

# Javamant

## Relazione per il progetto del corso OOP

Filippo Gaggi  
Andrea La Tosa  
Emir Wanes Aouioua

12 settembre 2025

# 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>8</b>
2.1 Architettura . . . . .	8
2.2 Design dettagliato . . . . .	11
2.2.1 Filippo Gaggi . . . . .	11
2.2.2 Emir Wanes Aouioua . . . . .	14
2.2.3 Andrea La Tosa . . . . .	21
<b>3 Sviluppo</b>	<b>25</b>
3.1 Testing automatizzato . . . . .	25
3.2 Note di sviluppo . . . . .	26
3.2.1 Filippo Gaggi . . . . .	26
3.2.2 Emir Wanes Aouioua . . . . .	27
3.2.3 Andrea La Tosa . . . . .	28
<b>4 Commenti finali</b>	<b>29</b>
4.1 Autovalutazione e lavori futuri . . . . .	29
4.1.1 Filippo Gaggi . . . . .	29
4.1.2 Emir Wanes Aouioua . . . . .	29
4.1.3 Andrea La Tosa . . . . .	30
4.2 Difficoltà incontrate e commenti per i docenti . . . . .	30
4.2.1 Emir Wanes Aouioua . . . . .	30
<b>A Guida utente</b>	<b>31</b>
A.1 Home . . . . .	31
A.2 Impostazioni . . . . .	31
A.3 Impostazioni non corrette . . . . .	32
A.4 Gameplay . . . . .	33

A.5	Leaderboard . . . . .	37
<b>B</b>	<b>Esercitazioni di laboratorio</b>	<b>38</b>
B.1	Emir Wanes Aouioua . . . . .	38

# Capitolo 1

## Analisi

### 1.1 Descrizione e requisiti

Il progetto ha come scopo la realizzazione di una trasposizione digitale del gioco da tavolo DIAMANT. Il gioco simula l'esplorazione di grotte, in cui i giocatori (dai 3 agli 8) cercano di accumulare il maggior numero possibile di gemme. Alla fine di una serie di round il vincitore sarà colui che riuscirà a raccogliere il maggior numero di gemme. L'esplorazione di una grotta è rappresentata da un round. Il percorso di esplorazione di una grotta rappresenta le carte pescate da un mazzo durante il round, le quali possono essere di tre tipi:

- **Carte tesoro**, 15 nel mazzo standard. Rappresentano una quantità di gemme da dividere tra i giocatori in esplorazione nella grotta. La divisione delle gemme delle carte tesoro è equa tra i giocatori: il resto della divisione rimane all'interno del percorso di esplorazione.
- **Carte trappola**, 15 nel mazzo standard. Rappresentano ostacoli per i giocatori. Pescare due carte trappola dello stesso tipo (ad esempio, due trappole lava o due trappole macigno) fa terminare il round forzatamente e fa perdere tutte le gemme guadagnate nell'esplorazione della grotta ai giocatori che non hanno abbandonato quest'ultima.
- **Carte reliquia**, 5 nel mazzo standard. Le reliquie hanno un valore in gemme e, a differenza delle carte tesoro, il valore in gemme di quest'ultime non viene distribuito tra i giocatori: le gemme relative alle carte reliquia vengono assegnate ad un solo giocatore, nel caso in cui esso decida di abbandonare l'esplorazione della grotta da solo.

Ogni giocatore possiede una sacca per raccogliere le gemme trovate durante l'esplorazione della grotta ed un forziere dove conservare le gemme alla

fine dell'esplorazione di quest'ultima.

L'esplorazione di una grotta si svolge in turni. Ogni turno è suddiviso in due fasi:

- **Fase di estrazione:** il giocatore del turno pesca una carta dal mazzo, la aggiunge al percorso di esplorazione e distribuisce eventuali gemme tra i giocatori presenti nella grotta; in caso di trappola doppia il round termina.
- **Fase di decisione:** tutti i giocatori all'interno della grotta decidono se continuare ad esplorare quest'ultima o se uscire. I giocatori uscenti dividono le gemme rimaste nel percorso di esplorazione e le depositano, assieme alla propria sacca, all'interno del rispettivo forziere.

Il gioco si fonda su un fattore di rischio legato alla decisione sul proseguimento dell'esplorazione:

- **Continuare l'esplorazione** comporta possibilmente un maggior numero di gemme guadagnato ma aumenta il rischio che due trappole identiche vengano pescate: in tal caso tutte le gemme dei giocatori rimasti in esplorazione della grotta non vengono aggiunte ai rispettivi forzieri.
- **Abbandonare l'esplorazione** preclude il giocatore dal trovare altre gemme ma gli permette di mettere al sicuro le gemme della sua sacca all'interno del proprio forziere, oltre a quelle delle carte reliquia se è abbastanza fortunato da abbandonare il round da solo durante il turno.

La trasposizione digitale del gioco prevede inoltre la possibilità di applicare degli effetti speciali al round che modificano il modo in cui l'esplorazione di una grotta termina (nella versione standard attraverso due carte trappola identiche pescate) e delle variazioni al numero di gemme guadagnate dai giocatori.

## Requisiti funzionali

- Il sistema deve gestire un mazzo di 35 carte divise in tre categorie (tesoro, trappola, reliquia).
- L'applicazione deve supportare partite da 3 ad 8 giocatori con la possibilità di giocatori comandati dal software.
- I turni dovranno essere composti da due fasi in sequenza: l'estrazione della carta e la decisione da parte di tutti i giocatori nella grotta sul continuare l'esplorazione.

- Dovrà essere possibile estrarre una carta e la sua visualizzazione nel percorso di esplorazione.
- L'applicazione deve calcolare la divisione equa delle gemme tra i giocatori attivi, tenendo in considerazione le gemme già rimaste sul percorso ed eventuali modificatori applicati a quest'ultime.
- La partita deve terminare automaticamente se l'effetto speciale del round ne decreta il termine (con le regole standard se due carte trappola uguali vengono pescate).
- Il sistema deve tracciare le gemme di ogni giocatore, sia nella sacca che nel forziere; il sistema deve stilare una classifica dei giocatori al termine di tutti i round che sia ordinata in base al numero di gemme nei forzieri.
- L'applicazione deve gestire l'assegnazione delle relique nel caso un solo giocatore esca dalla partita.

### **Requisiti non funzionali**

- L'architettura software deve essere progettata in modo modulare per permettere una facile estendibilità delle carte e degli effetti del round.
- L'interfaccia grafica dovrà essere chiara ed intuitiva, permettere la visualizzazione dei turni, delle carte pescate e dei punteggi.
- I bot devono utilizzare strategie logiche in base allo stato della partita.
- Il sistema deve dare la possibilità di configurare la partita: decidere il numero di round, gli effetti speciali, il deck usato e la presenza o meno di bot.

## **1.2 Modello del Dominio**

L'applicazione Javamant modella l'esplorazione di grotte, ognuna rappresentata da un round. Ogni partita coinvolge da 3 a 8 giocatori, che possono essere controllati da umani o dal software. Ogni round utilizza un mazzo formato da più carte che progressivamente formeranno il percorso di esplorazione della grotta. Le carte pescate dal mazzo si dividono in più tipi: le carte tesoro forniscono gemme da distribuire equamente tra i giocatori, le carte reliquia danno gemme aggiuntive se un solo giocatore abbandona l'esplorazione alla fine di un turno e le carte trappola, presenti in più tipologie,

rappresentano pericoli che possono, se l'effetto del round lo decide, terminare il round. In un round si giocano più turni, ognuno composto da due fasi: nella fase di estrazione il giocatore del turno pesca una carta che viene aggiunta al percorso dell'esplorazione del round. Nella fase di decisione i giocatori che esplorano la grotta scelgono se continuare l'esplorazione. Tutti i giocatori hanno una sacca temporanea per raccogliere le gemme durante un round ed un forziere per mantenere le proprie gemme tra un round e l'altro. Ad ogni round viene associato un effetto speciale che ne determina la condizione di fine e un modificatore applicato alle gemme. 1.1

