

Relazione per Programmazione a Oggetti

Gold-Hunt

Azzurra Quattrini

Sara Quirici

Luca Galassi

David Andrei Orosanu

Indice

1 Analisi	2
1.1 Descrizione e requisiti	2
1.2 Modello del Dominio	2
2 Design	4
2.1 Architettura	4
2.2 Design dettagliato	5
2.2.1 Azzurra Quattrini	5
2.2.2 Sara Quirici	10
2.2.3 Luca Galassi	12
2.2.4 David Andrei Orosanu	13
3 Sviluppo	19
3.1 Testing automatizzato	19
3.2 Note di sviluppo	21
3.2.1 Azzurra Quattrini	21
3.2.2 Sara Quirici	21
3.2.3 Luca Galassi	21
3.2.4 David Andrei Orosanu	22
4 Commenti finali	23
4.1 Autovalutazione e lavori futuri	23
4.1.1 Azzurra Quattrini	23
4.1.2 Sara Quirici	23
4.1.3 Luca Galassi	24
4.1.4 David Andrei Orosanu	24
A Guida utente	25
B Esercitazioni di laboratorio	27
B.1 davidandrei.orosanu@studio.unibo.it	27

1 Analisi

1.1 Descrizione e requisiti

L'applicazione mira alla realizzazione di un videogioco single-player ispirato a Microsoft Treasure Hunt. Ci sono tre difficoltà del livello tra cui scegliere. All'inizio di ogni livello il giocatore ha tre vite. Il giocatore si muove all'interno di una mappa e il suo obiettivo è raggiungere le scale per completare il livello.

La mappa è una griglia suddivisa in caselle: ogni casella può essere sicura o contenere una trappola. Su una casella si possono fare due operazioni: avanzare rivelandone il contenuto oppure contrassegnarla nel caso in cui si pensi che ci sia una trappola. Nel caso in cui la casella contenga una trappola, il giocatore perde una vita; se il giocatore perde tutte le vite la partita termina. Nel caso in cui la casella sia sicura si può trovare un numero da 1 a 8, che indica la quantità di trappole presenti nelle caselle adiacenti. Se una casella sicura non è adiacente a nessuna trappola, quando viene rivelata il gioco "ripulisce" automaticamente tutte le caselle vuote adiacenti a quella cliccata.

Una casella sicura senza trappole adiacenti può contenere anche degli oggetti utili al progresso nel gioco, utilizzabili dall'inventario del giocatore. Alla fine di ogni livello, si possono acquistare questi oggetti nello shop.

Requisiti funzionali

- Gestire il contenuto delle caselle e consentire al giocatore di muoversi su di esse.
- Permettere la raccolta di lingotti d'oro come valuta di gioco, che saranno utilizzabili in uno shop per l'acquisto di power-up e oggetti.
- Determinare la vittoria del giocatore al raggiungimento delle scale.

Requisiti non funzionali

- Il sistema deve garantire un'interfaccia chiara e intuitiva per facilitare la comprensione delle meccaniche di gioco.
- Il software deve essere facilmente estendibile per consentire l'aggiunta di nuove tipologie di oggetti o meccaniche di gioco.

1.2 Modello del Dominio

Il gioco è strutturato come una sequenza di livelli (**Level**), ciascuno caratterizzato da una specifica configurazione di difficoltà. La difficoltà determina la creazione di un oggetto **LevelConfig**, che rappresenta la configurazione del livello corrente. Fornisce infatti i parametri fondamentali del livello, tra cui la dimensione della **Board**, il numero di trappole (**Revealable**) e il numero di oggetti (**ItemTypes**) che saranno presenti nella **Board**. Una volta ottenuta la configurazione, il **BoardGenerator** utilizza i parametri forniti da **LevelConfig** per creare la **Board** del livello specificato. La **Board** viene infatti inizializzata all'inizio di ogni livello. Essa rappresenta la griglia di gioco ed è composta da un insieme di caselle (**Cell**). Ogni casella può contenere massimo un **CellContent**, che può essere un oggetto raccoglibile o una trappola. All'interno della **Board** si muove il **Player**. **Player** possiede una posizione corrente sulla griglia, un inventario (**Inventory**) e uno numero di vite. L'inventario contiene gli oggetti raccolti durante l'esplorazione della board o acquistati nello **Shop**. Le interazioni tra **Player** e **Board** avvengono grazie al coordinamento dell'**Engine**. Quando il **Player** raggiunge la casella di uscita della **Board**, il livello termina. Al termine del livello si accede allo **Shop**, dove è possibile acquistare nuovi **ItemTypes**. Gli oggetti acquistati vengono inseriti nell'inventario del **Player** e potranno essere utilizzati nei livelli successivi. Lo **Shop** rappresenta un sottoinsieme separato, ma integrato nel flusso del gioco. Tutte le operazioni sono gestite dall'**Engine**. Questo è infatti il coordinatore centrale del dominio, che riceve le richieste di azione, aggiorna lo stato del gioco e coordina le interazioni tra **Board**, **Player**, **Shop** e **Level**. L'**Engine** non contiene la logica specifica delle singole entità, ma si occupa di orchestrare il loro comportamento complessivo. L'**Engine** si appoggia a **Status**, che può essere descritto come un'istantanea dello stato corrente della partita. **Status** contiene tutte le informazioni necessarie a rappresentare l'istante attuale di gioco, come: stato della **Board**, posizione del **Player**,

inventario, se la partita è in corso, se è stata vinta o se è stata persa. Ogni volta che avviene un'azione, l'**Engine** aggiorna lo **Status** producendo una nuova fotografia coerente dello stato del sistema.

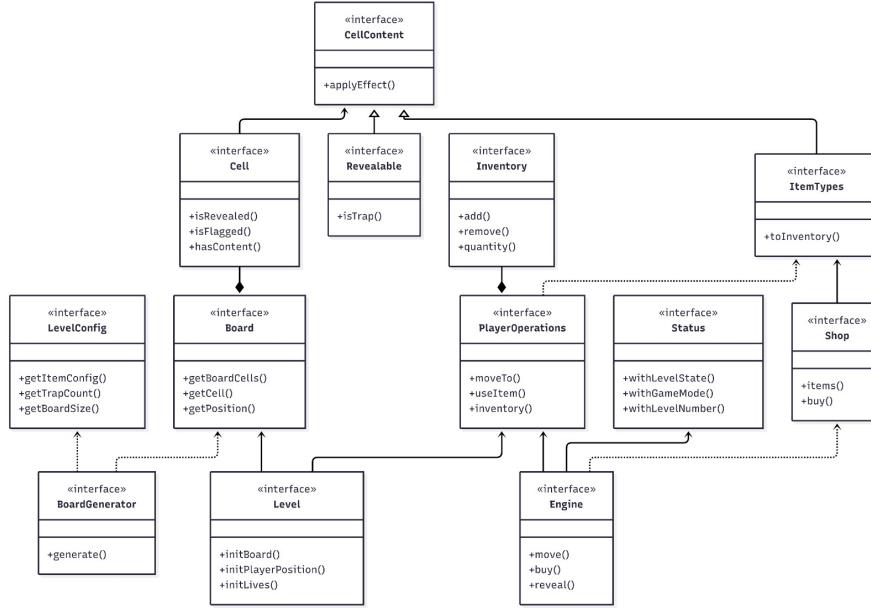


Figura 1: Rappresentazione UML del modello del dominio

2 Design

2.1 Architettura

Lo schema UML presentato descrive l'architettura di interazione tra model, controller e view, evidenziando come lo stato del gioco venga gestito e trasferito fino alla sua rappresentazione grafica. Il componente centrale del model è la classe **GameSession**, che svolge il ruolo di façade e costituisce l'unico punto di accesso alla logica di gioco. Essa espone i metodi necessari per eseguire le azioni principali, come il movimento del giocatore, la rivelazione delle celle, la gestione delle bandiere, l'utilizzo degli oggetti e gli acquisti nello shop. Questa soluzione consente di encapsulare completamente la logica interna e di evitare accessi diretti ai sottosistemi del model.

L'interfaccia **ViewStateMapper** ha la responsabilità di trasformare lo stato corrente del gioco, rappresentato da **GameSession**, in un oggetto **GameViewState**, ovvero una rappresentazione immutabile contenente tutte le informazioni necessarie alla visualizzazione. Questo livello di mapping garantisce un disaccoppiamento completo tra model e view, impedendo alla view di accedere direttamente ai dati interni e assicurando una chiara separazione delle responsabilità.

La view, rappresentata dall'interfaccia **GameView**, si occupa esclusivamente della visualizzazione dello stato ricevuto e della propagazione delle azioni dell'utente. Il coordinamento è gestito dal **GameController**, che riceve i comandi della view sotto forma di oggetti **GuiCommand**, li applica al model tramite **GameSession** e aggiorna successivamente la view utilizzando il **ViewStateMapper**. L'uso del Command Pattern consente di rappresentare in modo esplicito le azioni dell'utente, migliorando il disaccoppiamento tra i componenti.

