

Abyss Climber

Relazione di Progetto

Matteo Morellini

Artur Turchyn

Andrea Ciani

Gennaio 2026

Indice

1	Analisi	4
1.1	Descrizione e requisiti	4
1.2	Modello del dominio	6
2	Design	7
2.1	Architettura	7
2.2	Design dettagliato	9
2.3	Design di Andrea Ciani	9
2.4	Design di Artur Turchyn	12
2.4.1	Gestione del turno di combattimento (Pattern ECB) .	13
2.4.2	Sistema di caricamento delle mosse nemiche	14
2.5	Design di Matteo Morellini	16
2.5.1	Separazione MVC e ruolo dei servizi applicativi	17
2.5.2	Navigazione centralizzata: SceneRouter e SceneId . .	17
2.5.3	Gestione dello stato della run: GameState e reset senza residui	18
2.5.4	Gestione asset: AssetManager con caching e fallback .	20
2.5.5	Scelte guidate: creazione personaggio, mosse e selezio- ne porte	20
2.5.6	Contesto stanze: RoomContext e caching per piano . .	21
2.5.7	Fine partita: schermate <i>Game Over</i> e <i>Win</i> con anima- zioni	21
3	Sviluppo	22
3.1	Testing automatizzato (JUnit + Mockito)	22

3.2	Note di sviluppo	24
3.2.1	Andrea Ciani	24
3.2.2	Artur Turchyn	24
3.2.3	Matteo Morellini	25
3.3	Problemi affrontati durante lo sviluppo	26
3.4	Valutazione complessiva dello sviluppo	26
4	Commenti finali	27
4.1	Autovalutazione e lavori futuri	27
4.1.1	Andrea Ciani	27
4.1.2	Artur Turchyn	28
4.1.3	Matteo Morellini	29
4.2	Difficoltà incontrate e commenti per i docenti	30
4.3	Riflessioni generali sul progetto	31
4.4	Proposte di estensione future	32
4.5	Conclusione	33
5	Appendice	33
5.1	Guida utente	33

1 Analisi

1.1 Descrizione e requisiti

Il progetto *Abyss Climber* consiste nello sviluppo di un videogioco single-player di genere roguelike con la possibilità di migliorare le proprie statistiche durante la partita. Il roguelike è un genere caratterizzato dall'esplorazione di un dungeon attraverso livelli generati proceduralmente, gameplay a turni e morte permanente del personaggio giocante. Il giocatore esplora una serie di piani generati progressivamente, potendo scegliere a ogni piano in base alle porte generate se combattere nemici base, accedere allo shop per comprare oggetti oppure affrontare direttamente il boss che protegge l'accesso al piano successivo, la cui porta è sempre presente. L'obiettivo finale del gioco è sconfiggere il boss dell'ultimo livello, completando così l'intera scalata dell'abisso.

Ogni partita rappresenta un'esperienza autonoma, dovuta alla randomicità: il giocatore parte da uno stato iniziale definito tramite una fase di creazione del personaggio, durante la quale sceglie il proprio elemento, la classe, la difficoltà del gioco e un set di abilità. Durante l'esperienza di gioco, il giocatore può ottenere oggetti casuali che modificano i propri attributi, affrontare combattimenti a turni e interagire con negozi per migliorare il proprio equipaggiamento e aumentare le probabilità di vittoria.

Il sistema di gioco è progettato per incoraggiare la rigiocabilità grazie alla presenza di scelte strategiche (selezione delle abilità, percorso tra le stanze),

alla randomicità e alla scelta iniziale della difficoltà.

Requisiti funzionali

- Il sistema deve fornire un'interfaccia grafica che consenta al giocatore di interagire con il gioco, includendo un menu principale e le schermate di gioco principali.
- Il giocatore deve poter creare il proprio personaggio selezionando una classe, la difficoltà e un elemento di appartenenza, che determina a quale tipo di abilità nemiche è più vulnerabile.
- Il sistema deve gestire combattimenti a turni tra giocatore e nemici, includendo l'uso di abilità e una logica di combattimento.
- Il gioco deve includere un sistema di progressione del giocatore basato su attributi modificabili tramite oggetti ottenuti durante la partita.
- Il sistema deve offrire diverse tipologie di stanze, tra cui combattimento, negozio e combattimento elite.
- La difficoltà della partita deve poter essere selezionata a inizio gioco. Inoltre, a ogni piano superato, i nemici diventano più forti grazie a un moltiplicatore.

Requisiti non funzionali

- Il sistema deve garantire una chiara separazione tra logica di gioco e interfaccia utente.
- Il software deve essere facilmente estendibile, permettendo l'aggiunta di nuove abilità, oggetti o tipologie di stanze.
- L'interfaccia grafica deve essere intuitiva e comprensibile anche per utenti che non hanno mai utilizzato l'applicazione.

1.2 Modello del dominio

L'idea alla base di *Abyss Climber* è quella di un roguelike a progressione: il giocatore controlla un personaggio che attraversa una sequenza di piani, alternando fasi di combattimento e fasi di gestione risorse. Ad ogni piano il gioco presenta una scelta tra eventi principali (combattimento o negozio): nel **combattimento** il player affronta un nemico in un sistema a turni, utilizzando mosse con costo e danno; al termine del turno il sistema aggiorna lo stato dello scontro e la UI riflette i cambiamenti (vita, esito dell'azione, ecc.). Nel **negozio** il giocatore può spendere l'oro accumulato per acquistare oggetti, che modificano le statistiche e forniscono vantaggi persistenti nella run. Superato un piano, la run prosegue al piano successivo, fino a raggiungere la conclusione (vittoria o sconfitta), mantenendo l'obiettivo di offrire rigiocabilità tramite scelte e progressione.