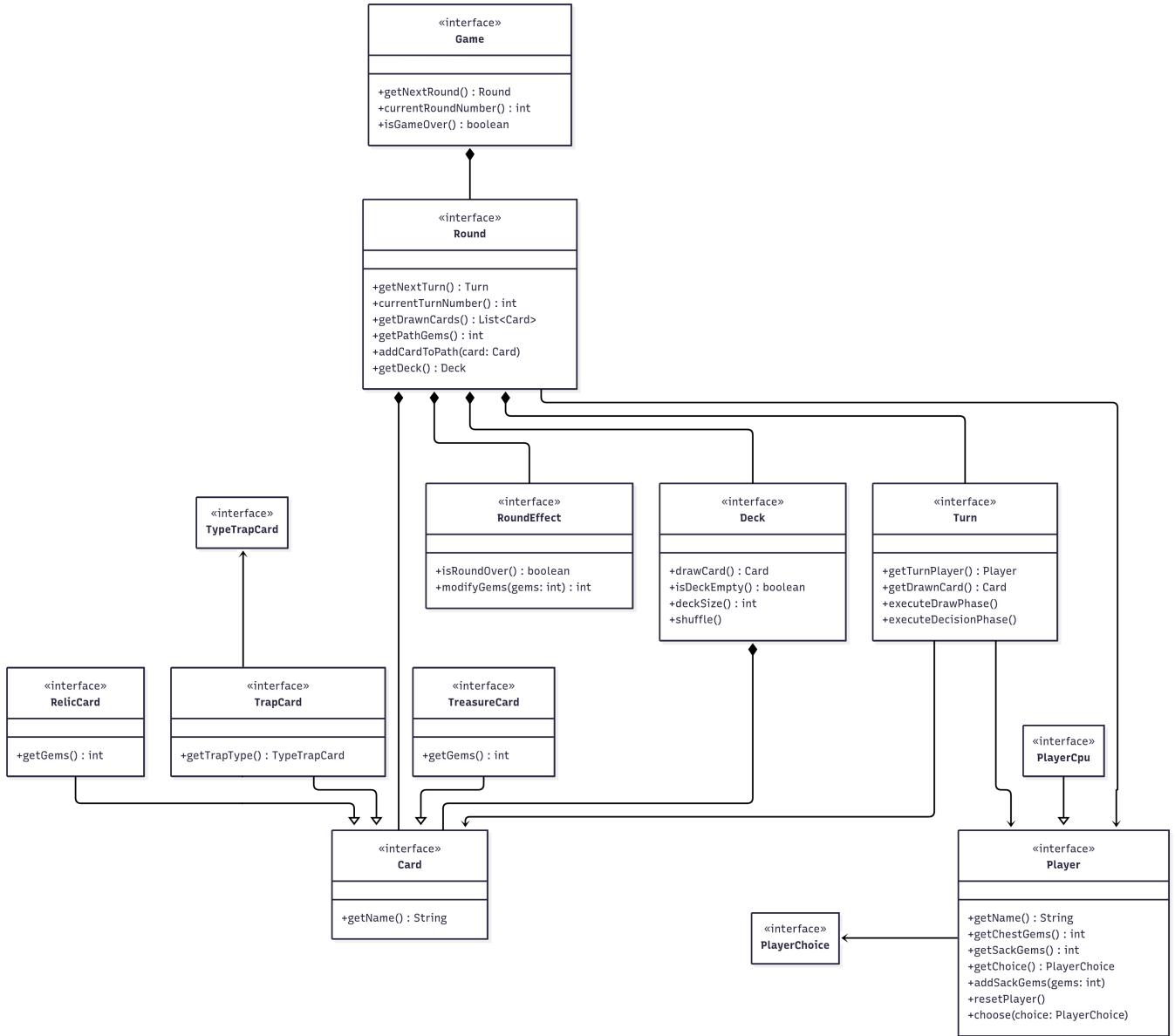


Figura 1.1: Schema UML rappresentante il dominio di Javamant.

# Capitolo 2

## Design

### 2.1 Architettura

L'architettura di Javamant adotta il pattern architettonico Model View Controller (MVC) per dare una separazione ai ruoli dei componenti principali, minimizzando le dipendenze. In particolare:

- **Model:** definito dall'interfaccia *Game*; rappresenta il punto di accesso al dominio dell'applicazione.
- **View:** viene definita dalle interfacce *Window*, rappresentante la finestra principale di visualizzazione, *Page* rappresentante le singole pagine che possono essere rappresentate su una Window e *ControllerAwarePage*, specializzazione astratta di *Page*, rappresenta una pagina interagibile (che adotta un controller).
- **Controller:** definita da *MainController*, il quale avvia l'applicazione, *PageController* che definisce l'interazione su una pagina specifica e *GameAwarePageController*, specializzazione di *PageController* con accesso ad informazioni sul model (*Game*). I controller per le pagine gestiscono la navigazione tra queste ultime tramite *PageNavigator*.

Le interazioni principali di questa architettura sono:

- *MainController* gestisce tutte le dipendenze tra model, view e controller. Inizializza la *Window*, il *PageNavigator*, le varie *Page* e rispettivi *PageController*.
- *GameAwarePageController* estende *PageController*; interagisce con *Game* per recuperare informazioni sullo stato della partita.

- Ogni PageController controlla un'unica ControllerAwarePage (pagina interagibile) e utilizza il PageNavigator per cambiare visualizzazione. Allo stesso modo ogni ControllerAwarePage utilizza un PageController per associare gli eventi sulla pagina.
- Window ha il compito di essere un contenitore per Page e si occupa di mostrarle all'utente.
- PageNavigator gestisce la transizione da una pagina all'altra.

L'architettura è stata progettata in modo da poter sostituire in modo semplice la view: per poter sostituire l'intera componente grafica è sufficiente dare nuove implementazioni di *Window* e *Page* (o estensioni di *ControllerAwarePage* per pagine interagibili), infatti i controller dipendono unicamente da queste interfacce ed il model non mantiene alcun riferimento ad elementi grafici.

Una versione schematizzata dell'architettura adoperata 2.1:

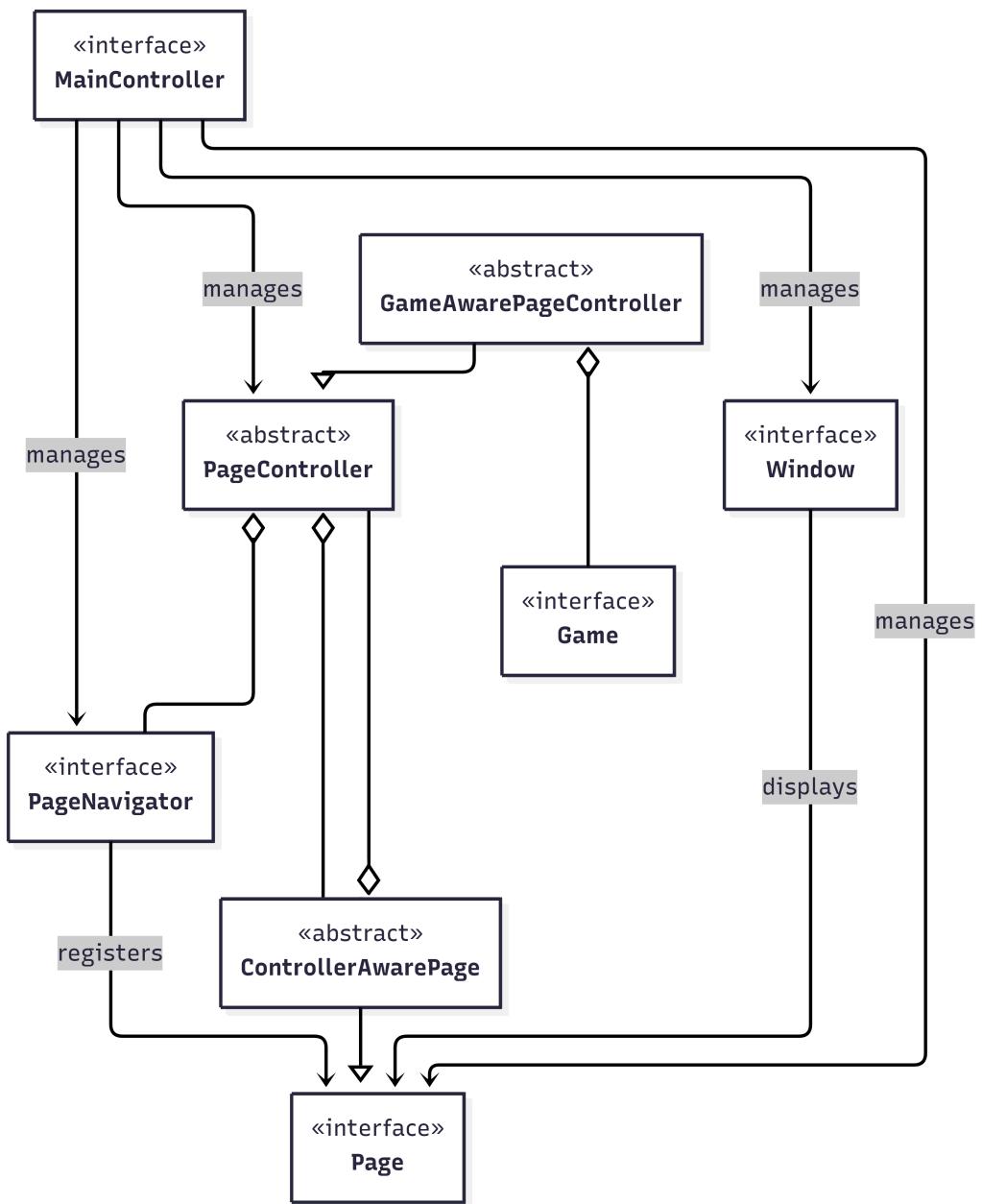


Figura 2.1: Schema UML architetturale di Javamant.