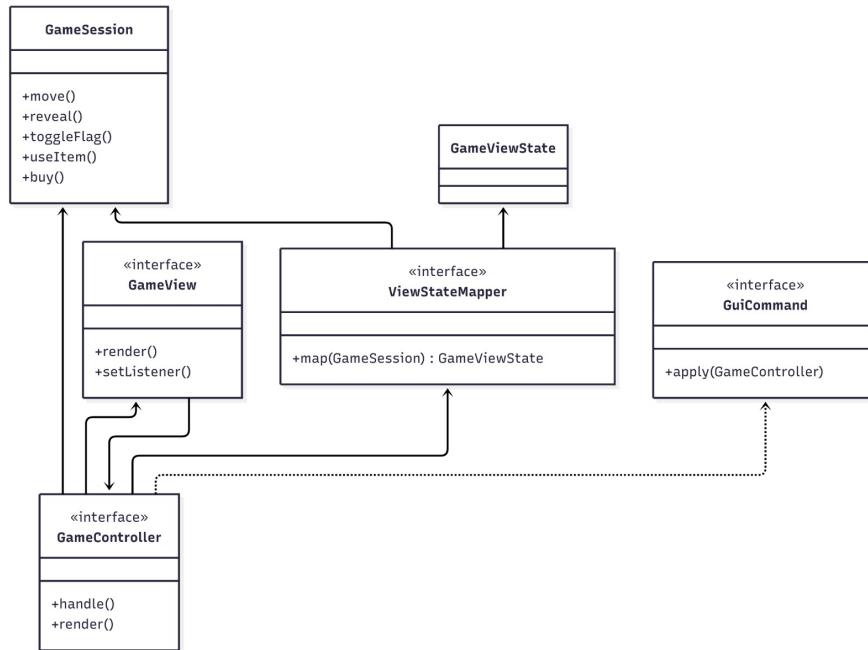


Figura 2: Rappresentazione UML architetturale di Gold-Hunt

2.2 Design dettagliato

2.2.1 Azzurra Quattrini

Creazione delle caselle e della griglia di gioco

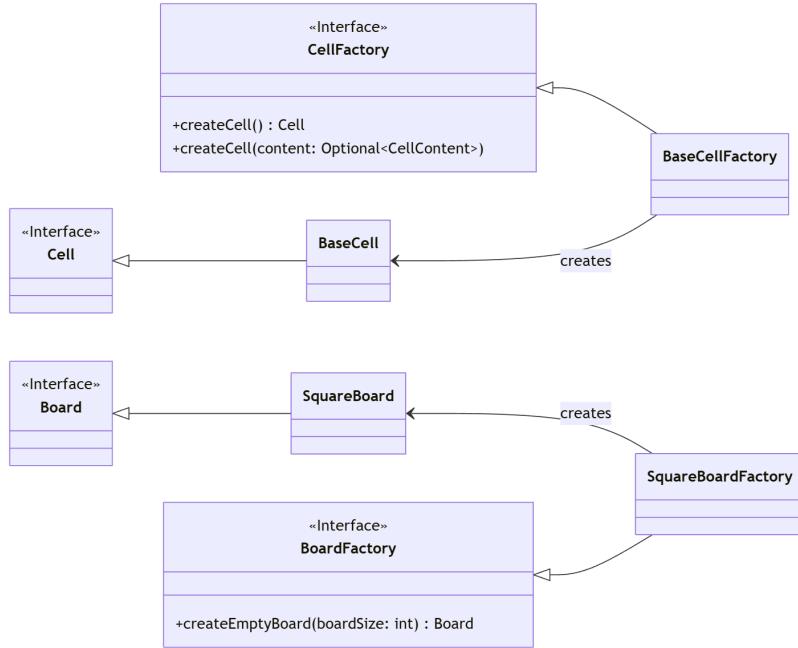


Figura 3: Rappresentazione UML del pattern Factory Method per la creazione delle caselle e della griglia di gioco

Problema La creazione delle caselle e della griglia di gioco deve avvenire senza dipendere dalle implementazioni concrete, per favorire la flessibilità e per evitare modifiche al codice già scritto e testato.

Soluzione Si utilizza il *pattern Factory Method* come da figura. Le interfacce `CellFactory` e `BoardFactory` definiscono i factory methods `createCell()`, `createCell(Optional<Content> content)` e `createEmptyBoard(int boardSize)`, restituendo le interfacce `Cell` e `Board`. Le classi concrete `BaseCellFactory` e `SquareBoardFactory` incapsulano la logica di istanziazione degli oggetti concreti `BaseCell` e `SquareBoard`. In questo modo la creazione degli oggetti è separata dal loro utilizzo, riducendo l'accoppiamento e migliorando l'estendibilità del sistema.

Strategie di rivelazione delle caselle

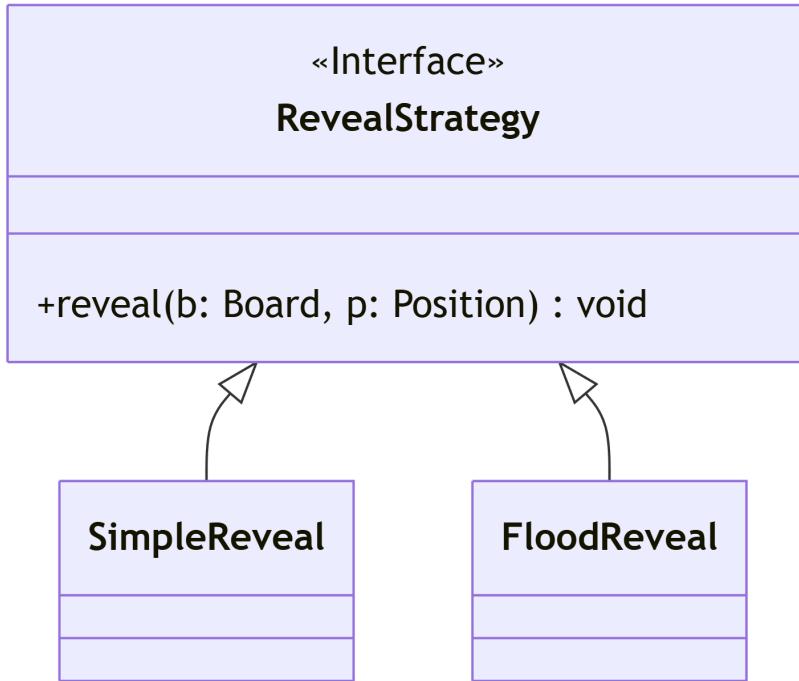


Figura 4: Rappresentazione UML del pattern Strategy per la rivelazione delle caselle

Problema La rivelazione delle caselle della griglia di gioco può avvenire seguendo diversi algoritmi (es: rivelazione semplice, rivelazione a cascata), e i componenti che usano la griglia devono potere agire indipendentemente dall'implementazione concreta di questi algoritmi.

Soluzione Si utilizza il *pattern Strategy* come da figura. Il comportamento di rivelazione delle caselle è contenuto nell'interfaccia **RevealStrategy**, che viene implementata da classi concrete per definire comportamenti alternativi e intercambiabili.

Accesso in sola lettura alle caselle e alla griglia di gioco

Problema I client devono potere consultare lo stato del modello di gioco senza poterlo modificare direttamente. Le interfacce `Cell` e `Board` espongono metodi che mutano le condizioni della casella e della griglia di gioco. Per esempio, il metodo `getCell(Position p)` in `Board` restituisce un riferimento diretto alla `Cell` che, combinato ai metodi mutanti di `Cell` (es: `reveal()`, `toggleFlag()`, `setContent()` e altri), permetterebbe di modificare lo stato di gioco fuori dalla logica e porterebbe a una rottura dell'incapsulamento.

Soluzione Si utilizza il *pattern Proxy* come da figura. Le interfacce `ReadOnlyCell` e `ReadOnlyBoard` espongono solo metodi per operazioni read-only (*Protection Proxy*). Le classi `ReadOnlyCellAdapter` e `ReadOnlyBoardAdapter` fungono da proxy e delegano le chiamate agli oggetti reali `Cell` e `Board` impedendo l'accesso ai metodi mutanti. La classe `ReadOnlyBoardAdapter` restituisce oggetti `ReadOnlyCell` invece di `Cell`, impedendo la fuga di riferimenti modificabili (*Cascading Proxy*).