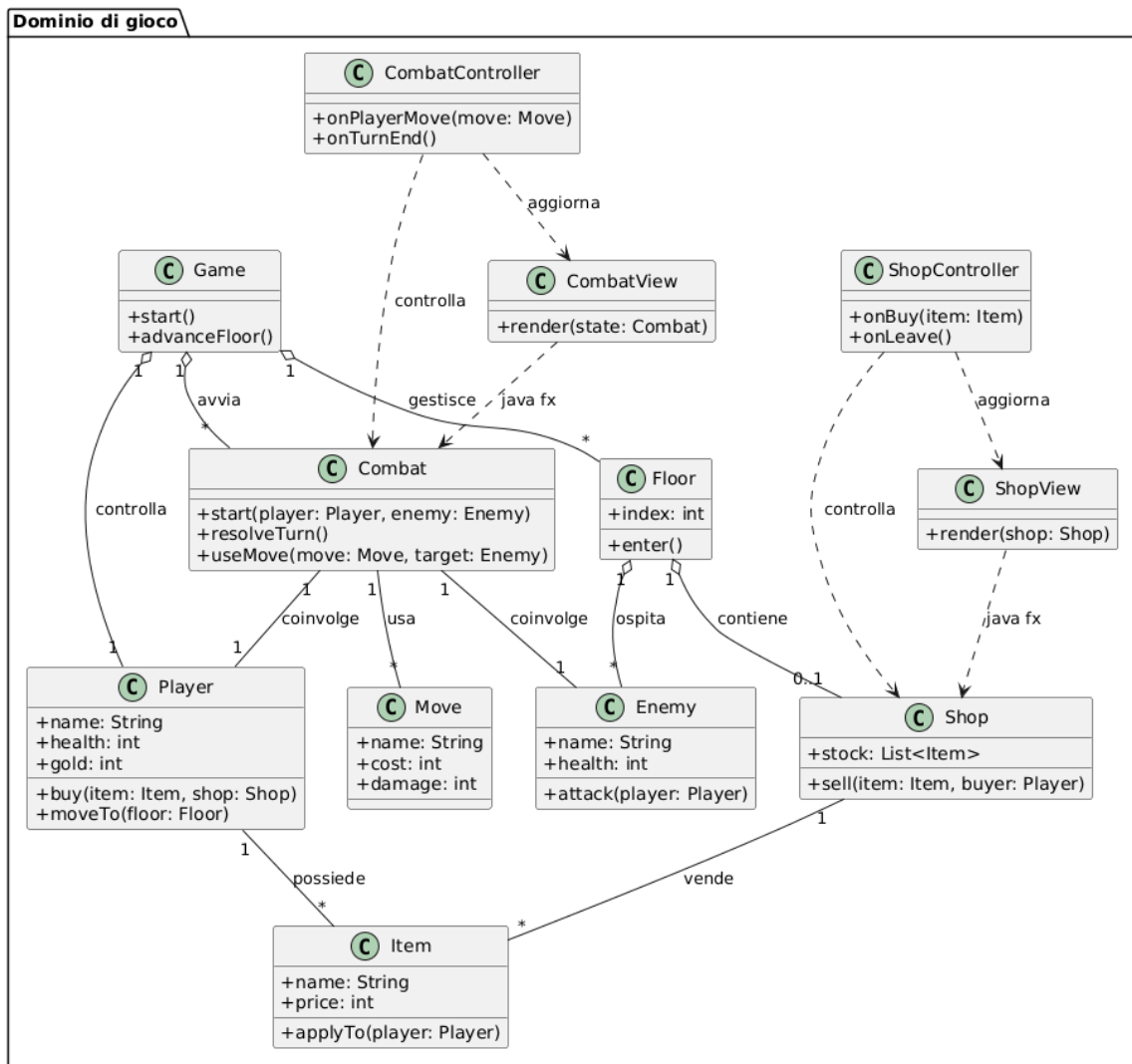


Figura 1: Modello del dominio di gioco semplificato: entità principali (Player, Enemy, Item, Move) e interazioni (Combat e Shop) con separazione tra logica e interfaccia.

2 Design

2.1 Architettura

Il design architeturale di *Abyss Climber* è stato impostato con l'obiettivo di garantire modularità, chiarezza delle responsabilità e possibilità di estensione futura. Poiché il progetto è un videogioco con interfaccia grafica interattiva, è stato fondamentale separare in modo netto la logica di gioco dalla gestione

dell'interfaccia e dalle strutture dati. Questa separazione riduce la complessità complessiva e permette di intervenire in modo localizzato su specifiche funzionalità senza propagare modifiche all'intero sistema.

Il sistema è organizzato secondo una struttura a strati, che può essere descritta come segue:

- **Strato di presentazione (UI)**: costituito dalle schermate grafiche e dai relativi controller. Ha il compito di interagire con l'utente, raccogliere input e visualizzare lo stato corrente del gioco.
- **Strato di logica applicativa (core)**: contiene la logica che governa il flusso della partita, la generazione delle stanze, la gestione del combattimento e il routing tra schermate.
- **Strato del modello (model)**: rappresenta le entità di dominio fondamentali come giocatore, creature, oggetti, classi, elementi e progressione.

Questa scelta architetturale è coerente con un approccio di tipo MVC semplificato (Model–View–Controller), in cui la vista è rappresentata dalle schermate FXML, il controller gestisce l'interazione e il modello contiene lo stato e le regole di dominio. La separazione è sufficiente a garantire una progettazione pulita e manutenibile.

Un elemento centrale è la gestione della navigazione tra schermate. In giochi con molte fasi è cruciale evitare transizioni incoerenti o dipendenze dirette tra controller. Per questo motivo è stato introdotto un componente di routing centralizzato, che si occupa di caricare le schermate e applicare gli stili grafici corretti. Questo approccio ha diversi vantaggi:

- le schermate non si conoscono direttamente tra loro;
- la logica di navigazione è unificata in un solo punto;

- la UI può essere modificata senza alterare il core.