## 2.2 Design dettagliato

### 2.2.1 Filippo Gaggi

Creazione di giocatori con scelte automatizzate

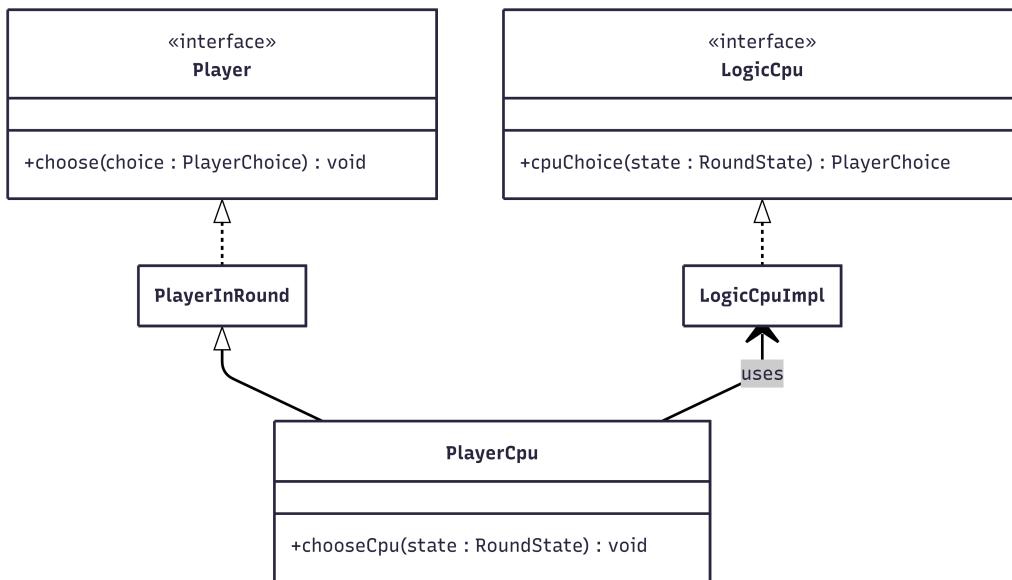


Figura 2.2: Rappresentazione UML

**Problema** : Si vogliono creare giocatori che danno scelte automatizzate.

**Soluzione** : Ho creato la classe **PlayerCpu** come estensione della classe rappresentante i giocatori reali **PlayerInRound**, la quale ha al suo interno un'istanza di **LogicCpuImpl** che le permette di assumere scelte automatizzate basate sulle informazioni della partita tramite il metodo `cpuChoice()`. In questo modo **PlayerCpu** funziona come un'effettiva CPU in grado di prendere decisioni autonomamente. Questa implementazione permette di tenere distinti i **PlayerCpu** dalle proprie logiche rendendo il tutto più modulare.

**Uso di pattern** : Nessuno.

## Creazione di più difficoltà per la LogicCpuImpl

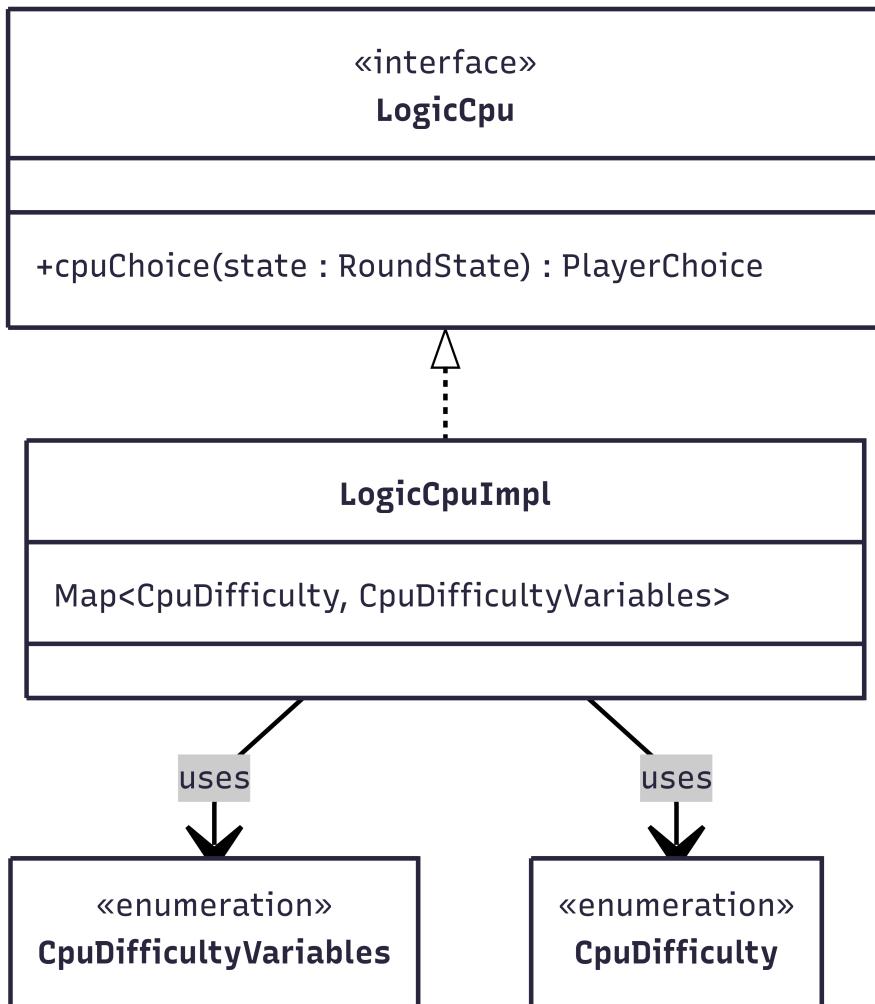


Figura 2.3: Rappresentazione UML

**Problema** : Si vogliono creare più difficoltà per la `LogicCpuImpl`.

**Soluzione** : Ho creato delle enumerazioni chiamate `CpuDifficulty` e `CpuDifficultyVariables` che rispettivamente contengono la denominazione delle difficoltà e le variabili di difficoltà utilizzate nella `LogicCpuImpl`. In questo modo tramite una mappa interna a `LogicCpuImpl` vengono abbinate le denominazioni delle difficoltà con i corrispettivi valori per le variabili che verranno usate all'interno di `LogicCpuImpl`. Questa implementazione

modulare rende l'aggiunta di nuove difficoltà e/o variabili di difficoltà in LogicCpuImpl facile e chiara.

**Uso di pattern** : Nessuno.

### Iterazione dei Round durante la partita

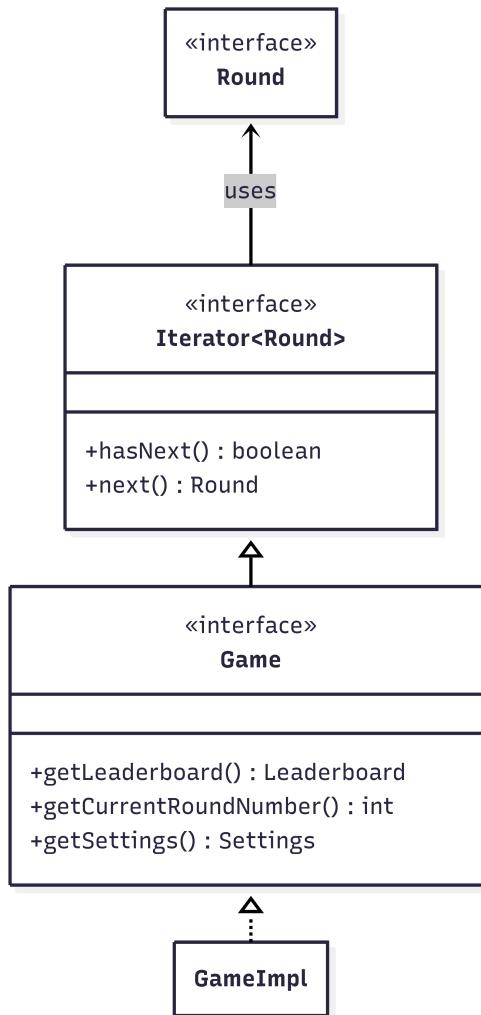


Figura 2.4: Rappresentazione UML

**Problema** : Si vuole passare da un Round ad un altro, avendo a disposizione le informazioni di questi e delle impostazioni della partita.

**Soluzione** : Ho creato l’interfaccia `Game` che estende `Iterator<Round>`. In questo modo posso passare da un round ad un altro avendo a disposizione tutte le loro informazioni.