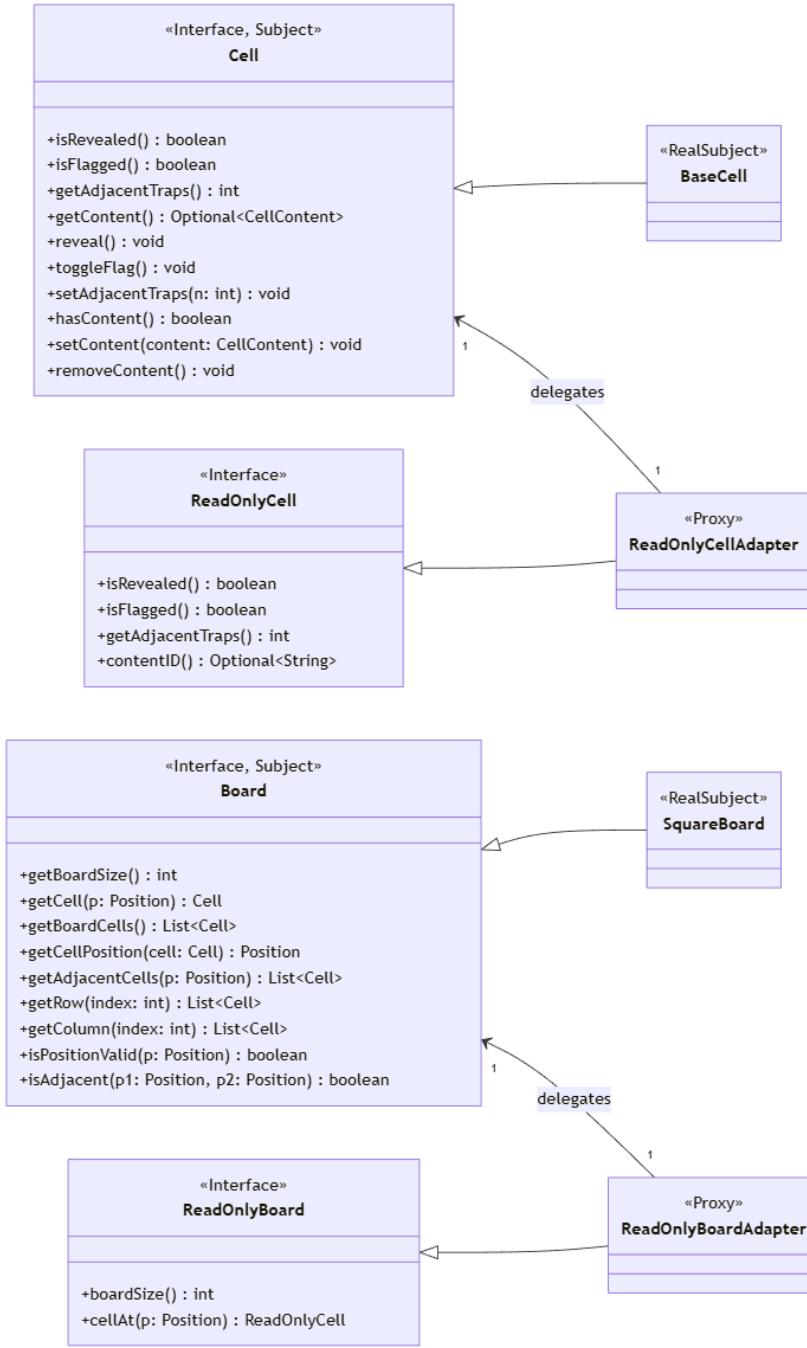


Figura 5: Rappresentazione UML del pattern Proxy per l'accesso in sola lettura alle caselle e alla griglia di gioco

Creazione dei componenti grafici

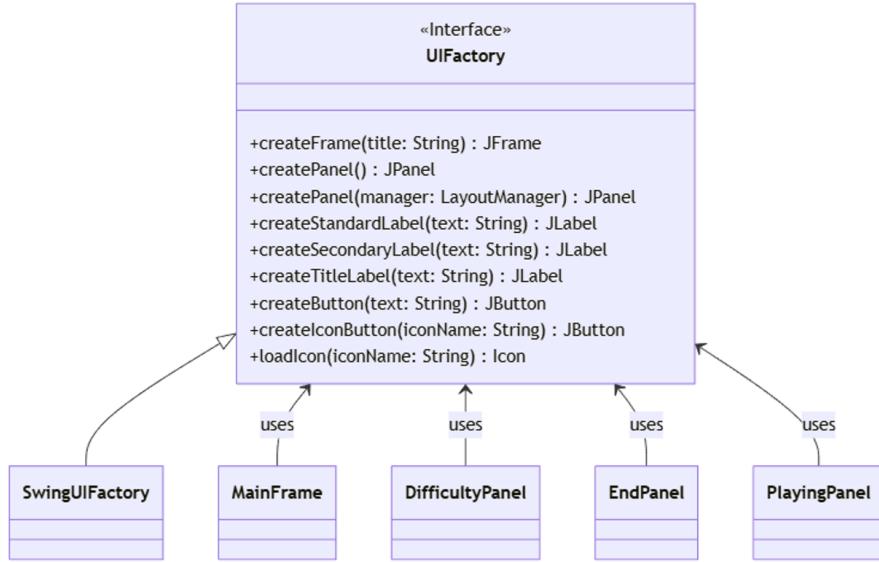


Figura 6: Rappresentazione UML del pattern Abstract Factory per la rivelazione delle caselle

Problema Si vuole uniformare la creazione dei componenti grafici e centralizzare le scelte di stile per essi, in modo da ridurre la ripetizione di codice nella creazione di ogni singolo elemento grafico.

Soluzione Si utilizza il *pattern Abstract Factory* come da figura. L’interfaccia **UIFactory** espone una serie di factory methods per creare i vari componenti grafici. Questa interfaccia può essere implementata da più classi (es: la classe **SwingUIFactory**) al fine di avere più modalità di creazione della grafica interscambiabili. Le classi **MainFrame**, **DifficultyPanel**, **EndPanel** e **PlayingPanel** usano **UIFactory** per la creazione dei loro sottocomponenti.

2.2.2 Sara Quirici

Creazione della configurazione di livello in base alla difficoltà

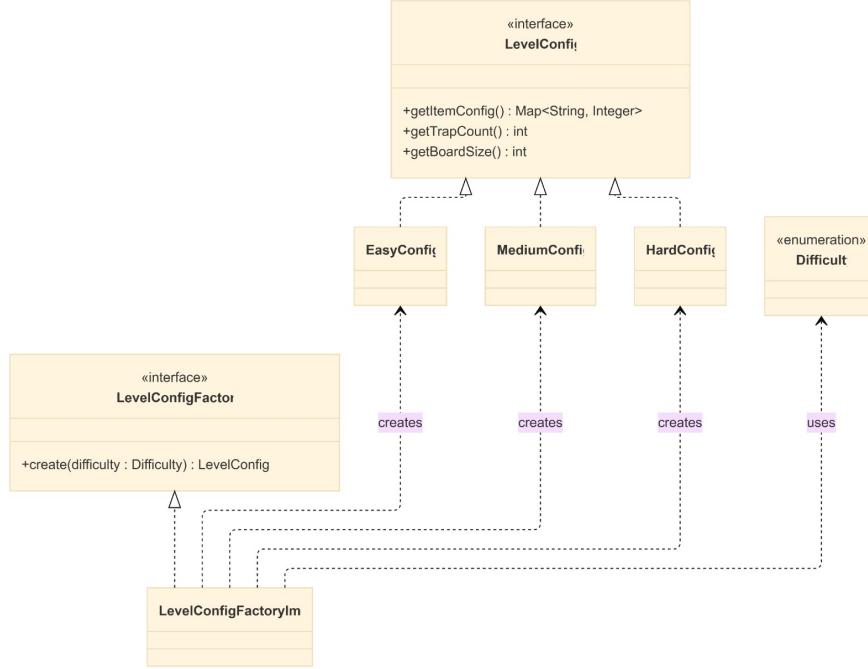


Figura 7: Rappresentazione UML dei pattern Factory Method per la creazione del livello in base alla difficoltà e Strategy per la gestione delle configurazioni di livello

Problema Il sistema deve supportare diverse modalità di configurazione del livello (**EASY**, **MEDIUM**, **HARD**), ciascuna caratterizzata da parametri differenti. È quindi necessario istanziare la configurazione che corrisponde alla difficoltà selezionata garantendo che il codice client lavori sempre con l’interfaccia comune senza conoscere le specifiche implementazioni concrete. Istanziare direttamente le classi concrete associate alle diverse difficoltà introdurrebbe un accoppiamento esplicito tra il client e le implementazioni (**EasyConfig**, **MediumConfig**, **HardConfig**) rendendo il codice meno flessibile e più difficile da estendere o modificare.

Soluzione Si adottano i *pattern Factory Method* e *Strategy*. La classe `LevelConfigFactory` espone un metodo factory che, dato un valore di tipo `Difficulty`, restituisce sempre un oggetto coerente di tipo `LevelConfig`. A seconda della difficoltà richiesta, la factory istanzia una delle implementazioni concrete (`EasyConfig`, `MediumConfig`, `HardConfig`), encapsulando la logica di selezione e creazione dell’oggetto appropriato. Il client, quindi, interagisce esclusivamente con l’interfaccia `LevelConfig`, mentre la scelta dell’implementazione concreta è delegata alla factory. Le diverse classi che implementano `LevelConfig` rappresentano una famiglia di strategie intercambiabili e il motore di gioco utilizza l’interfaccia `LevelConfig` per ottenere i parametri necessari, senza conoscere quale implementazione sia stata scelta. Un’alternativa sarebbe stata associare direttamente i parametri caratteristici di ogni configurazione all’interno dell’enumerazione `Difficulty`, oppure gestire la selezione tramite uno `switch` nel codice client. Tali soluzioni avrebbero tuttavia aumentato l’accoppiamento tra il client e le implementazioni concrete, rendendo necessarie modifiche al codice che utilizza la configurazione in caso di introduzione di una nuova difficoltà. Inoltre, inserire tutti i parametri all’interno dell’enum avrebbe violato il Single Responsibility Principle, attribuendo all’enum sia il ruolo di rappresentare il livello logico di difficoltà sia quello di contenere i dati della configurazione. La scelta di introdurre una factory consente invece di centralizzare la logica di creazione e di localizzare le modifiche, preservando l’estendibilità del sistema e mantenendo il client dipendente esclusivamente dall’astrazione `LevelConfig`.

Generazione dinamica della board in funzione della configurazione di livello

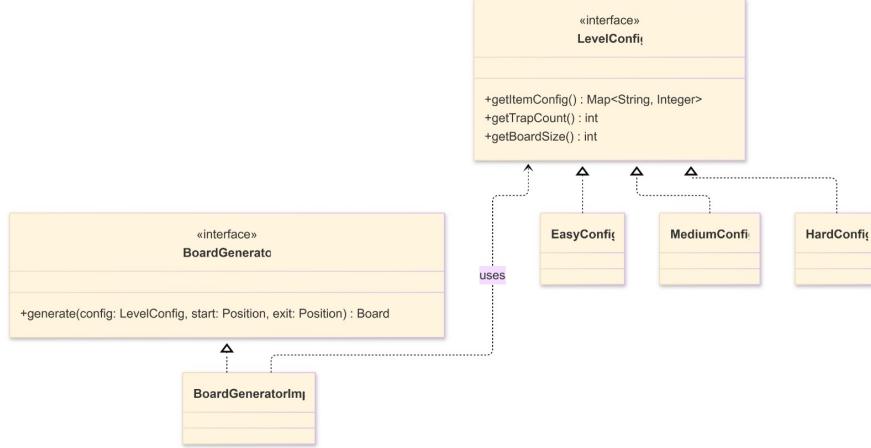


Figura 8: Rappresentazione UML della strategia utilizzata per la generazione della board di livello

Problema La generazione della griglia di gioco deve dipendere dai parametri del livello selezionato senza che la logica di generazione sia differente per ciascuna configurazione. È quindi necessario separare la definizione dei parametri del livello dall'algoritmo che costruisce la board.