La gestione delle risorse grafiche (immagini delle creature, sfondi, icone) è anch'essa centralizzata. L'adozione di un sistema di caching per le immagini evita il caricamento ripetuto di asset identici, migliorando le prestazioni e la reattività dell'interfaccia. Inoltre, la mappatura tra ID delle creature e asset grafici assicura coerenza tra dati di gioco e rappresentazione visiva.

Nel complesso, l'architettura è stata progettata per bilanciare semplicità di implementazione e robustezza del sistema. Non si tratta di una struttura eccessivamente complessa, ma di un insieme di componenti ben separati che consente di mantenere il controllo sulla complessità crescente tipica di un videogioco con molte funzionalità.

2.2 Design dettagliato

Il design dettagliato esplicita come le scelte architettoniche si concretizzano nei principali sottosistemi: gestione dello stato globale, progressione tra piani e stanze, combattimento, gestione degli oggetti e caricamento dati. In questa sezione vengono analizzati i principali flussi logici e le componenti chiave, evidenziando come siano state progettate per rispondere ai requisiti individuati in fase di analisi.

2.3 Design di Andrea Ciani

Il contributo progettuale di Andrea Ciani si concentra sulla gestione degli oggetti, sulla gestione dei mostri (e caricamento delle loro immagini), sul giocatore e le sue statistiche e sul negozio per acquistare oggetti.

Struttura dati del gioco

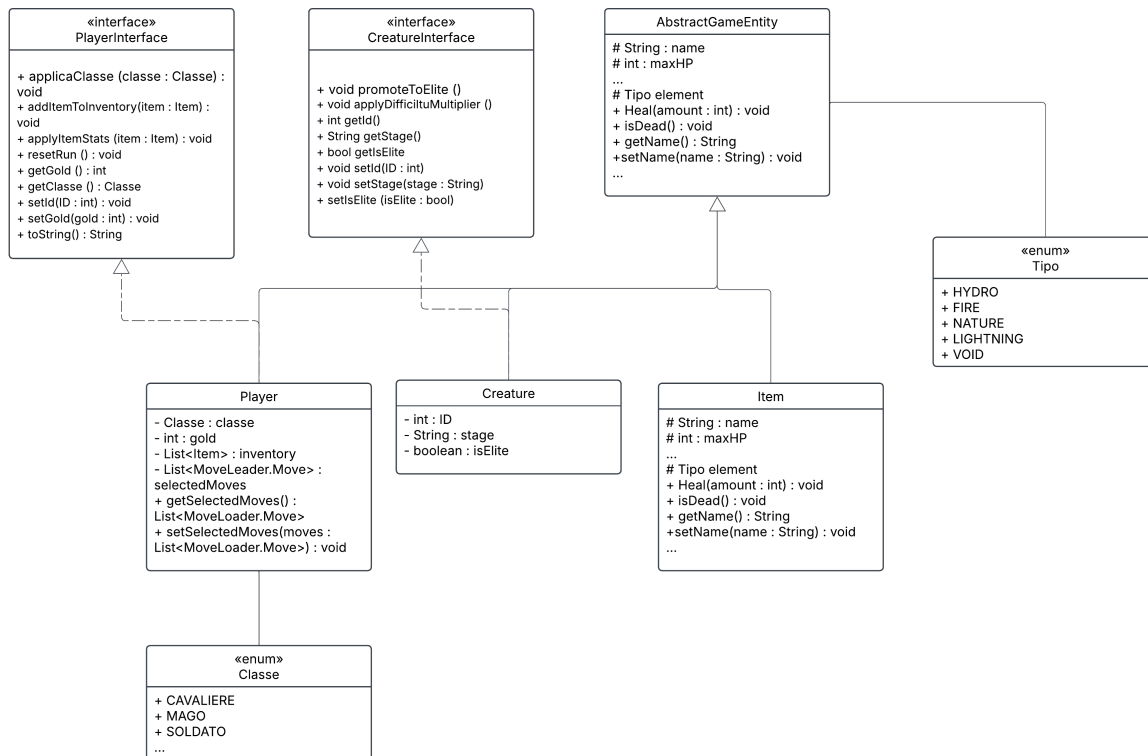


Figura 2: Diagramma della struttura dati del gioco

Problema Necessità di definire un modello dati scalabile che unifichi entità diverse (Giocatore, Mostri, Oggetti) condividendo logiche comuni, ma differenziando i comportamenti (inventario per il player, scaling per i mostri).

Soluzione Si è adottato un design basato su **template method**. L'introduzione della classe astratta **GameEntity** ha permesso di fattorizzare gli attributi di combattimento, riducendo la duplicazione del codice. Per la caratterizzazione del **Player**, invece dell'ereditarietà multipla (es. classe *Mago*), si è preferito un approccio composito tramite l'Enum **Classe**, che agisce come oggetto di configurazione iniettando le statistiche base. Analogamente, la differenziazione dei mostri e la loro evoluzione in varianti *Elite* sono gestite tramite flag interni e metodi di mutazione dello stato, evitando la proliferazione di sottoclassi. Inoltre sia la classe **Player** che

Monster fanno uso di interfacce.