**Uso di pattern** : `Iterator`.

## 2.2.2 Emir Wanes Aouioua

### Effetto speciale del round

**Problema** : L’effetto speciale del round determina le regole per cui un round deve finire e la quantità di gemme distribuite ai giocatori: questi due aspetti possono variare da una partita all’altra e possono dipendere da informazioni riguardanti lo stato del round. Serve un modo semplice per cambiare queste regole senza modificare come un round opera.

**Soluzione** : Viene utilizzato il pattern *Strategy* per incapsulare, in modo modulare

- come un round deve terminare, tramite l’interfaccia `EndCondition` che decide se un round ha raggiunto la sua fine ispezionando lo stato di quest’ultimo.
- come le gemme vengono modificare prima di essere guadagnate dai giocatori, tramite l’interfaccia `GemModifier` che applica una trasformazione alle gemme ricevute dopo aver ispezionato lo stato del round.

Entrambi vengono creati tramite *Factory Method* e accoppiati in un `RoundEffect`, usato dal Round senza conoscere i dettagli di come esso agisce 2.5.

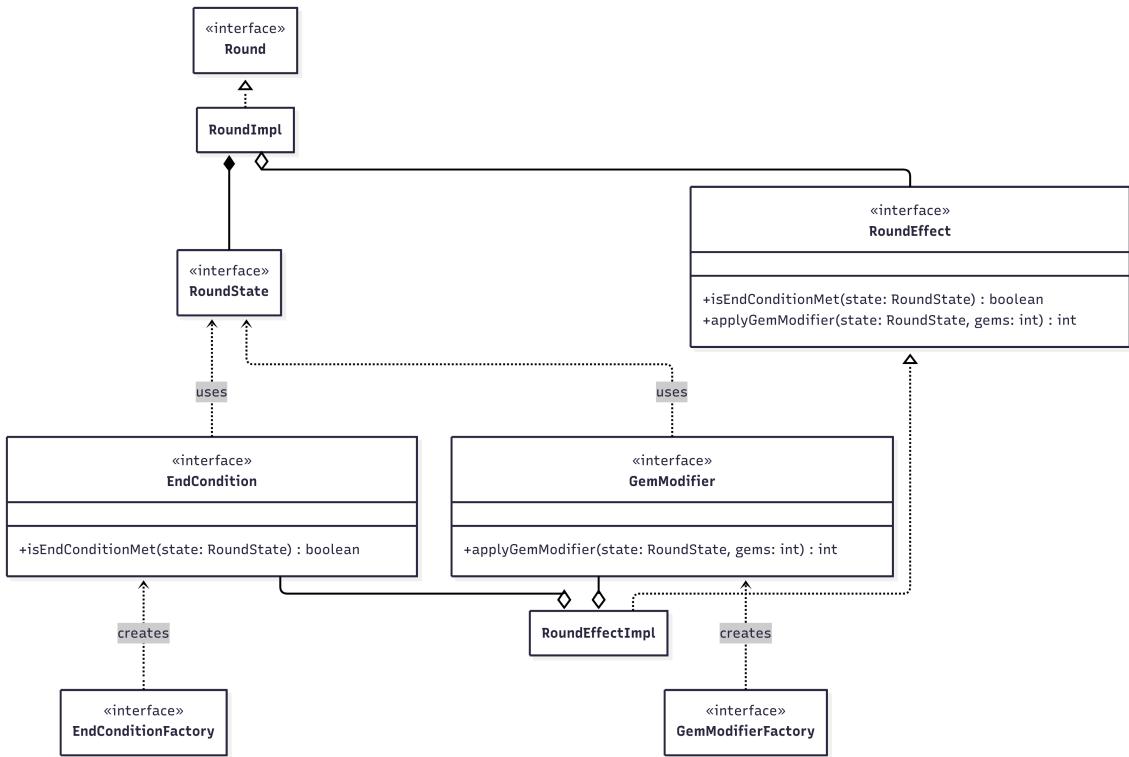


Figura 2.5: Il diagramma mostra come `EndCondition` e `GemModifier` rappresentino le strategie, creati dalle rispettive factory e composti in `RoundEffectImpl`. `RoundImpl` può utilizzare le due strategie in modo completamente interscambiabile.

**Pro :** l'utilizzo del pattern Strategy permette una facile interscambiabilità tra diversi tipi di approcci alla fine del round e alla modifica delle gemme, senza che avvenga alcuna modifica al Round. L'utilizzo di Factory Method permette di centralizzare la creazione delle varie strategie, riducendo la proliferazione di classi.

**Contro :** se si volessero combinare più stategie (come ad esempio rendere due diverse condizioni di fine vere per terminare il round), occorrerebbe introdurre nuove strategie composte, come classi singole o metodi nella factory. Una possibile soluzione a questo problema è l'utilizzo del pattern Decorator, il quale non viene utilizzato.

## Iterazione solo sui giocatori in esplorazione

**Problema** : durante un round i giocatori possono decidere di uscire. Serve un modo per poter passare al prossimo giocatore attivo, saltando quelli che hanno abbandonato l'esplorazione della grotta.

**Soluzione** : Uso del pattern *Iterator* da parte dell'interfaccia RoundPlayerManager che estende *Iterator<Player>* per ottenere con una sola chiamata il giocatore del prossimo turno 2.6.

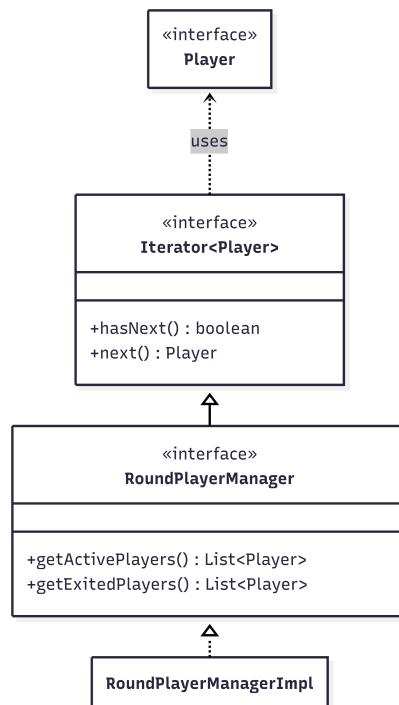


Figura 2.6: Il diagramma mostra come RoundPlayerManager sia un iteratore di Player, permettendo di ottenere il prossimo giocatore attivo e capire se ci sono ancora giocatori in esplorazione.

**Pro** : l'utilizzatore di RoundPlayerManager non deve preoccuparsi della logica con cui il prossimo giocatore della partita viene scelto, questo garantisce flessibilità e rimuove la responsabilità di scegliere il prossimo giocatore dal round.

**Contro** : una volta che l'Iterator viene consumato non è più riutilizzabile ed è necessario crearne un altro.

## Inizializzazione degli handler nelle pagine

**Problema** : per via dell'architettura scelta, ogni Page che necessita di interagibilità ha bisogno che venga associata ad uno specifico PageController; gli handler degli eventi relativi ai componenti grafici delle pagine variano per ognuna di esse e devono essere inizializzati solo quando lo specifico PageController viene assegnato alla pagina. Serve un modo per impostare gli handler dei componenti grafici non appena viene scelto quale PageController utilizzare sulla pagina.

**Soluzione** : tramite l'applicazione del pattern *Template Method* la specializzazione astratta di Page che modella pagine interagibili, `ControllerAwarePage`, richiama il metodo astratto `setHandlers()` nel momento in cui nella Page viene associato il proprio controller effettuando una chiamata al metodo `setController(PageController)`. In questo caso il metodo `setController(PageController)` costituisce il template method: esso associa il controller ed imposta gli handler della visualizzazione senza preoccuparsi di come le pagine `ControllerAwarePage` inizializzino questi ultimi. 2.7.

