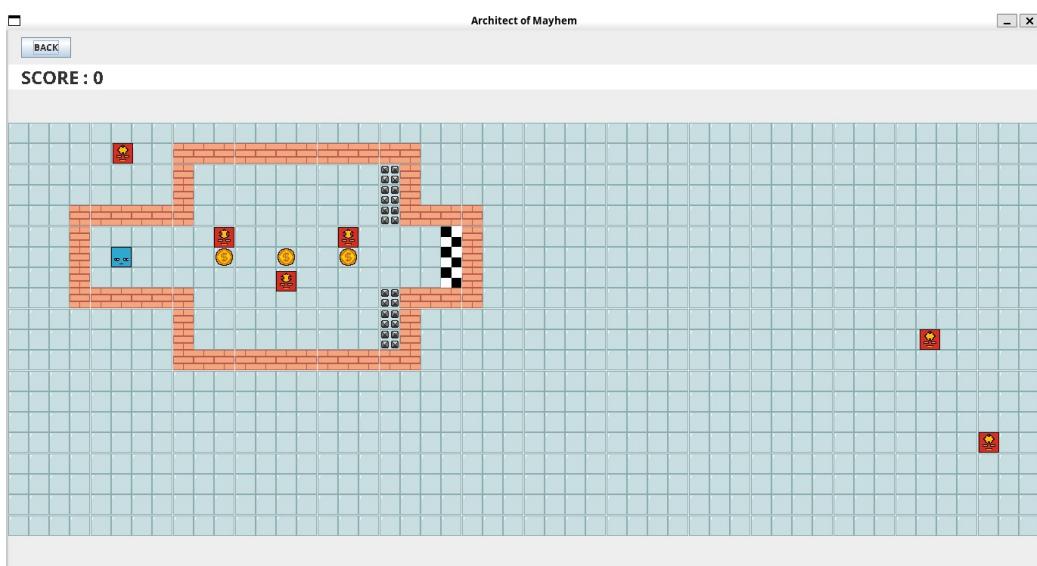


Figura A.4: Schermata di gameplay

# **Appendice B**

## **Esercitazioni di laboratorio**

### **B.1 gioele.foschi@studio.unibo.it**

- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=179154#p247997>
- Laboratorio 11: <https://virtuale.unibo.it/mod/forum/discuss.php?d=181206#p250617>

### **B.2 nicholas.magi@studio.unibo.it**

- Laboratorio 08: <https://virtuale.unibo.it/mod/forum/discuss.php?d=178723#p247235>
- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=179154#p247850>
- Laboratorio 11: <https://virtuale.unibo.it/mod/forum/discuss.php?d=181206#p250319>

### **B.3 ludovico.spitaleri@studio.unibo.it**

- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=22222#p222222>
- Laboratorio 10: <https://virtuale.unibo.it/mod/forum/discuss.php?d=99999#p999999>

## B.4 elettra.ventura@studio.unibo.it

- Laboratorio 07: <https://virtuale.unibo.it/mod/forum/discuss.php?d=177162#p246082>
- Laboratorio 08: <https://virtuale.unibo.it/mod/forum/discuss.php?d=178723#p247277>
- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=179154#p248309>
- Laboratorio 11: <https://virtuale.unibo.it/mod/forum/discuss.php?d=181206#p250797>

# Bibliografia

- [HK76] J. Hoshen and R. Kopelman. Percolation and cluster distribution. i. cluster multiple labeling technique and critical concentration algorithm. *Phys. Rev. B*, 14:3438–3445, Oct 1976.