Utilizzo dei dati di gioco

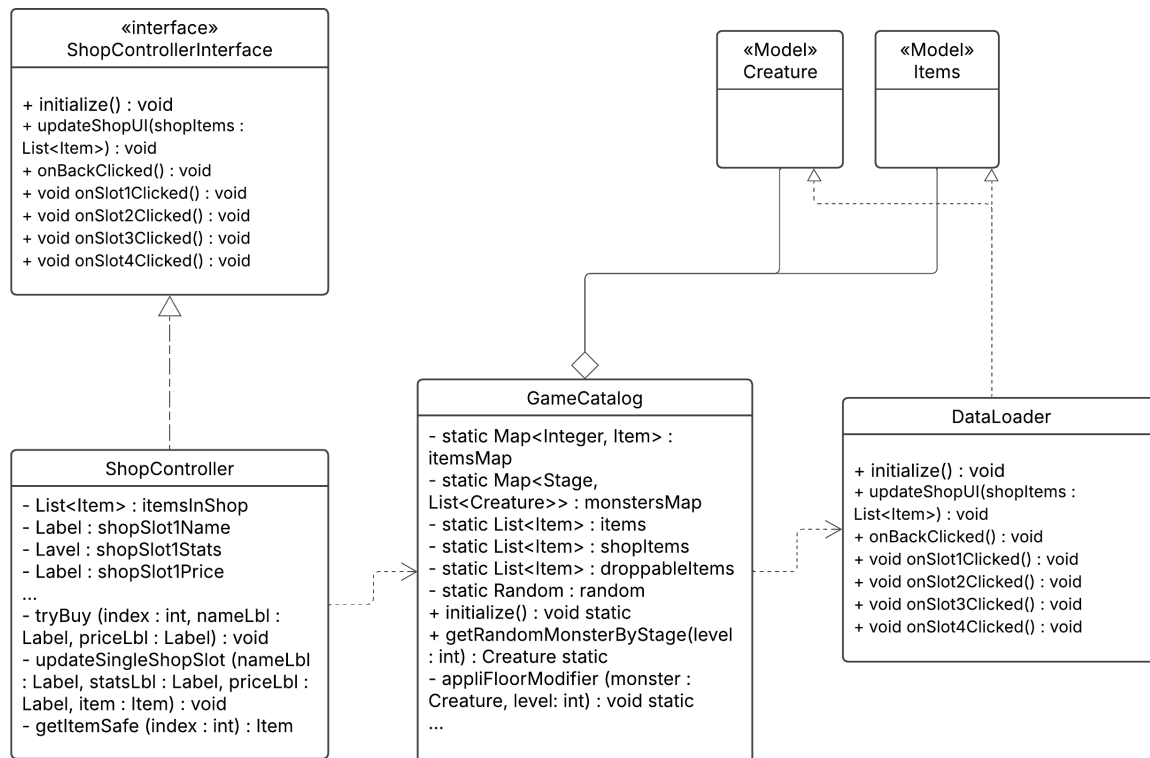


Figura 3: Diagramma della gestione dei dati del gioco

Problema Il sistema richiede un registro centralizzato per gestire l'accesso efficiente agli oggetti Item e Creature, creati a partire da sorgenti esterne, supportare la generazione procedurale in base alla progressione di gioco e fornire i dati necessari ai componenti dell'interfaccia utente, come il sistema di acquisto nel negozio.

Soluzione L'architettura segue il pattern **Data-Driven**. La persistenza è delegata a file JSON esterni, che vengono mappati su oggetti Java tramite la classe `DataLoader`. Questa utilizza la libreria *Jackson*. Per l'accesso a runtime, è stato implementato il pattern **Registry** tramite la classe statica `GameCatalog`, che agisce come *Single Source of Truth*. All'inizializzazione, il catalogo non si limita a caricare i dati, ma applica

una logica procedurale di distribuzione: la lista globale degli oggetti viene mescolata casualmente (`Collections.shuffle`) e suddivisa in due sottoinsiemi disgiunti (inventario del negozio e oggetti droppabili), garantendo che ogni partita offra opportunità economiche diverse. Per la gestione dei nemici, si è utilizzata una `EnumMap` ottimizzata per indicizzare le creature in base allo **Stage**, migliorando le performance ed evitando che vengano letti ogni volta tutti i mostri. Il sistema include inoltre un sistema di *Scaling* (`applyFloorModifier`): invece di definire varianti dello stesso mostro, il catalogo istanzia copie dei mostri base ed in base al piano corrente del giocatore, ne calcola le statistiche applicando moltiplicatori matematici, permettendo una progressione della difficoltà. Infine, il catalogo espone i dati processati ai controller della UI, come il `ShopController`, separando la logica di business dalla visualizzazione.

2.4 Design di Artur Turchyn

Il design di Artur Turchyn riguarda il sistema di **combattimento 1v1**, **l'uso delle abilità** e **l'intelligenza artificiale** dei nemici. Il gioco privilegia meccaniche intuitive che, pur nella loro semplicità, permettono al giocatore di sviluppare strategie efficaci.

Gli elementi progettuali principali includono:

- un sistema di turni con risorse limitate;
- una selezione randomizzata vincolata delle mosse nemiche;
- un calcolo del danno semplice ma arricchito da critici e debolezze elementali;
- un sistema di drop differenziato in base alla tipologia di nemico.

Questo design permette di mantenere equilibrio tra semplicità e profondità, rendendo il combattimento il vero fulcro dell'esperienza.

2.4.1 Gestione del turno di combattimento (Pattern ECB)

Problema:

Il sistema di combattimento deve:

- gestire una sequenza di turni alternati (player -> monster -> player -> ...) e tutti i calcoli necessari;
- aggiornare la UI a ogni passo;
- evitare input multipli o fuori turno;
- mantenere separata la logica di gioco dalla presentazione.

Serve quindi una struttura che separi:

- Entity -> Player, Creature, CombatMove;
- Control -> Combat;
- Boundary -> CombatPresenter.

Soluzione:

È stato adottato un approccio **ECB (Entity-Control-Boundary)**, una variante architetturale di MVC.

- Combat assume ruolo di Control: contiene tutta la logica del turno, del danno, morte e dei drop.
- CombatPresenter assume il ruolo di Boundary: aggiorna la UI, stampa i log, mostra HP del player ecc.
- Player, Creature, CombatMove sono Entity: contengono i valori dagli attributi da utilizzare per il combattimento.

Questa separazione impedisce alla UI di contenere logica di gioco e permette di sostituire o modificare la view senza dover intervenire sul codice.