Soluzione Il sistema introduce un'interfaccia **BoardGenerator** che definisce il contratto per la generazione della griglia di gioco. La classe concreta **BoardGeneratorImpl** implementa l'algoritmo effettivo di creazione della board, utilizzando i parametri forniti da un oggetto **LevelConfig**. In questo modo viene separata la definizione dei parametri del livello dall'algoritmo di generazione, garantendo disaccoppiamento tra configurazione e costruzione della board. La struttura è progettata secondo il *pattern Strategy*: l'interfaccia **BoardGenerator** rappresenta la strategia, mentre le sue implementazioni concrete incapsulano specifici algoritmi di generazione. Attualmente è presente una sola implementazione concreta; tuttavia l'architettura consente l'introduzione futura di generatori alternativi senza modificare il codice client, migliorando estendibilità e riuso.

2.2.3 Luca Galassi

Creazione della configurazione di livello in base alla difficoltà

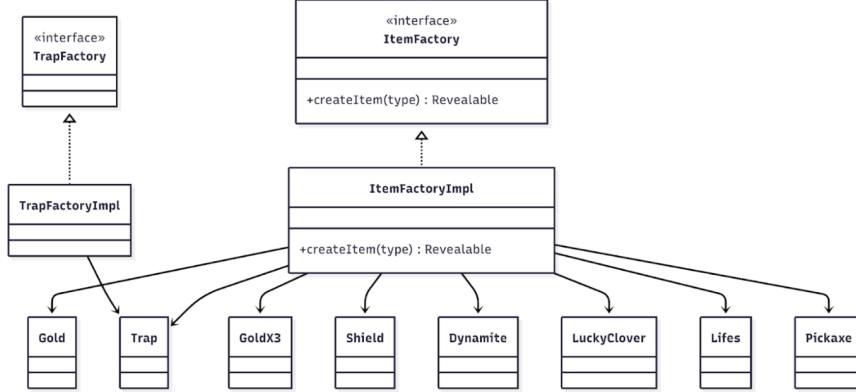


Figura 9: Rappresentazione UML per la factory di items e traps

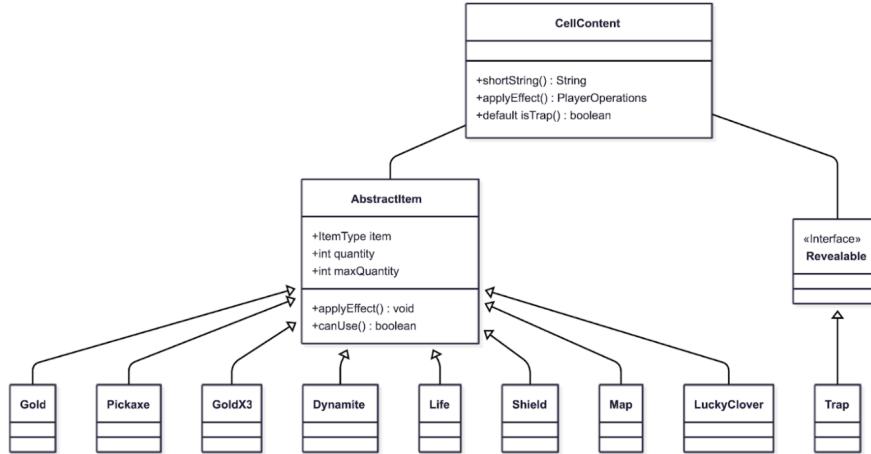


Figura 10: Rappresentazione UML del pattern Template Method

Problema Il sistema deve creare e modellare degli oggetti che interagiscono con il giocatore, la griglia e l'inventario. È importante che questi ultimi abbiano comportamenti diversi ma allo stesso tempo devono essere trattati in modo uniforme da tutto il sistema, in particolare è necessario separare la loro logica da quella del giocatore. Permettere una creazione dinamica di diversi tipi di `Item` e, nel mentre, il sistema deve essere facilmente estendibile e permettere la gestione di effetti differenti.

Soluzione *Template Method* usato nella classe `AbstractItem` che definisce una struttura base per un comportamento condiviso, iniettando tramite un record, il contesto dall'esterno e encapsulando in tutti i metodi. La `AbstractItem` è una classe astratta che definisce il comportamento comune e le costanti che tutte le classi `Item` erediteranno. In alcune classi che rappresentano gli item sono presenti algoritmi ricorsivi quali BFS e un metodo che in `Pickaxe` usa la random per rivelare e disarmare una riga casuale nella griglia. Usato anche *Strategy Template* per far sì che il metodo `applyEffect` venga implementato diversamente ogni volta da ogni singolo item, evitando così di centrare la logica su un'unica classe, violando l'*open-close principle*. *Factory Method* è stato usato per la generazione di items e traps senza conoscerne la classe concreta, disaccoppiando così il codice dal client e centralizzando il tutto, di conseguenza il client lavora direttamente con `AbstractItem`. Nel sistema i due pattern lavorano in modo complementare, la factory gestisce la creazione dell'oggetto selezionando così quale strategia istanziare, strategy invece usa l'astrazione comune per applicarne il comportamento.

2.2.4 David Andrei Orosanu

Coordinamento del flusso di gioco tramite GameSession

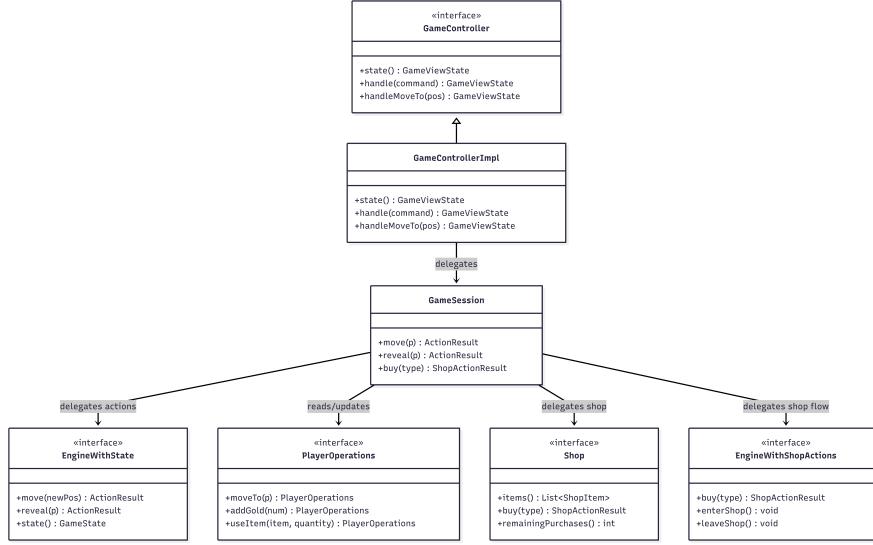


Figura 11: Rappresentazione UML del coordinamento del flusso di gioco

Problema Il controller deve poter invocare operazioni di gioco (movimento, rivelazione, flag, shop, utilizzo oggetti, avanzamento livello) senza dipendere direttamente dai dettagli interni del modello (engine/player/shop). Un accesso diretto ai sottosistemi aumenterebbe l'accoppiamento e renderebbe più fragile l'evoluzione del sistema.

Soluzione È stata introdotta la classe **GameSession** come punto di accesso principale al modello. **GameSession** coordina i sottosistemi e fornisce al controller un'interfaccia di alto livello, delegando ai componenti interni l'esecuzione delle operazioni specifiche (ad esempio a engine, player e shop). In questo modo il controller resta stabile anche in caso di modifiche interne al modello.

Pattern e principi utilizzati *Facade*: **GameSession** fornisce un'interfaccia semplificata verso un sottosistema complesso.

SRP/DIP (principi): responsabilità e dipendenze concentrate su contratti e punti d'accesso chiari.