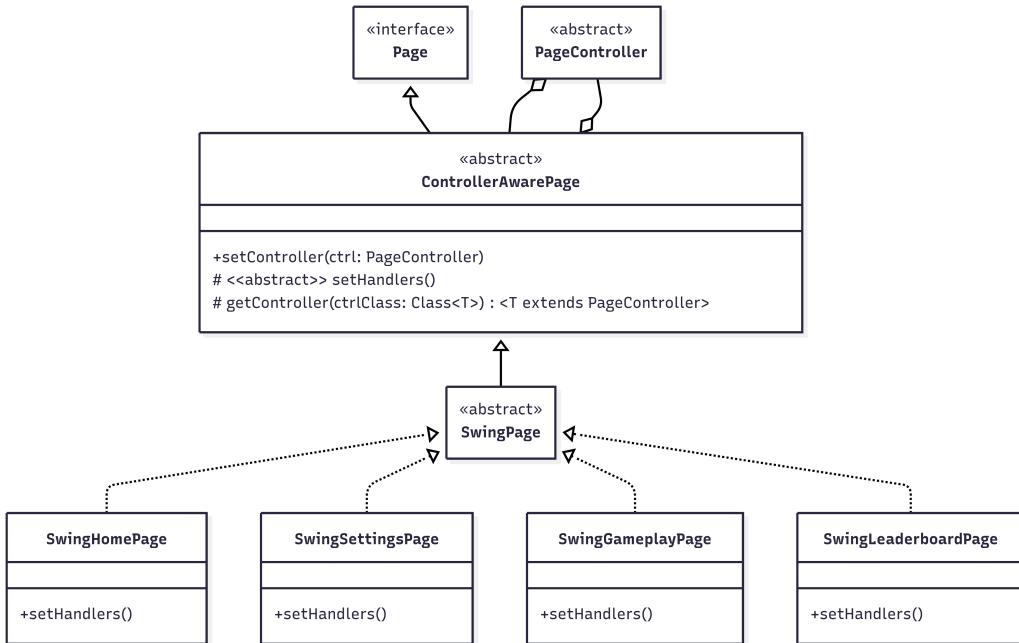


Figura 2.7: Il diagramma mostra il pattern Template Method applicato a `ControllerAwarePage`, in cui il metodo `setController(PageController)` fa da metodo template e `setHandlers()` da hook. Nota: il metodo `getController(class)` viene utilizzato per evitare la propagazione dei generici a tutto l'albero strutturale: esso è un metodo utilizzabile dalle `ControllerAwarePage` che effettua un cast sicuro alle specializzazioni di `PageController`.

**Pro** : c’è una separazione netta delle responsabilità: la logica comune (l’impostazione del controller) viene centralizzata in `ControllerAwarePage` mentre le sue estensioni si occupano solo di specificare quali eventi devono essere associati al controller. Questo comporta anche una riduzione del codice duplicato; estendibilità semplice: per aggiungere una nuova pagina interagibile basta implementare `setHandlers()`.

**Contro** : con Template Method la struttura diventa rigida: se si volesse cambiare un comportamento di `ControllerAwarePage` comporterebbe probabile duplicazione di codice. Un’altra problematica è il fatto che una pagina interagibile deve forzatamente estendere `ControllerAwarePage`, bloccando la possibilità di estendere da altre classi (ad esempio, `SwingPage` avrebbe potuto estendere direttamente `JPanel`).

## Centralizzazione della navigazione tra pagine

**Problema** : essendo la natura dell'applicazione interattiva e multipagina, occorre un modo per poter passare da una visualizzazione all'altra senza che un **PageController** mantenga riferimenti a pagine su cui non deve gestire l'interazione (un PageController deve gestire una ed una sola pagina).

**Soluzione** : si centralizza la logica di navigazione tra **Page** all'interno di un navigatore: **PageNavigator**. Il navigatore si occupa di registrare le pagine associandole ad una chiave (in questo caso l'enumerazione **PageId**) e cambiare la visualizzazione tramite quest'ultima. In questo modo PageController non dovrà preoccuparsi di mantenere riferimenti a pagine su cui non deve gestire l'interazione. 2.8 Questo approccio non utilizza alcun design pattern.<sup>1</sup>

---

<sup>1</sup>Questo approccio non rappresenta alcun OOP design pattern ma è un pattern architettonicale molto utilizzato in ambito mobile e web per la navigazione multipagina.

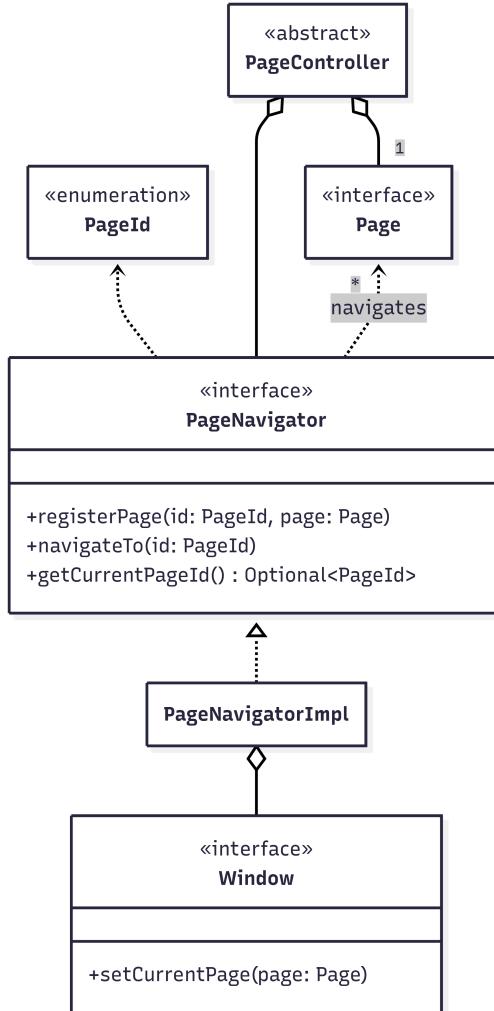


Figura 2.8: Il diagramma mostra come `PageController` gestisca una singola pagina e come `PageNavigator` coordini la navigazione verso altre pagine.

**Pro** : tutta la logica del passaggio da una pagina all'altra viene centralizzata in `PageNavigator`, riducendo la duplicazione di codice e favorendo il riutilizzo. `PageController` non necessita di alcuna informazione sulle altre pagine: gli basta conoscere la chiave associata. Si possono aggiungere nuove pagine navigabili in modo facile: basta registrare una coppia (`PageId`, `Page`).

**Contro** : l'utilizzo di `PageId` come chiave per navigare tra le pagine rende impossibile l'aggiunta di pagine create in modo dinamico poiché `PageId` presenta un numero finito di chiavi associabili (ad esempio, in futuro

si potrebbe voler aggiungere pagine senza struttura statica, create sul momento in base allo stato della partita).

### 2.2.3 Andrea La Tosa

#### GESTIONE DEL MAZZO

**Problema:** La gestione del mazzo doveva consentire sia di pescare una nuova carta, sia di verificare se fossero ancora presenti carte disponibili.

**Soluzione:** È stata adottata l'implementazione del design pattern *Iterator* che consente un accesso controllato e sequenziale alle carte del mazzo.

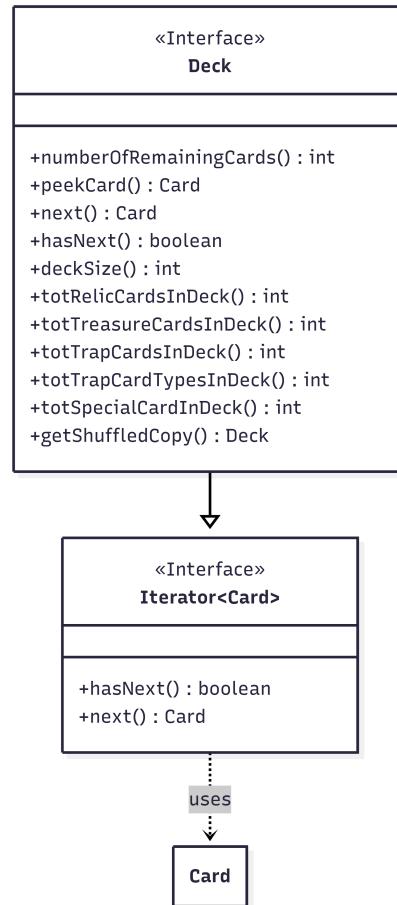


Figura 2.9: Diagramma UML dell'utilizzo del design pattern *Iterator*

## CREAZIONE MODULARE DEI MAZZI:

**Problema:** La creazione di mazzi modulari, composti da diversi tipi di carte e da configurazioni predefinite, richiedeva una struttura flessibile, riusabile e facilmente estendibile.

**Soluzione:** L'adozione dei design pattern *Builder* (in `DeckBuilderImpl`) e *Factory* (in `DeckFactoryImpl`) ha permesso rispettivamente la costruzione modulare dei mazzi e la generazione di configurazioni predefinite in modo semplice e manutenibile.

### Vantaggi:

- **manutenibilità:** ogni classe ha una responsabilità ridotta e ben definita.
- **estendibilità:** è semplice aggiungere nuove funzionalità o varianti di mazzi.
- **testing:** i test sono semplici da effettuare grazie alla separazione delle responsabilità.
- **Riusabilità del codice:** la logica di costruzione e configurazione dei mazzi può essere riutilizzata per creare nuove varianti senza duplicazioni.

**Svantaggi:** Lo svantaggio principale riguarda la maggiore complessità iniziale nella comprensione della struttura del codice, dovuta all'introduzione di più classi e pattern.

**Possibile implementazione alternativa:** Una possibile alternativa consiste nella creazione di un'unica classe per la gestione del mazzo e delle sue statistiche avente, al suo interno, una classe innestata dedicata alla costruzione delle diverse tipologie di deck, contenente anche i metodi per l'aggiunta delle carte. Questa implementazione è stata scartata perché risultava confusa, difficilmente manutenibile e violava il principio SRP (single responsibility principle) trasformandosi di fatto in una God Class.