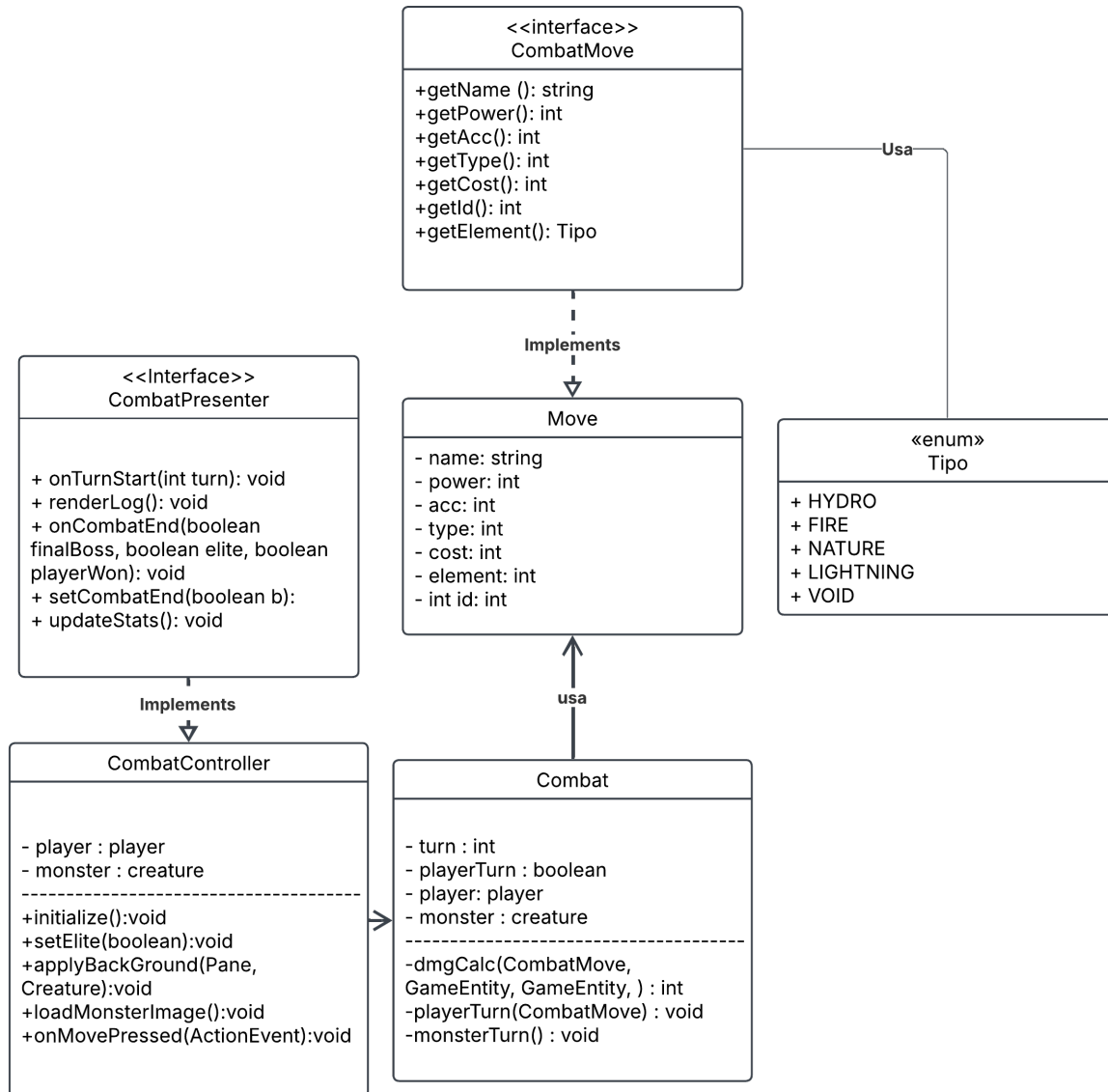


Figura 4: Diagramma UML del loop principale di combattimento

2.4.2 Sistema di caricamento delle mosse nemiche

Problema:

Il sistema di combattimento richiede che ogni nemico disponga di un set di mosse:

- casuale a ogni incontro, ma limitato all'interno di un singolo combattimento;

- almeno una mossa economica;
- mossa speciale disponibile solo al boss finale.

È inoltre necessario un meccanismo che permette di:

- aggiungere nuove mosse senza modificare codice;
- caricare le mosse da un file JSON in modo sicuro ed estendibile;
- selezionare le mosse in base alle caratteristiche del nemico.

Soluzione:

Il sistema di gestione delle mosse nemiche è composto da due componenti principali: `MoveLoader` e `LoadEnemyMoves`. Le mosse sono definite secondo un approccio **data-driven** e caricate da file JSON tramite un componente di loading e istanziamento (`MoveLoader`), che viene eseguito una sola volta all'inizio dell'applicazione. Questo permette l'aggiunta o modifica di mosse senza necessita di interventi su codice e permette una futura estensione del file JSON.

La classe `LoadEnemyMoves` si occupa della selezione delle mosse per ogni nemico ad ogni incontro. La selezione è prevalentemente casuale ma vincolata ad alcune regole:

- viene sempre scelta una mossa a basso costo;
- una preferenza iniziale in base alle statistiche;
- una mossa riservata al boss finale.