Gestione delle azioni di gioco: risultati esplicativi e servizi dedicati

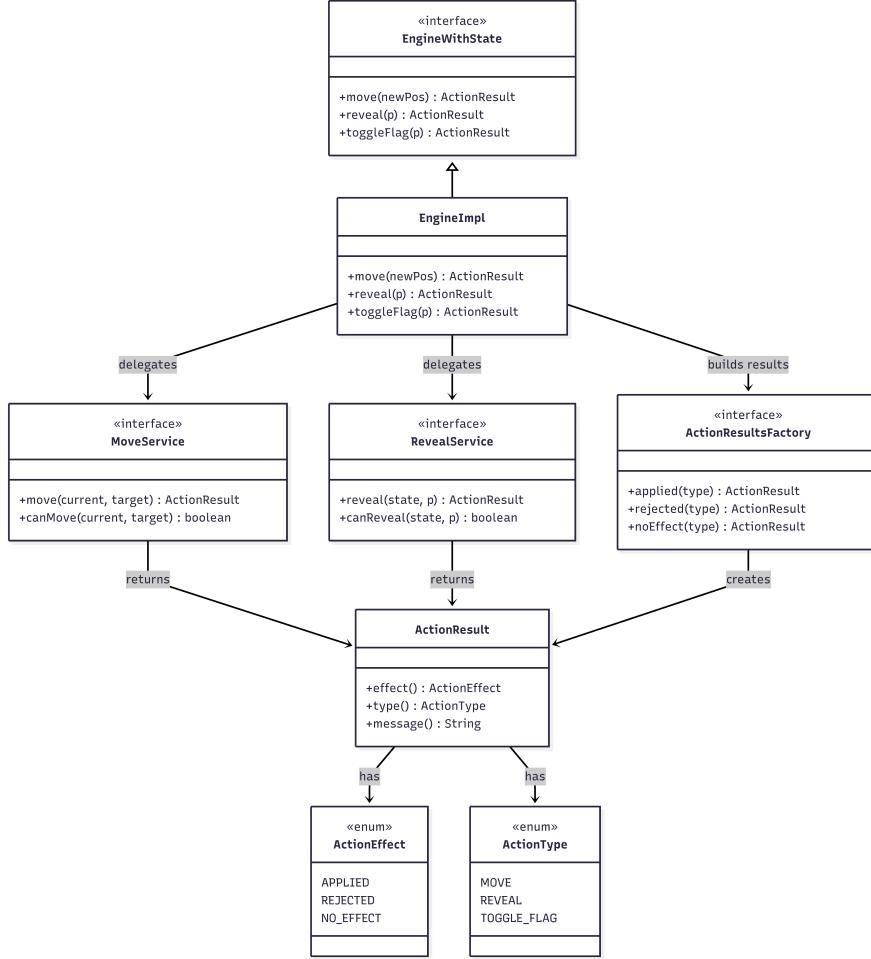


Figura 12: Rappresentazione UML della gestione delle azioni di gioco

Problema Le azioni di gioco possono fallire per motivi leciti (mossa non valida, cella già rivelata, partita terminata, ecc.). Usare eccezioni per gestire questi casi “normali” rende il flusso meno chiaro e più difficile da gestire lato controller/view. Inoltre, concentrare tutta la logica in un’unica classe (ad esempio `EngineImpl`) rischia di creare una classe troppo grande e poco testabile.

Soluzione Le operazioni del motore restituiscono un oggetto risultato (`ActionResult`) che descrive in modo esplicito l’esito dell’azione tramite `ActionEffect` e il tipo di azione tramite `ActionType`. In questo modo il controller può gestire in modo deterministico i casi (applicata / rifiutata / nessun effetto), senza eccezioni per il controllo di flusso.

Inoltre, la logica dei casi d’uso principali è stata separata in servizi dedicati: `MoveService` e `RevealService`. `EngineImpl` coordina e mantiene lo stato complessivo, mentre i servizi incapsulano le regole e i passi operativi delle singole azioni, rendendo il design più modulare e più facile da testare.

Per uniformare la creazione dei risultati e mantenere coerenza tra le varie azioni, è presente anche `ActionResultsFactory`, che centralizza la costruzione di `ActionResult`.

Pattern e principi utilizzati *Service Layer*: i servizi incapsulano casi d’uso specifici (movimento/rivelazione).

Factory: creazione standardizzata dei risultati (`ActionResultsFactory`).

Result Object (modellazione): esiti tipizzati tramite `ActionResult`/`ActionEffect`. SRP: engine coordina, servizi eseguono, factory costruisce risultati.

Stato di gioco tipizzato e regole intercambiabili

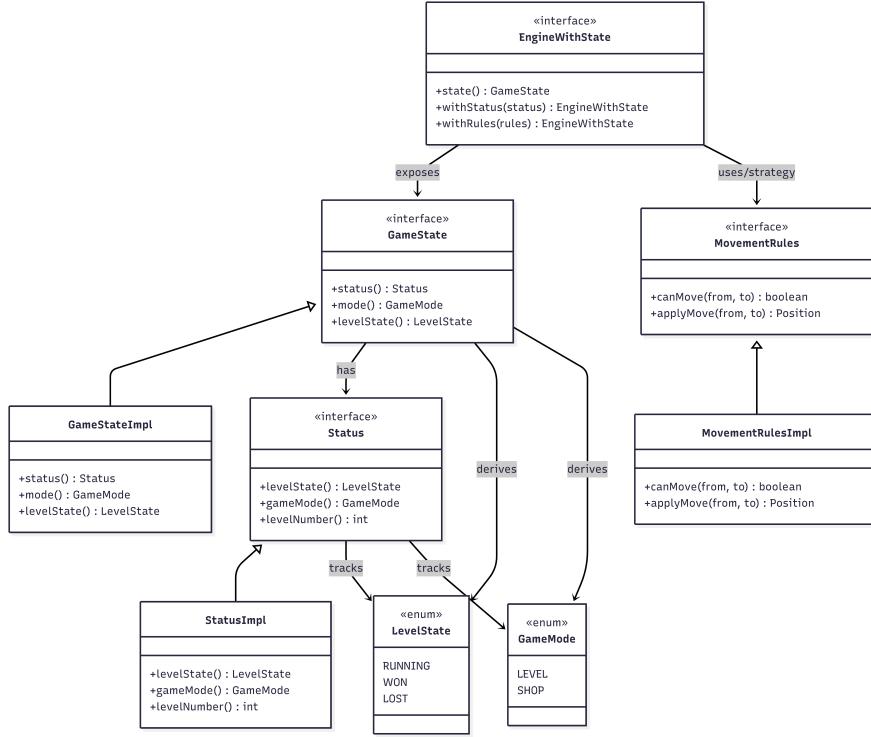


Figura 13: Rappresentazione UML dello stato di gioco e della strategia delle regole di movimento

Problema Un gioco presenta stati e modalità differenti (partita in corso/terminata, modalità di gioco, avanzamento livello, ecc.). Gestire questi aspetti con booleani sparsi o logiche implicite aumenta il rischio di incoerenze. Inoltre, alcune regole (es. movimento) devono poter essere cambiate o estese senza riscrivere la logica di alto livello.

Soluzione Lo stato è modellato in modo esplicito tramite tipi dedicati (**GameState**, **GameMode**, **LevelState**) e implementazioni come **GameStateImpl**. La classe **StatusImpl** incapsula lo “stato corrente” in forma immutabile: ogni modifica produce una nuova istanza, riducendo effetti collaterali e facilitando ragionamento e test.

Le regole di movimento sono astratte tramite l’interfaccia **MovementRules**, con implementazione **MovementRulesImpl**. Questo permette di mantenere l’engine indipendente dalle regole concrete e consente eventuali estensioni future (nuove regole o modalità).

Pattern e principi utilizzati *State modeling* (modellazione): stati e modalità esplicativi (**GameState**, **LevelState**, **GameMode**). *Immutable Value Object*: **StatusImpl** come snapshot immutabile dello stato corrente. *Strategy*: **MovementRules** come regole sostituibili. OCP (principio): nuove regole/stati possono essere aggiunti senza riscrivere l’uso ad alto livello.

Controller UI, comandi e snapshot immutabili per la View

Problema La UI Swing deve poter inviare interazioni in modo uniforme e ricevere uno stato renderizzabile senza accedere direttamente al modello. Inoltre, la view deve rimanere “passiva”: nessuna business logic e nessuna conoscenza dei dettagli del model, ma solo rendering.

Soluzione È stato definito un controller UI-facing (`GameController` / `GameControllerImpl`) che gestisce le richieste della GUI e pubblica snapshot immutabili (`GameViewState`). Le interazioni della UI sono rappresentate tramite `GuiCommand` (con implementazioni record come `MoveTo`, `Reveal`, `ToggleFlag`, `Buy`, `UseItem`, `Continue`, `LeaveToMenu`, `StartGame`, `SetDifficulty`), in modo da incapsulare ogni azione utente e applicarla polimorficamente al controller.

La conversione tra stato del modello (`GameSession`) e stato della view è delegata a `ViewStateMapper` / `ViewStateMapperImpl`, che produce i record di viewstate (`GameViewState`, `HudViewState`, `InventoryViewState`, `ShopViewState`, `CellViewState`, ecc.). Questo garantisce una separazione netta: la view Swing renderizza solo dati derivati e immutabili, senza mai manipolare il modello.

La classe `SwingGameView` si limita a collegare gli eventi dei pannelli (`MenuPanel`, `DifficultyPanel`, `PlayingPanel`, `ShopPanel`, `EndPanel`) all’invio di comandi al controller, e a renderizzare lo stato risultante.

Pattern e principi utilizzati *Command*: `GuiCommand` incapsula azioni utente come oggetti applicabili al controller.

Mapper: `ViewStateMapperImpl` traduce model → viewstate.

Immutable Snapshot: i record viewstate sono dati immutabili consumati dalla view.

SRP: view = rendering, controller = coordinamento UI, mapper = trasformazione dati.

Null Object (micro-pattern): listener di default no-op in pannelli (es. `InventoryPanel`, `ShopPanel`, `SwingGameView`) per evitare null e semplificare la gestione eventi.