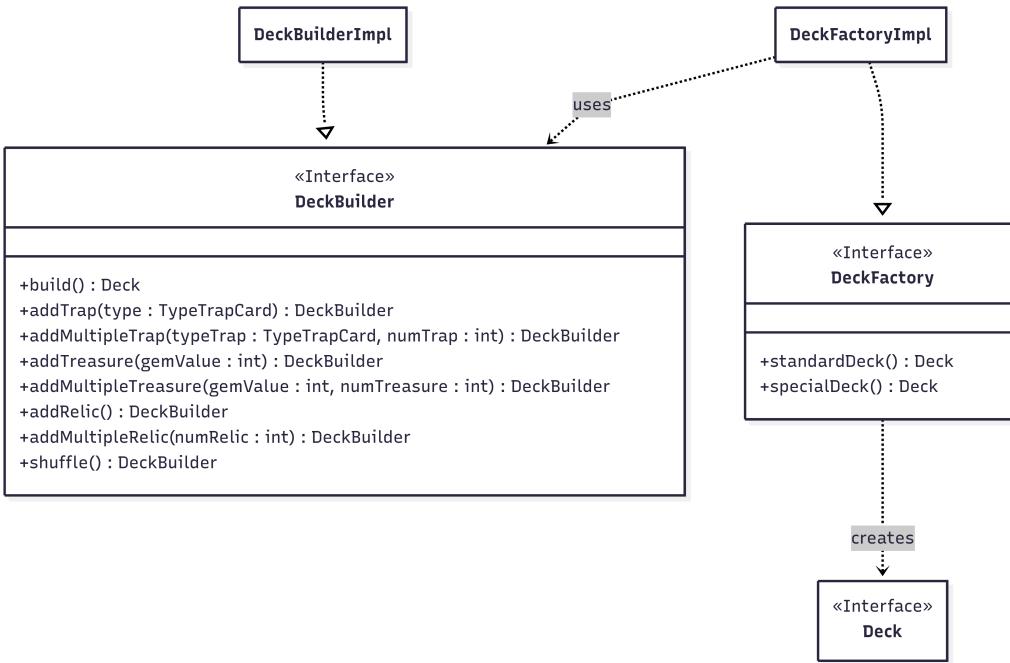


Figura 2.10: Diagramma UML dell'utilizzo dei design pattern *Builder* e *factory*

## GESTIONE ESTENDIBILE DELLE CARTE

**Problema:** Era necessario definire una struttura flessibile per rappresentare le carte. Questa doveva permettere di modellare facilmente le tre tipologie di carte attuali (reliquia, trappola e tesoro) e supportare, inoltre, l'aggiunta di nuove tipologie in futuro, senza modificare il codice esistente.

### Soluzione:

E' stata progettata una gerarchia basata sulla classe **Card** contenente i comportamenti comuni a tutte le carte. Per rispettare il principio DRY (Don't Repeat Yourself) e mantenere una logica flessibile è stata introdotta la classe **CardWithGem** che estende **Card** e aggiunge la logica relativa alla gestione delle gemme. In questo modo le tipologie di carte attuali (reliquia, trappola e tesoro) possono estendere **Card** o **CardWithGem** a seconda della presenza o meno di gemme associate alla carta e l'aggiunta di nuove tipologie di carte potrà essere effettuata senza modificare il codice prodotto.

### Vantaggi:

- **Estendibilità:** è possibile introdurre facilmente nuove tipologie di carte senza modificare la struttura esistente.
- **Riutilizzo del codice:** i comportamenti comuni sono centralizzati nella classe base di Card.

**Svantaggi:**

- **Rigidità della gerarchia:** ogni nuova tipologia di carta richiede la creazione di una sottoclassificazione dedicata.
- **Impossibilità di combinare più effetti:** la struttura scelta non prevede che le carte possano combinare più effetti contemporaneamente.

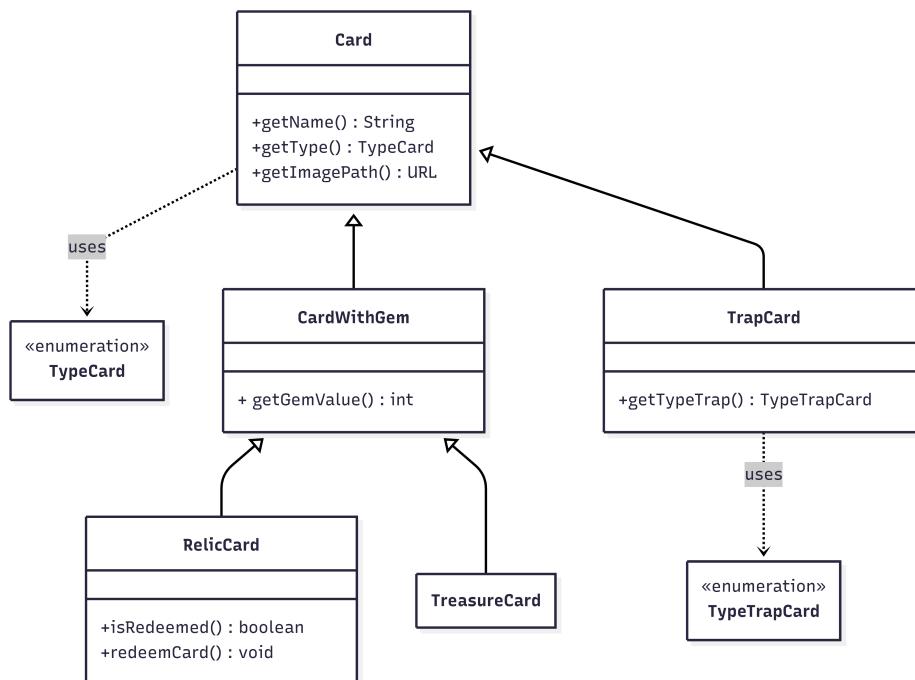


Figura 2.11: Diagramma UML della gerarchia delle carte

# Capitolo 3

## Sviluppo

### 3.1 Testing automatizzato

Il testing automatizzato è stato effettuato con *JUnit*. Per motivi di tempo non siamo riusciti a testare la GUI e la maggior parte del controller, decidendo di dare precedenza ai test delle classi di dominio. Di seguito i test effettuati da ogni componente del gruppo:

- **Andrea La Tosa:**

- `DeckTest`: verifica della corretta composizione del mazzo standard, inclusa l'estrazione delle carte.
- `RelicCardTest`: verifica dei valori di gemme ammessi, del riscatto (`redeemCard()`) e stato (`isRedeemed()`) della carta.
- `TrapCardTest`: verifica della correttezza dei percorsi delle immagini per ogni tipologia di trappola e del confronto tra carte.
- `TreasureCardTest`: verifica dei valori di gemme validi, della correttezza dei percorsi immagine e del confronto tra carte tesoro.

- **Emir Wanes Aouioua:**

- `EndConditionFactoryImplTest`: mi sono assicurato che tutte le condizioni di fine round rispettassero i criteri con cui erano state definite.
- `GemModifierFactoryImplTest`: mi sono assicurato che i modificatori di gemme restituissero la quantità corretta di gemme, modificate in base allo stato di un round.

- **RoundImplTest**: mi sono assicurato che un round gestisse correttamente l’iterazione dei turni e che alcune chiamate ai metodi avvenissero nel giusto ordine.
- **RoundPlayersManagerImplTest**: mi sono assicurato che in un round venisse riconosciuto il giocatore corretto per il turno successivo.
- **RoundStateImplTest**: mi sono assicurato che le informazioni condivise sullo stato del round fossero corrette.
- **TurnImplTest**: mi sono assicurato che i metodi di un turno venissero utilizzati nel giusto ordine e che la distribuzione delle gemme fosse quella aspettata.
- **FullGameTest**: questo test simula un numero di N partite (nel test fissato a 1000).
- **PageNavigatorImpl**: mi sono assicurato che il sistema di navigazione tra pagine funzionasse correttamente, spostando la visualizzazione e rilevando la pagina corrente quando richiesto.

- **Filippo Gaggi:**

- **LogicCpuTest**: Testing riguardante la correttezza di comportamento di LogicCpuImpl nelle 3 difficoltà.
- **PlayerInRoundTest**: Testing riguardante la correttezza dei metodi di PlayerInRound.
- **PlayerCpuTest**: Testing riguardante la correttezza dei metodi di PlayerCpu.
- **GameSettingsTest**: Testing riguardante la correttezza di comportamento dei controlli di GameSettingsImpl.
- **LeaderboardTest**: Testing riguardante la correttezza dei metodi di LeaderboardImpl.