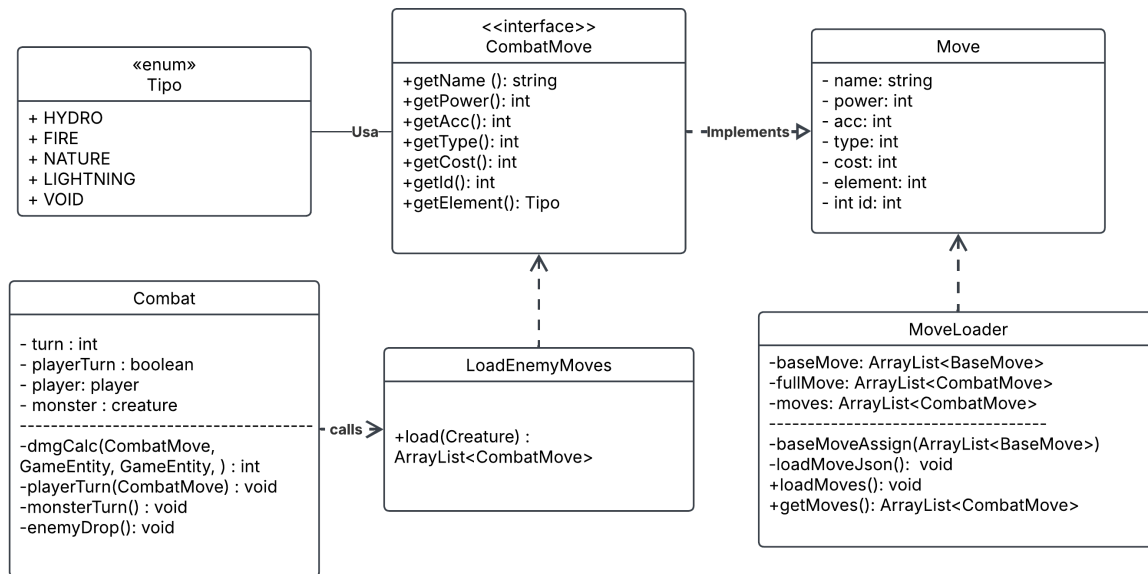


Figura 5: Diagramma UML della gestione mosse nemiche

2.5 Design di Matteo Morellini

Il contributo di design di Matteo Morellini include sia l'aspetto grafico sia componenti chiave della **gestione dello stato di gioco** e della **navigazione tra schermate**. L'obiettivo è stato quello di garantire un flusso di run naturale e comprensibile: schermate coerenti, aggiornamenti dei dati sempre corretti (HP, oro, piano), transizioni robuste e nessuna eredità di stato non più valido tra una fase e l'altra (es. dopo morte o vittoria).

Le scelte progettuali principali includono:

- **routing centralizzato** delle schermate, per evitare dipendenze dirette tra controller e mantenere la navigazione in un solo punto;
- **gestione dello stato globale** della run (player, floor, reset) per supportare restart e ritorno al menu senza residui;
- un sistema di **caricamento e caching degli asset** per rendere la UI stabile, efficiente e manutenibile;

- schermate di **selezione guidata** (personaggio, mosse, porte) con feedback immediato e vincoli di validazione;
- schermate di **fine partita** con animazioni leggere (game over / win) per rafforzare la percezione dell'esito.

2.5.1 Separazione MVC e ruolo dei servizi applicativi

Problema In JavaFX è facile concentrare logica di gioco dentro i controller della UI, rendendo difficile mantenere il codice manutenibile e aumentando il rischio di incoerenze (stato duplicato tra schermate, controlli ripetuti, transizioni non uniformi).

Soluzione È stata adottata una separazione netta tra:

- **View:** schermate FXML (struttura grafica).
- **Controller:** classi `ui.*`, responsabili di leggere input, aggiornare label e bottoni e invocare operazioni di alto livello.
- **Core/Services:** logica di orchestrazione e validazione concentrata in servizi (`core.services.*`), richiamati dai controller per avviare la run, confermare selezioni e gestire le transizioni.

In questo modo i controller rimangono leggeri e la logica non è duplicata tra schermate.

2.5.2 Navigazione centralizzata: SceneRouter e SceneId

Problema La presenza di molte schermate (menu, creazione personaggio, selezione mosse, selezione stanze, combat, shop, game over, win) richiede transizioni coerenti, caricamento uniforme di FXML e CSS e gestione corretta dei parametri (es. differenza tra fight e boss).

Soluzione È stato introdotto un router centralizzato (**SceneRouter**) guidato da un enum (**SceneId**). Il router:

- mappa ogni **SceneId** al relativo FXML;
- carica automaticamente lo stylesheet corretto (stile generale o combattimento);
- gestisce casi speciali, come la creazione del controller di combattimento con flag per le boss fight;
- notifica i controller che implementano **Refreshable** tramite **onShow()**, così che la schermata aggiorni sempre HUD e contenuti dinamici quando viene visualizzata.

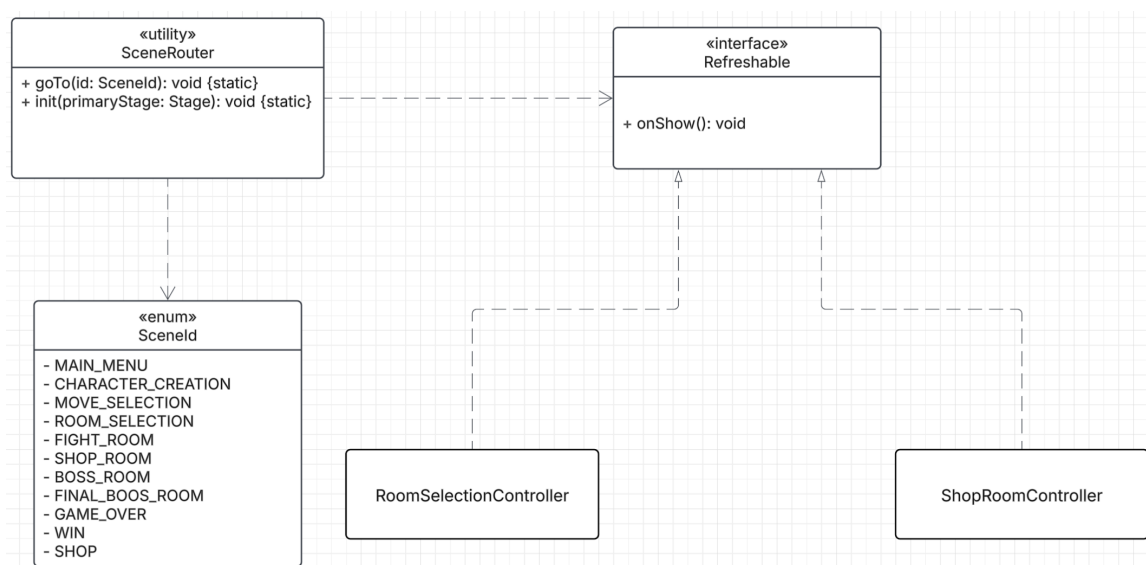


Figura 6: Diagramma UML del routing centralizzato e del meccanismo **Refreshable** per l'aggiornamento delle schermate.