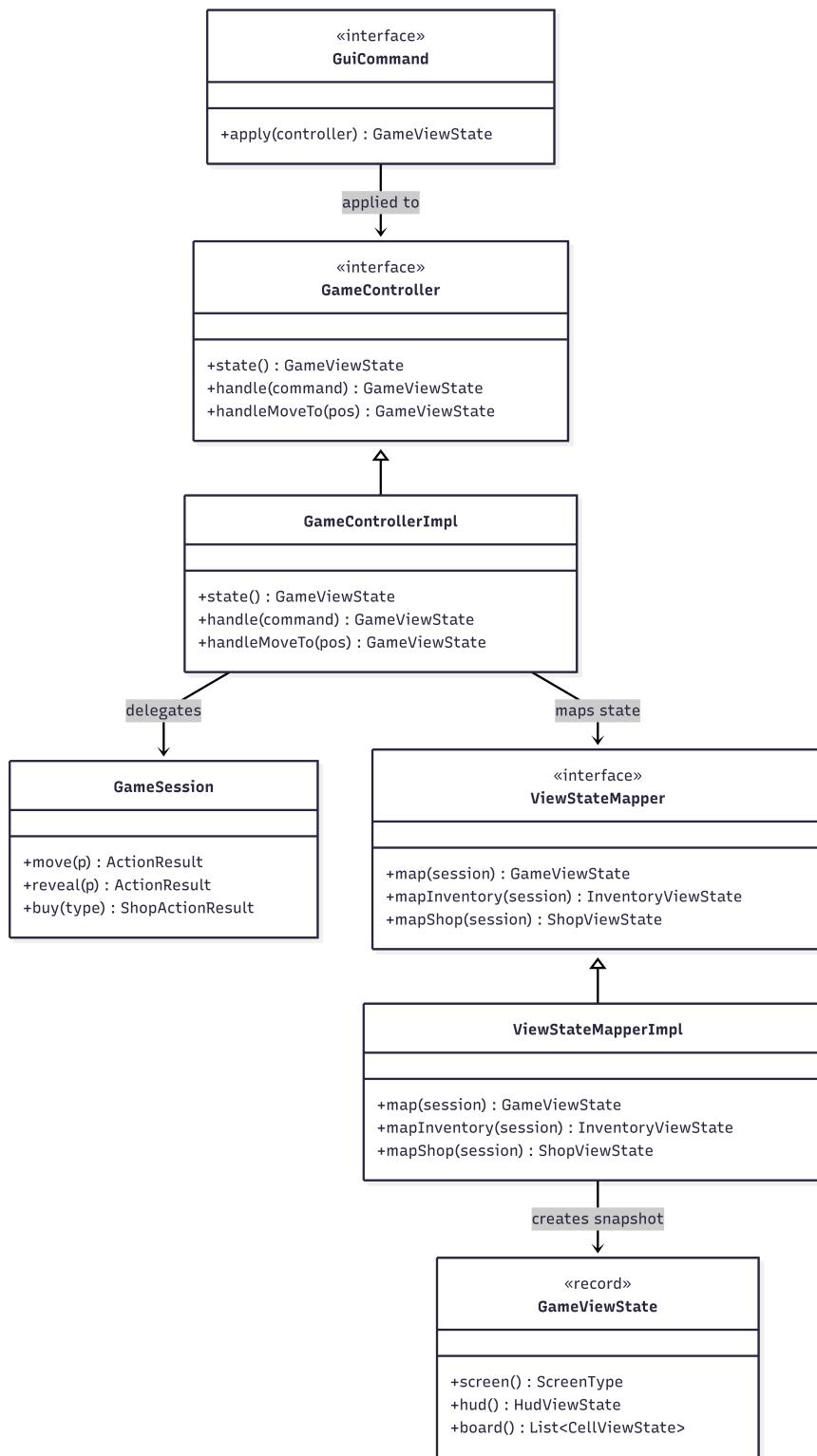


Figura 14: Rappresentazione UML del Controller UI, dei comandi e degli snapshot immutabili per la View

Sottosistema shop e motivazioni esplicite di fallimento

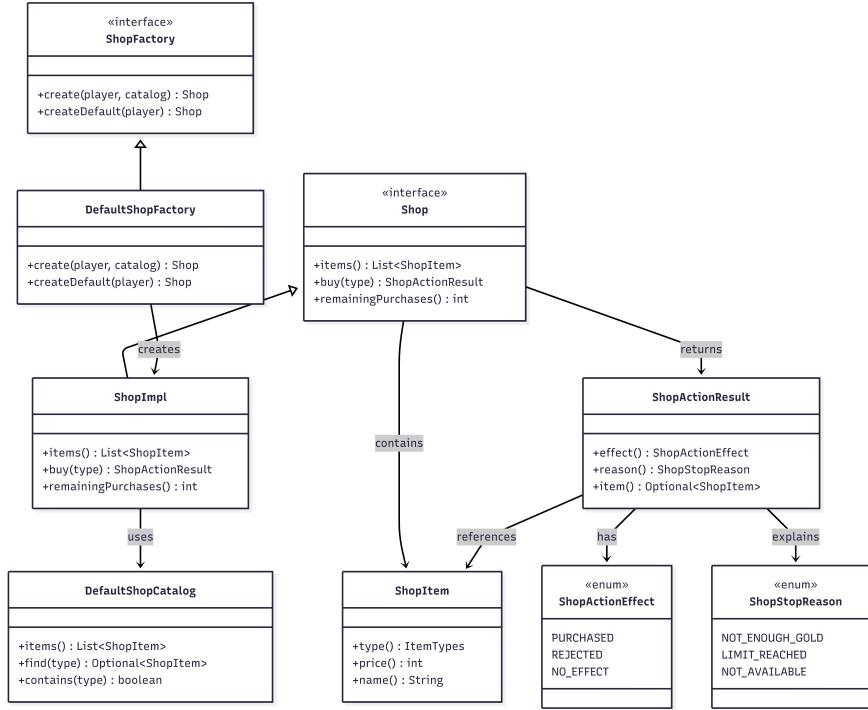


Figura 15: Rappresentazione UML del sottosistema Shop con risultati tipizzati e factory di creazione

Problema Lo shop deve gestire acquisti con vincoli e comunicare chiaramente l'esito delle operazioni (successo, limite raggiunto, oro insufficiente, item non venduto, ecc.).

Soluzione Lo shop usa esiti tipizzati (ShopActionResult, ShopActionEffect) e motivazioni esplicite (ShopStopReason). La costruzione è isolata tramite ShopFactory/DefaultShopFactory, mentre l'offerta è separata nel DefaultShopCatalog. La view non interpreta direttamente la logica: il mapper produce ShopViewState e messaggi utente a partire dai risultati.

Pattern e principi utilizzati *Factory*: ShopFactory/DefaultShopFactory.
Result Object: ShopActionResult/ShopActionEffect.

Domain-rich enum: ShopStopReason per motivazioni esplicite.
 SRP: catalogo e logica shop separati.

3 Sviluppo

3.1 Testing automatizzato

Le classi di testing sono state realizzate con JUnit ed eseguite tramite Gradle. Per ogni classe di implementazione sono stati creati metodi che testassero tutte le possibili sfaccettature di comportamento della classe.

- **BaseCellTest:** testa le funzionalità di rivelazione e segnalazione della casella, il suo corretto contenuto e il calcolo e l'impostazione del numero di trappole adiacenti.
- **BaseCellFactoryTest:** testa che la creazione delle caselle avvenga con i parametri giusti.
- **SquareBoardTest:** testa l'inizializzazione delle caselle della griglia di gioco, la correttezza dei collegamenti tra casella e posizione nella griglia, il calcolo delle adiacenze e i suoi metodi di utilità.
- **SquareBoardFactoryTest:** testa la creazione della griglia di gioco.
- **SimpleRevealTest:** testa la rivelazione semplice, una casella alla volta, nella griglia di gioco.
- **FloodRevealTest:** testa la rivelazione a cascata delle caselle senza trappole adiacenti nella griglia di gioco.
- **ReadOnlyCellAdapter:** testa la corretta delegazione di `ReadOnlyCell` a `Cell`.
- **ReadOnlyBoardAdapter:** testa la corretta delegazione di `ReadOnlyBoard` a `Board` e a `Cell`.
- **BoardGeneratorImplTest:** testa che la generazione della `Board` produca una griglia coerente con la configurazione del livello, con dimensioni e quantità di elementi corrette.
- **EasyConfigTest:** testa che la configurazione del livello facile esponga correttamente dimensioni, numero di trappole e quantità degli oggetti previsti.
- **MediumConfigTest:** testa che la configurazione del livello medio restituisca parametri coerenti con le specifiche di difficoltà intermedia.
- **HardConfigTest:** testa che la configurazione del livello difficile definisca correttamente dimensioni elevate e quantità adeguate di trappole e oggetti.
- **LevelConfigFactoryImplTest:** testa che la factory crei l'istanza corretta di configurazione in base alla difficoltà richiesta.
- **LevelImplTest:** verifica che l'inizializzazione del livello costruisca correttamente la board e imposti lo stato iniziale del gioco in modo coerente.
- **GameStateTest:** verifica la corretta creazione e immutabilità dello stato di gioco (`GameState`), assicurando che tutte le dipendenze siano valide e correttamente esposte.
- **RevealServiceTest:** verifica la logica di rivelazione e gestione delle flag, controllando che le operazioni rispettino lo stato del gioco e le condizioni della board.
- **EngineTest:** verifica il corretto coordinamento delle operazioni principali del motore (`EngineImpl`), inclusi movimento, rivelazione, gestione dello shop e aggiornamento dello stato.
- **ActionResultsFactoryTest:** verifica la corretta costruzione degli oggetti `ActionResult`, assicurando che rappresentino in modo coerente l'esito e il tipo delle azioni di gioco.
- **StatusTest:** verifica la correttezza e l'immutabilità della classe `StatusImpl`, assicurando la validità delle transizioni di stato e dei relativi vincoli.
- **MovementRulesTest:** verifica la correttezza delle regole di movimento (`MovementRulesImpl`), inclusa la validazione dei percorsi e la raggiungibilità delle posizioni.