## 3.2 Note di sviluppo

### 3.2.1 Filippo Gaggi

#### Utilizzo di Stream e Lambda expressions

Utilizzati in vari punti. Un esempio è <https://github.com/Wanes01/OOP24-jvmt/blob/ccdbae592c490476d3226350cbdbbeb99e32a596/src/main/java/jvmt/controller/impl/LeaderboardControllerImpl.java#L48C1-L53C6>

## Utilizzo di Optional

<https://github.com/Wanes01/OOP24-jvmt/blob/ccdbae592c490476d3226350cbdbbeb99e32a596/src/main/java/jvmt/controller/impl/GameplayControllerImpl.java#L197-L206>

### 3.2.2 Emir Wanes Aouioua

#### Progettazione con generici

Utilizzati in vari punti, sia per design di classe che su singoli metodi: <https://github.com/Wanes01/OOP24-jvmt/blob/ccdbae592c490476d3226350cbdbbeb99e32a596/src/test/java/jvmt/navigator/PageNavigatorImplTest.java#L99C1-L101C64>

#### Uso di lamda expressions

Utilizzate molto spesso: <https://github.com/Wanes01/OOP24-jvmt/blame/ccdbae592c490476d3226350cbdbbeb99e32a596/src/main/java/jvmt/model/round/impl/roundeffect/gemmodifier/GemModifierFactoryImpl.java#L43>

#### Uso di Stream

Utilizzati molto spesso: <https://github.com/Wanes01/OOP24-jvmt/blob/ccdbae592c490476d3226350cbdbbeb99e32a596/src/main/java/jvmt/model/round/impl/roundeffect/endcondition/EndConditionFactoryImpl.java#L79C1-L80C52>

#### Uso di Optional

Utilizzati frequentemente per evitare null: <https://github.com/Wanes01/OOP24-jvmt/blame/ccdbae592c490476d3226350cbdbbeb99e32a596/src/main/java/jvmt/model/round/impl/turn/TurnImpl.java#L60>

#### Uso di interfacce funzionali

Utilizzate frequentemente: <https://github.com/Wanes01/OOP24-jvmt/blob/ccdbae592c490476d3226350cbdbbeb99e32a596/src/main/java/jvmt/model/round/impl/RoundStateImpl.java#L88C1-L89C44>

## Codice di terze parti

- ho effettuato un adattamento dello snippet di *Joel Carranza* come risposta alla sequente domanda su StackOverflow.<sup>1</sup> L'adattamento viene utilizzato nella classe `SwingWindow` per adattare la dimensione del font per tutte le pagine Swing in base al DPI dello schermo utilizzato.

### 3.2.3 Andrea La Tosa

**Utilizzo di generici:**

<https://github.com/Wanes01/OOP24-jvmt/blob/bbeaba836d375717ec403006b1f065be87256src/main/java/jvmt/view/page/utility/ComboboxWithLabel.java#L91C1-L96C6>

**Utilizzo di stream e lambda:**

<https://github.com/Wanes01/OOP24-jvmt/blob/ccdbae592c490476d3226350cbdbbeb99e32asrc/main/java/jvmt/view/page/impl/SwingSettingsPage.java#L313C9-L318C10>

**Utilizzo della dipendenza esterna MigLayout:**

Utilizzato in vari punti, un esempio: <https://github.com/Wanes01/OOP24-jvmt/blame/3db0df6d99f04fbc8d3c0bd161ff8eabd5c3c483/src/main/java/jvmt/view/page/impl/SwingHomePage.java#L43-L43C50>

**Utilizzo di consumer:**

<https://github.com/Wanes01/OOP24-jvmt/blame/3db0df6d99f04fbc8d3c0bd161ff8eabd5c3src/main/java/jvmt/controller/impl/SettingsControllerImpl.java#L66>

---

<sup>1</sup>Joel Carranza, font scalabile a prescindere dal DPI: <https://stackoverflow.com/questions/1102216/can-i-set-the-dpi-resolution-of-my-java-swing-application-without-changing-the-s>

# Capitolo 4

## Commenti finali

### 4.1 Autovalutazione e lavori futuri

#### 4.1.1 Filippo Gaggi

Nonostante non abbia utilizzato molte tecniche avanzate di Java, ritengo di aver scritto codice pulito e ben comprensibile. Vado molto fiero della logica delle CPU, la quale ha preso una moderata quantità tempo per farla comportare come desideravo. Seppur non abbia brillato molto nella parte di model, mi sono impegnato quanto più potevo nei controller e nelle corrispettive pagine (Gameplay e Leaderboard), portando un risultato che reputo più che apprezzabile sia dal lato grafico che dal lato del codice. Ritengo tutto sommato che il mio contributo al gruppo sia stato importante anche se un po' grezzo e non troppo brillante.

#### 4.1.2 Emir Wanes Aouioua

Nel complesso, sono soddisfatto del lavoro svolto nel progetto. Dopo la fase iniziale di analisi del dominio assieme ai miei colleghi, mi sono occupato in modo particolare della parte di model e della definizione dello scheletro architettonale. Questo mi ha portato ad assumere un ruolo piuttosto centrale nelle scelte progettuali, che spesso è risultato anche più impegnativo di quanto avessi previsto. Ammetto che la divisione del carico di lavoro non sia stata ben bilanciata, ma considero questo un aspetto naturale ed inevitabile in un primo progetto di sviluppo cooperativo, e nonostante senta di aver avuto un maggior numero di responsabilità penso che ciascuno di noi abbia messo a disposizione al meglio le proprie competenze contribuendo al progetto. Sono contento della coesione e disponibilità reciproca che si è mantenuta all'interno del gruppo dall'inizio alla fine. Col senno di poi, alcune scelte architettoniche

che ho fatto potrebbero essere migliorate: ad esempio rivedrei il modo in cui pagine e relativi controller vengono associati. Nel complesso però sono fiero della struttura che ho realizzato. Non credo che porterei avanti il progetto nel suo complesso, ma trovo che l'architettura possa avere del potenziale: mi piacerebbe rifinirla per utilizzarla in altre applicazioni personali.

#### **4.1.3 Andrea La Tosa**

Ho svolto la mia parte di progetto al meglio delle mie capacità e ritengo di aver dato un valido contributo al gruppo. Il progetto è stato svolto in un ambiente sereno e tranquillo il che ha reso molto facile scambiarsi idee e punti di vista, senza i quali avrei prodotto codice meno leggibile e manutenibile. Sono molto contento del codice che ho prodotto anche se devo ammettere che ho percepito molto la mia inesperienza in certi argomenti, primo fra tutti GitHub.

### **4.2 Difficoltà incontrate e commenti per i docenti**

#### **4.2.1 Emir Wanes Aouioua**

Sono soddisfatto di come sia stato strutturato il corso di OOP ma presento una piccola critica legata ad una mia passata difficoltà: al giorno d'oggi sono fiero di poter dire di aver capito i design pattern che sono stati presentati a lezione e, dopo tanto allenamento personale, non saprei più come lavorare senza (soprattutto strategy, che ritengo uno dei più versatili); detto ciò, però, penso che durante il corso si dia davvero poco tempo a disposizione ad un argomento così importante e su cui verte così tanto la valutazione tra progetto ed esame pratico in laboratorio. Personalmente concederei almeno una lezione teorica ed una pratica aggiuntive al fine di far comprendere meglio gli OOP design pattern.

# Appendice A

## Guida utente

### A.1 Home



Figura A.1: Scermata Home di Javamant.

### A.2 Impostazioni

Nella pagina delle impostazioni è possibile selezionare le impostazioni desiderate, tra le quali:

- nomi dei giocatori (un nome corrisponde ad un giocatore reale)
- numero delle CPU

- difficoltà delle CPU
- tipo di deck
- condizione di fine round
- modificatori delle gemme
- numero di round

Una volta settate le impostazioni si può iniziare la partita premendo il pulsante "START GAME".

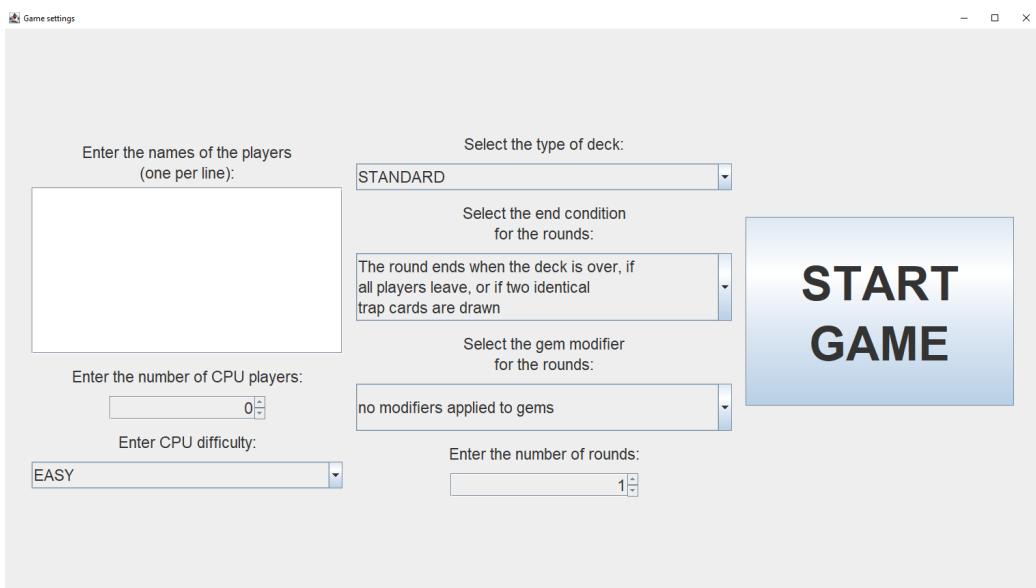


Figura A.2: Scermata delle impostazioni di Javamant.