2.5.3 Gestione dello stato della run: GameState e reset senza residui

Problema In un roguelike, morte o vittoria devono riportare il gioco a uno stato valido. Senza una gestione centralizzata, le schermate rischiano di

riutilizzare dati obsoleti (porte disabilitate dal piano precedente, opzioni cached non valide, player non inizializzato).

Soluzione È stato centralizzato lo stato in **GameState** (singleton), che mantiene **Player** e **floor**. La logica include:

- inizializzazione del player a creazione personaggio;
- avanzamento di piano con **nextFloor()**;
- reset completo della run con **resetRun()**, che azzerava player e floor e ripulisce lo stato legato alle stanze (**RoomContext**), evitando qualunque incoerenza dopo un restart o un ritorno al menu.

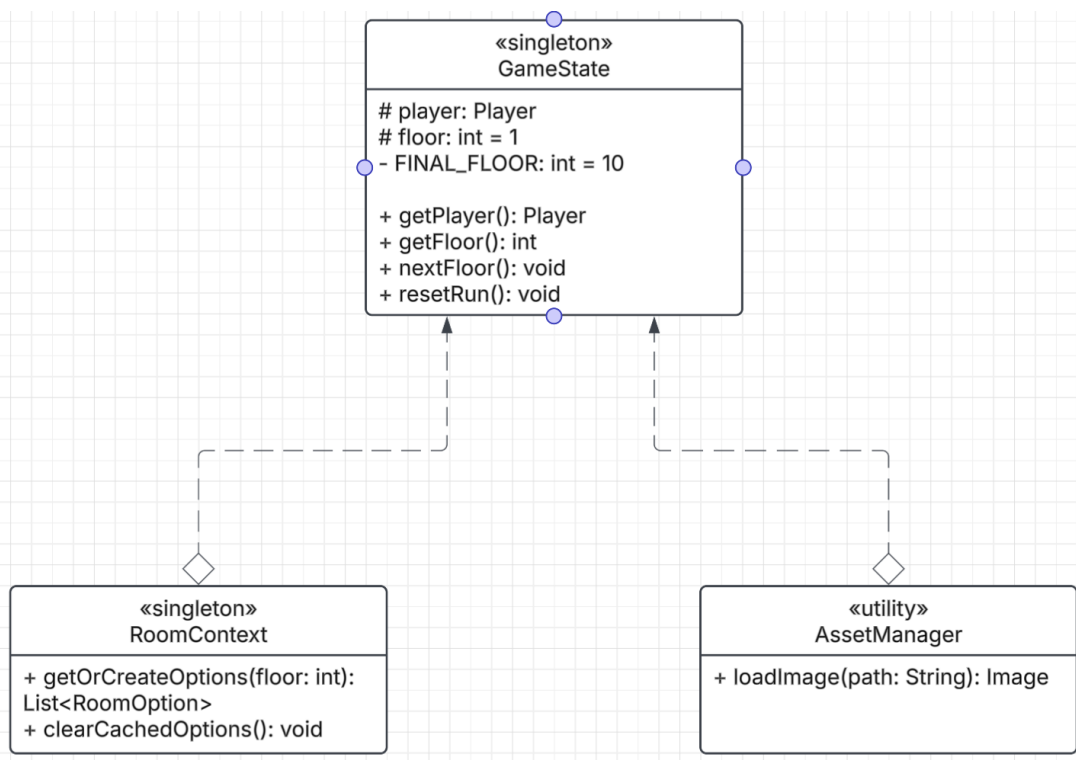


Figura 7: Diagramma UML della gestione dello stato di run (**GameState**) e del contesto di selezione delle stanze (**RoomContext**), con utility di supporto.

2.5.4 Gestione asset: `AssetManager` con caching e fallback

Problema Il caricamento ripetuto delle immagini in JavaFX è costoso e rende fragile la UI in caso di asset mancanti (crash o schermate vuote non gestite).

Soluzione È stato implementato `AssetManager` come utility centralizzata con:

- cache concorrente delle immagini per riuso immediato;
- `loadImage` per caricare in modo rigoroso e segnalare errori;
- `tryLoadImage` per gestire fallback senza interrompere l'esecuzione (warning e ritorno `null`).

Questa scelta migliora prestazioni e robustezza, evitando duplicazioni della logica di caricamento nei controller.

2.5.5 Scelte guidate: creazione personaggio, mosse e selezione porte

Problema Le schermate iniziali devono guidare l'utente e impedire configurazioni incoerenti (avvio run senza scelte, selezione mosse non valida, rigenerazione casuale delle porte tornando indietro).

Soluzione La UI è stata progettata con vincoli e feedback immediati:

- **Creazione personaggio:** selezione di tipo, classe e difficoltà tramite `ToggleGroup` con label riassuntiva aggiornata live e conferma vincolata a scelte complete.

- **Selezione mosse:** popolamento dinamico per elemento e vincoli di validazione (numero massimo e presenza di almeno una mossa a costo minimo), con pulsante di avvio abilitato solo se la selezione è valida.
- **Selezione stanze:** aggiornamento HUD (piano/HP/oro) e opzioni porta recuperate in modo consistente per piano.

2.5.6 Contesto stanze: RoomContext e caching per piano

Problema La schermata di scelta porte deve essere prevedibile: se si torna alla mappa nello stesso piano, le opzioni non devono cambiare casualmente, e alcune porte devono poter risultare non selezionabili (ad esempio dopo una scelta già effettuata).

Soluzione È stato introdotto `RoomContext` per conservare:

- l'ultima stanza selezionata;
- una cache delle opzioni generate per il piano corrente;
- un insieme di indici delle porte disabilitate, usato dal controller per disattivare e rendere visivamente bloccate le porte già consumate.