- **MoveServiceTest:** verifica la logica del servizio di movimento (`MoveService`), assicurando il corretto aggiornamento della posizione del giocatore e la gestione dei casi non validi.
- **ShopTest:** verifica il funzionamento del sottosistema shop (`ShopImpl`), inclusi i vincoli sugli acquisti, la gestione dell'oro e il rispetto dei limiti di acquisto.
- **InventoryTest:** verifica la gestione dell'inventario, assicurando la corretta aggiunta, rimozione e verifica degli oggetti nel rispetto dei vincoli e dell'immutabilità.
- **PlayerTest:** verifica il comportamento della classe `Player`, assicurando la corretta gestione di posizione, vite, oro e inventario.
- **GameFactoryTest:** verifica la corretta creazione di una nuova sessione di gioco (`GameSession`) e l'inizializzazione coerente dei suoi sottosistemi.
- **GameSessionTest:** verifica che `GameSession` coordini correttamente le operazioni di gioco, restituendo risultati coerenti per movimento, rivelazione e gestione delle flag.

3.2 Note di sviluppo

3.2.1 Azzurra Quattrini

Utilizzo di lambda expressions e method references

Usate spesso nelle classi di testing. Permalink a un esempio: <https://github.com/orodav/OOP25-Gold-Hunt/blob/27e81c968dc15b6fe6c6be5ed8363929b4e4e544/src/main/java/it/unibo/goldhunt/board/impl/FloodReveal.java#L34-L40>

Utilizzo di Stream

Permalink ad alcuni esempi:

<https://github.com/orodav/OOP25-Gold-Hunt/blob/27e81c968dc15b6fe6c6be5ed8363929b4e4e544/src/main/java/it/unibo/goldhunt/board/impl/SquareBoard.java#L58-L62>

<https://github.com/orodav/OOP25-Gold-Hunt/blob/27e81c968dc15b6fe6c6be5ed8363929b4e4e544/src/main/java/it/unibo/goldhunt/board/impl/SquareBoard.java#L117-L123>

Utilizzo di Optional

Usato per rappresentare l'idea di una casella che può avere o non avere contenuto. Permalink: <https://github.com/orodav/OOP25-Gold-Hunt/blob/27e81c968dc15b6fe6c6be5ed8363929b4e4e544/src/main/java/it/unibo/goldhunt/board/impl/BaseCell.java#L100-L130>

Utilizzo di switch expressions

Permalink: <https://github.com/orodav/OOP25-Gold-Hunt/blob/27e81c968dc15b6fe6c6be5ed8363929b4e4e544/src/main/java/it/unibo/goldhunt/view/swing/components/CellButton.java#L206-L223>

3.2.2 Sara Quirici

Utilizzo di lambda expressions

Uso di lambda expressions sia nella logica di test sia nella gestione di eventi nella view (ActionListener nei pannelli Swing). Permalink ad alcuni esempi: <https://github.com/orodav/OOP25-Gold-Hunt/blob/1ca130fc82f076f9f29d1145b3314279644105a7/src/test/java/it/unibo/goldhunt/configuration/impl/BoardGeneratorImplTest.java#L107-L112>
<https://github.com/orodav/OOP25-Gold-Hunt/blob/1ca130fc82f076f9f29d1145b3314279644105a7/src/main/java/it/unibo/goldhunt/view/swing/screens/DifficultyPanel.java#L68-L70>

Utilizzo di Stream API

Uso di Stream nella logica di test. Permalink a un esempio: <https://github.com/orodav/OOP25-Gold-Hunt/blob/1ca130fc82f076f9f29d1145b3314279644105a7/src/test/java/it/unibo/goldhunt/configuration/impl/BoardGeneratorImplTest.java#L80-L88>

Utilizzo di method references

Uso di method reference in combinazione con Stream API. Permalink a un esempio: <https://github.com/orodav/OOP25-Gold-Hunt/blob/1ca130fc82f076f9f29d1145b3314279644105a7/src/test/java/it/unibo/goldhunt/configuration/impl/BoardGeneratorImplTest.java#L100-L105>

Utilizzo di switch expressions

Uso di switch expression con sintassi case -> per implementare in modo conciso e sicuro la creazione delle configurazioni in base alla difficoltà. Permalink: <https://github.com/orodav/OOP25-Gold-Hunt/blob/1ca130fc82f076f9f29d1145b3314279644105a7/src/main/java/it/unibo/goldhunt/configuration/impl/LevelConfigFactoryImpl.java#L20-L27>

BFS implementata manualmente

Permalink: <https://github.com/orodav/OOP25-Gold-Hunt/blob/1ca130fc82f076f9f29d1145b3314279644105a7/src/main/java/it/unibo/goldhunt/configuration/impl/BoardGeneratorImpl.java#L153C5-L205C6>

3.2.3 Luca Galassi

DFS ricorsiva implementata

Permalink: <https://github.com/orodav/OOP25-Gold-Hunt/blob/21a8d58068cc5e1ba9dfd528d32e8b9a668c2a2e/src/main/java/it/unibo/goldhunt/items/impl/Chart.java#L52>

Uso di metodi default nelle API successivamente overrideati

Permalink: <https://github.com/orodav/OOP25-Gold-Hunt/blob/21a8d58068cc5e1ba9dfd528d32e8b9a668c2a2e/src/main/java/it/unibo/goldhunt/items/api/ClearCells.java#L26>

ItemFactory e TrapFactory (*Factory method*) Permalink: <https://github.com/orodav/OOP25-Gold-Hunt/blob/21a8d58068cc5e1ba9dfd528d32e8b9a668c2a2e/src/main/java/it/unibo/goldhunt/items/impl/>

ItemFactoryImpl.java#L20

Uso di lambda e Stream

Permalink: <https://github.com/orodav/OOP25-Gold-Hunt/blob/21a8d58068cc5e1ba9dfd528d32e8b9a668c2a2e/src/main/java/it/unibo/goldhunt/items/api/ClearCells.java#L26>

Utilizzo della libreria javax.swing.plaf.DimensionUIResource

Permalink: <https://github.com/orodav/OOP25-Gold-Hunt/blob/21a8d58068cc5e1ba9dfd528d32e8b9a668c2a2e/src/main/java/it/unibo/goldhunt/view/swing/components/LegendPanel.java#L14>

3.2.4 David Andrei Orosanu

Gestione esplicita Optional nello shop engine

Permalink: <https://github.com/orodav/OOP25-Gold-Hunt/blob/21a8d58068cc5e1ba9dfd528d32e8b9a668c2a2e/src/main/java/it/unibo/goldhunt/root/GameSession.java#L176>

<https://github.com/orodav/OOP25-Gold-Hunt/blob/21a8d58068cc5e1ba9dfd528d32e8b9a668c2a2e/src/main/java/it/unibo/goldhunt/view/viewstate/GameViewState.java#L34>

Record immutabile per snapshot della View

Permalink: <https://github.com/orodav/OOP25-Gold-Hunt/blob/21a8d58068cc5e1ba9dfd528d32e8b9a668c2a2e/src/main/java/it/unibo/goldhunt/view/viewstate/GameViewState.java#L34>

Uso di lambda e Stream per mapping della Board

Permalink: <https://github.com/orodav/OOP25-Gold-Hunt/blob/21a8d58068cc5e1ba9dfd528d32e8b9a668c2a2e/src/main/java/it/unibo/goldhunt/view/impl/ViewStateMapperImpl.java#L142>

Null-object listener

Permalink: <https://github.com/orodav/OOP25-Gold-Hunt/blob/21a8d58068cc5e1ba9dfd528d32e8b9a668c2a2e/src/main/java/it/unibo/goldhunt/view/swing/components/InventoryPanel.java#L28>

Separazione della logica tramite service layer

Permalink: <https://github.com/orodav/OOP25-Gold-Hunt/blob/21a8d58068cc5e1ba9dfd528d32e8b9a668c2a2e/src/main/java/it/unibo/goldhunt/engine/impl/EngineImpl.java#L73>

4 Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Azzurra Quattrini

Il mio ruolo in questo progetto è stato occuparmi della struttura della griglia di gioco, delle sue caselle e della gestione dei loro stati e condizioni.

Ritengo che il mio maggiore punto di forza in questo progetto sia stata la parte di modellazione. Ho voluto ragionare sin da subito concentrandomi sui pattern architettonici e ho cercato di porre importanza al non sforare nelle responsabilità dei miei compagni e al progettare tenendo conto di possibili future modifiche ed estensioni del lavoro svolto. Ho notato di essere riuscita nel mio intento perché, anche nelle fasi più critiche del progetto, non ho mai dovuto modificare nulla del mio Model, indice del fatto che sia riuscita a dargli una base solida. Inoltre, nel corso del progetto ho preso sempre più dimestichezza con l'utilizzo di GitHub: nonostante all'inizio fosse una delle parti in cui temevo di avere più difficoltà, grazie a questo progetto ne ho capito a pieno l'uso e le potenzialità e ho aiutato spesso i miei compagni di gruppo a risolvere i loro dubbi su Git.

Mi sono sentita meno forte nel collegamento tra Model, View e Controller, soprattutto per il fatto di non essere stata io a scrivere i sorgenti più rilevanti del Controller: ho notato che faccio fatica a immedesimarmi nei modi di ragionare delle altre persone, motivo per cui ho avuto qualche difficoltà a comprendere i sorgenti scritti dai miei compagni di gruppo. Il difetto più grande di questo progetto è stata sicuramente l'organizzazione del tempo: non siamo riusciti a gestire bene il tempo a disposizione e la realizzazione del progetto è stata una corsa contro il tempo. Proprio per questo non mi reputo completamente soddisfatta del risultato finale: abbiamo fatto il possibile per il tempo a disposizione che ci siamo dati ma, per quanto io abbia reso la mia logica aperta all'estensione, non c'è stato il tempo materiale per realizzare una buona parte delle funzionalità opzionali o anche solo per rifinire la grafica.