### A.3 Impostazioni non corrette

Se per caso non vengono rispettati i requisiti delle impostazioni, quali:

- nomi dei giocatori che superano i 12 caratteri.
- numero dei giocatori inferiori a 3 o superiori a 8.

Apparirà un pop up che informa riguardo i settaggi non corretti.

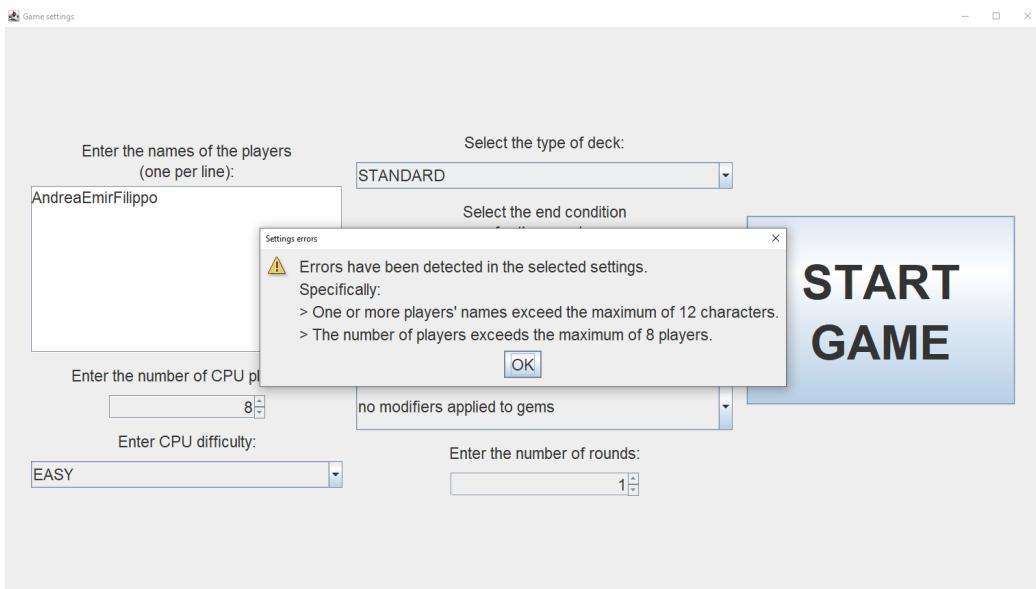


Figura A.3: Pop up che informa sui settaggi sbagliati delle impostazioni di gioco.

## A.4 Gameplay

### Fase di pesca

Se è il turno di un giocatore reale sarà possibile premere il pulsante "DRAW" per pescare una carta dal mazzo.

Se è il turno di una CPU il pulsante per pescare verrà disabilitato e la pesca verrà eseguita automaticamente.



Figura A.4: Scermata del gioco nel turno di un giocatore reale.



Figura A.5: Scermata del gioco nel turno di una CPU.

### Fase di decisione

Dopo la fase di pesca ciascun giocatore dovrà scegliere se rimanere in gioco oppure uscire premendo il pulsante con la scelta desiderata, le CPU

decideranno automaticamente.

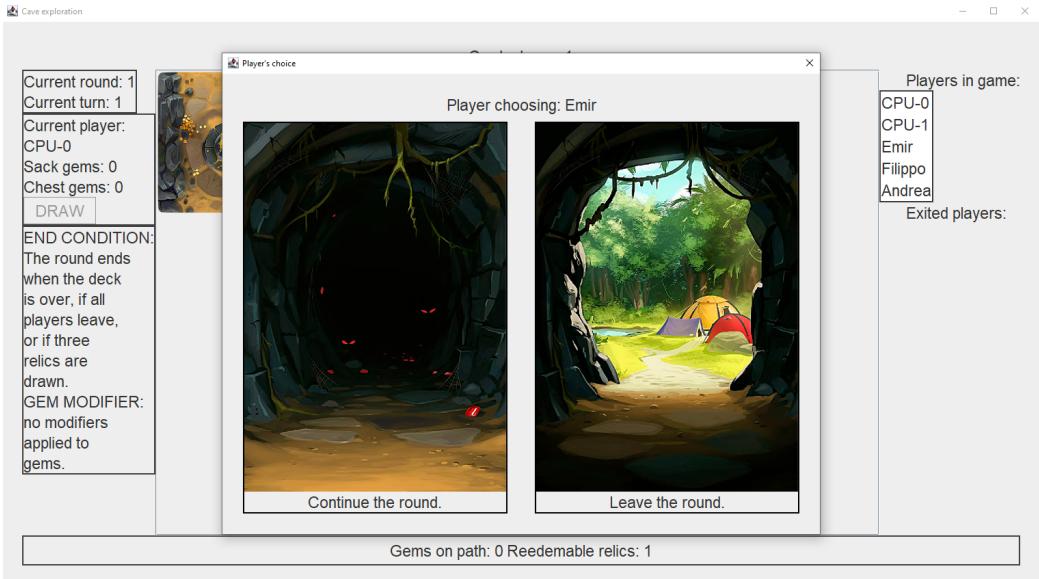


Figura A.6: Pop up contenente le possibili scelte.

### Fine round/Fine partita

Il turno continuerà finché la condizione di fine round non si verifica.  
Similmente, la partita continuerà finché il numero di round non finisce.



Figura A.7: Pop up che informa sulla fine del round.



Figura A.8: Pop up che informa sulla fine della partita.

## A.5 Leaderboard

Una volta che finisce la partita si verrà indirizzati al tabellone con la classifica finale, dal quale è possibile tornare alla Home premendo il pulsante "Go back to Home page".

Il vincitore sarà colui che accumulerà più gemme.

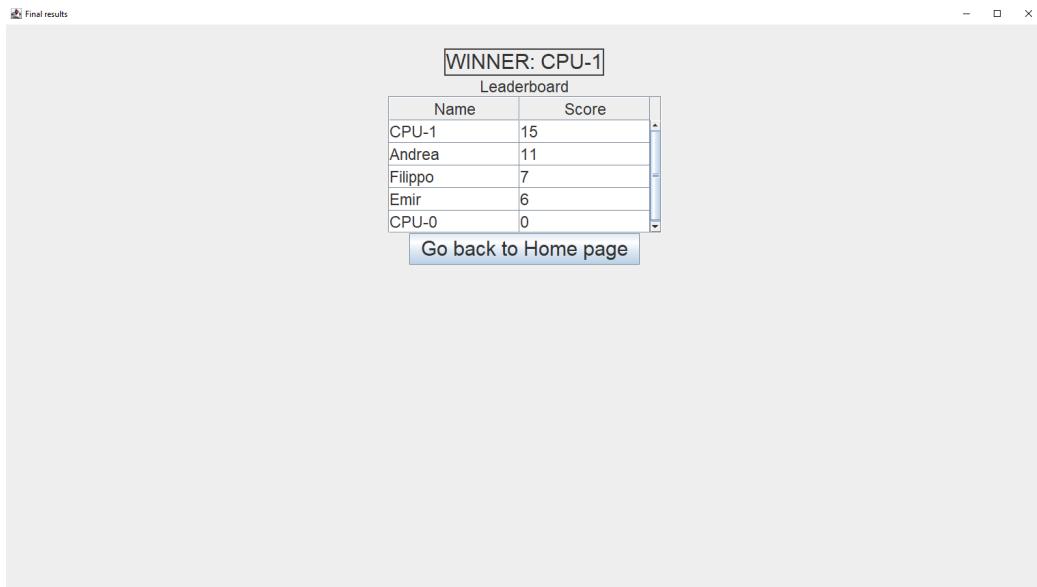


Figura A.9: Scermata della leaderboard.

# Appendice B

## Esercitazioni di laboratorio

### B.1 Emir Wanes Aouioua

- Laboratorio 07: <https://virtuale.unibo.it/mod/forum/discuss.php?d=177162#p245995>
- Laboratorio 08: <https://virtuale.unibo.it/mod/forum/discuss.php?d=178723#p247237>
- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=179154#p247922>
- Laboratorio 10: <https://virtuale.unibo.it/mod/forum/discuss.php?d=180101#p249324>
- Laboratorio 11: <https://virtuale.unibo.it/mod/forum/discuss.php?d=181206#p250921>