In ogni caso, sento che realizzare questo progetto sia stato molto formativo e custodirò ciò che ho imparato per i prossimi lavori futuri.

4.1.2 Sara Quirici

Il mio ruolo all'interno del gruppo ha riguardato la progettazione e l'implementazione della configurazione iniziale del gioco. Mi sono quindi occupata della gestione delle diverse configurazioni di livello in base alle tre difficoltà previste e dell'implementazione dell'algoritmo di generazione della board. Al termine della fase di sviluppo della logica di gioco, ho realizzato i test automatici relativi al mio sottoinsieme. Inoltre, ho contribuito alla realizzazione di alcune schermate della view.

Ritengo che il mio sottoinsieme di model sia stato sviluppato in maniera ordinata e coerente, con particolare attenzione alla separazione delle responsabilità. L'aspetto di cui mi ritengo maggiormente soddisfatta è l'implementazione della generazione della board iniziale, che ha rappresentato la parte più difficile del mio contributo. Durante il corso del progetto è stata revisionata più volte, soprattutto a causa di problematiche legate alla presenza di comportamenti non deterministici, ma la versione finale è risultata solida e affidabile.

Al contrario, come punto di debolezza riconosco uno scarso utilizzo di feature avanzate del linguaggio. Ho infatti la sensazione di aver programmato in maniera semplicistica, privilegiando soluzioni semplici e lineari e facendo uso delle funzionalità più avanzate solo quando strettamente necessario. Ritengo che avrei potuto sperimentare maggiormente con strumenti e costrutti più evoluti, approfondendo ulteriormente le potenzialità offerte dal linguaggio. Considero tuttavia questa consapevolezza un elemento di crescita, che mi permetterà di affrontare la progettazione con maggiore sicurezza e con un po' più di ambizione nella scelta degli strumenti a mia disposizione.

4.1.3 Luca Galassi

Nel progetto mi sono occupato della creazione degli oggetti, dell'applicazione degli effetti, dell'iniettabilità di board, player e inventory nel contesto degli oggetti. Il mio obiettivo principale era far sì che gli oggetti rispettassero OCP, DRY e KISS. Le classi di implementazione puntano a rispettare l'SRP assieme alle interfacce. Sono state cambiate un pochino di volte delle cose ma sono sempre state risolte bene. Difficoltà ce ne sono state ma tutto sommato abbiamo lavorato bene. Sono abbastanza soddisfatto di come abbiamo affrontato le cose.

Lavori futuri: al momento non ne ho ma vorrei provare a fare un qualcosa di simile a Terraria o Minecraft, o usare anche motori grafici per migliorare l'esperienza di gioco.

4.1.4 David Andrei Orosanu

Nel progetto mi sono occupato della progettazione e implementazione del motore di gioco (package engine), della gestione del giocatore (player), del sottosistema dello shop (shop) e del coordinamento generale tramite la classe `GameSession` nel package root. Inoltre, ho progettato e implementato il controller del pattern MVC, definendo l'interfaccia `GameController`, la sua implementazione (`GameControllerImpl`) e il sistema di comandi `GuiCommand`, che consente di rappresentare in modo uniforme le interazioni utente.

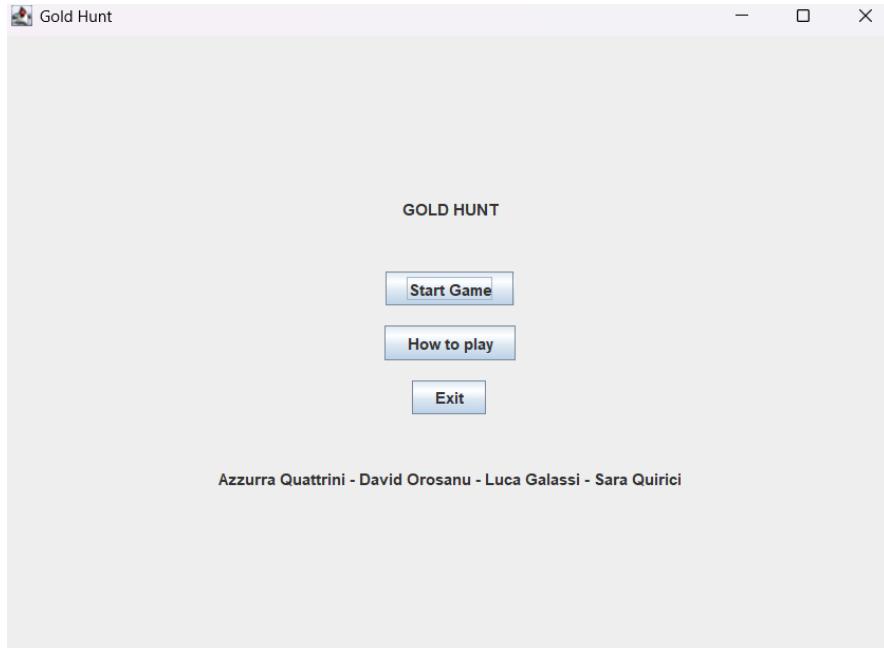
Uno degli aspetti principali su cui mi sono concentrato è stato mantenere una chiara separazione tra model, controller e view. In questo contesto, `GameSession` svolge il ruolo di punto di accesso centrale al modello, permettendo al controller di operare senza dipendere direttamente dai dettagli interni dei sottosistemi. Questo ha contribuito a mantenere il codice più modulare, ordinato ed estendibile.

Ritengo particolarmente efficace la separazione delle operazioni in servizi dedicati, come `MoveService` e `RevealService`, e l'utilizzo di oggetti risultato tipizzati (`ActionResult` e `ShopActionResult`), che rendono esplicito l'esito delle operazioni e migliorano la chiarezza del flusso di gioco.

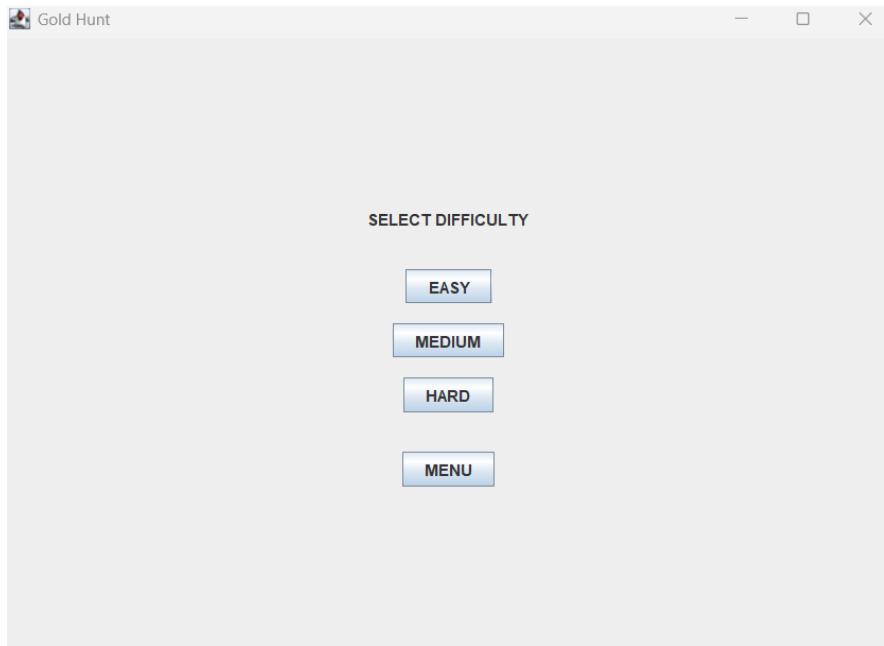
Questa esperienza è stata impegnativa ma estremamente stimolante. Mi ha permesso di comprendere meglio cosa significhi progettare un sistema software mantenibile e coerente, e di vedere concretamente come le scelte architettoniche influenzino l'intero sistema. È stato molto soddisfacente vedere il progetto crescere gradualmente fino a diventare una struttura solida e funzionante. Nel complesso, considero questa esperienza molto formativa e coinvolgente, e ritengo che rappresenti il primo passo importante nel mio percorso di crescita come sviluppatore.

A Guida utente

All'avvio del gioco ci si troverà di fronte a questa schermata.



- **Start Game:** porta alla scelta della difficoltà del livello.
- **How to play:** mostra una breve spiegazione del gioco.
- **Exit:** esce dal gioco.



- **EASY / MEDIUM / HARD:** scelta della difficoltà.
- **MENU:** torna alla schermata iniziale.



- **Inventory:** cliccare col tasto sinistro per utilizzare gli oggetti dell'inventario.
- **Griglia di gioco:** cliccare col tasto sinistro per rilevare la casella, cliccare col tasto destro per contrassegnarla.
- **Legend:** informazioni sulle icone della mappa.
- In basso a sinistra è visibile il numero di vite e di oro rimasto. In basso a destra il tasto **MENU** riporta alla schermata principale.



Cliccando sui pulsanti dello shop, se si ha abbastanza oro, si possono comprare degli oggetti da usare nel livello successivo. Cliccando sul pulsante **Leave Shop** si prosegue nella selezione della difficoltà del livello.

B Esercitazioni di laboratorio

B.1 davidandrei.orosanu@studio.unibo.it

- Laboratorio 07: <https://virtuale.unibo.it/mod/forum/discuss.php?d=207193#p285015>
- Laboratorio 08: <https://virtuale.unibo.it/mod/forum/discuss.php?d=207921#p286179>
- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=208718#p287154>
- Laboratorio 10: <https://virtuale.unibo.it/mod/forum/discuss.php?d=209589#p288455>