La generazione delle opzioni segue una regola semplice e leggibile: boss al centro (elite o finale) e due porte laterali fight/shop. L'aggiornamento della schermata avviene in `onShow()` tramite `Refreshable`.

2.5.7 Fine partita: schermate *Game Over* e *Win* con animazioni

Problema L'esito di una run dovrebbe essere immediato e comunicativo anche dal punto di vista visivo, senza introdurre complessità nella logica.

Soluzione Sono state implementate animazioni leggere e non invasive:

- **Game Over:** animazione “caduta + oscillazione” del titolo (effetto cartello appeso) per enfatizzare la sconfitta;
- **Win:** animazione di “bounce” verticale del titolo per comunicare la vittoria.

In entrambi i casi la schermata gestisce solo il feedback grafico e il ritorno al menu tramite **SceneRouter**, mantenendo il flusso pulito e uniforme.

3 Sviluppo

3.1 Testing automatizzato (JUnit + Mockito)

Per verificare la correttezza delle componenti principali del progetto, abbiamo implementato una suite di test automatici con **JUnit 5**. I test coprono sia aspetti di logica pura (stato globale, caching, generazione stanze) sia parti del combattimento; per queste ultime, dove necessario, è stato utilizzato **Mockito** per isolare le dipendenze (ad esempio generatori random e metodi statici).

Di seguito si riportano i file di test principali e l’obiettivo di ciascuno:

- **GameStateFloorTest.java:** verifica che `nextFloor()` incrementi correttamente il piano e che `resetRun()` ripristini il `floor` a 1.
- **GameStateTest.java:** verifica che `resetRun()` azzeri lo stato della run, rimuovendo il player e ripulendo il `RoomContext` (`lastChosen` e porte disabilitate).
- **RoomContextClearCacheTest.java:** verifica che `clearCachedOptions()` elimini cache e porte disabilitate e permetta la rigenerazione corretta delle opzioni.

- `RoomContextTest.java`: verifica il caching per piano (stesso floor → stessa lista) e il reset automatico delle porte disabilitate al cambio di piano.
- `RoomGeneratorTest.java`: verifica che la generazione produca sempre 3 opzioni, con boss centrale (elite prima del floor finale, final boss al floor finale) e vincoli sulle porte laterali.
- `RoomGeneratorExtraTest.java`: verifica che i boss compaiano esclusivamente in posizione centrale e mai nelle porte laterali.
- `CombatTest.java`: verifica il calcolo del danno (hit, miss, critico) e l'applicazione dei moltiplicatori elementali, isolando dipendenze tramite Mockito (Random, metodi statici e selezione mosse).
- `GameCatalogTest.java`: verifica l'inizializzazione degli oggetti shop, creazione di una copia del mostro, spawn corretto del mostro, unicità degli oggetti droppabili (finché la lista non termina) e ordinamento degli item nello shop.
- `CreatureTest.java`: verifica il copy constructor (deep copy), la promozione a elite (incremento statistiche e idempotenza) e la corretta gestione dello stato `isDead()`.
- `PlayerTest.java`: verifica creazione player con statistiche base, applicazione corretta della classe, gestione oro e applicazione effetti degli item (bonus stats, cure, cap a MaxHP e incremento MaxHP che cura).

3.2 Note di sviluppo

3.2.1 Andrea Ciani

Di seguito sono elencate le funzionalità avanzate utilizzate, con i riferimenti diretti al codice sorgente:

- **Utilizzo di Jackson per lettura dati:** Utilizzo di jackson per la deserializzazione automatica dei file JSON in oggetti Java complessi.

[Codice su GitHub: [DataLoader.java](#)]

- **Utilizzo di Lambda:** Uso di una lambda per ordinare gli oggetti nel negozio in base al loro costo..

[Codice su GitHub: [GameCatalog.java](#)]

- **Uso dei Generici:** Utilizzo della classe `TypeReference<T>` per implementare il pattern "Super Type Token", permettendo di preservare e recuperare le informazioni sui tipi generici a runtime (aggirando la Type Erasure).

[Codice su GitHub: [DataLoader.java](#)]

- **Utilizzo di JavaFX e Dependency Injection:** Sfruttamento del meccanismo di Dependency Injection del framework tramite l'annotazione `@FXML`, che collega automaticamente i componenti definiti nel layout XML alle variabili d'istanza del controller.

[Codice su GitHub: [ShopController.java](#)]

3.2.2 Artur Turchyn

Di seguito sono riportate alcune funzionalità avanzate utilizzate nell'implementazione del sistema di combattimento, con riferimento diretto al codice

sorgente.

- **Utilizzo di Lambda e Stream:**

[Codice su GitHub: `LoadEnemyMoves.java`]

- **Utilizzo di Jackson per il parsing di file JSON:**

[Codice su GitHub: `MoveLoader.java`]

- **Utilizzo di JavaFX:**

[Codice su GitHub: `CombatController.java`]

3.2.3 Matteo Morellini

La parte grafica e la navigazione tra schermate hanno richiesto un coordinamento continuo tra asset visivi e logica di gioco. È stata progettata una UI coerente e dinamica, supportata da un sistema di caricamento centralizzato delle immagini, con meccanismi di caching e fallback per garantire stabilità e buone prestazioni anche in presenza di asset mancanti.

[Codice su GitHub: `AssetManager.java`]

Accanto alla parte visiva, è stata gestita la logica relativa allo stato globale della partita. In particolare, è stato implementato un gestore centralizzato dello stato della run che mantiene informazioni sul player corrente e sul piano attivo, consentendo un reset completo e consistente in caso di morte, vittoria o ritorno al menu principale.

[Codice su GitHub: `GameState.java`]

Per garantire transizioni coerenti tra le numerose schermate del gioco, è stato inoltre introdotto un sistema di navigazione centralizzato. Il router si occupa del caricamento delle schermate JavaFX, dell'applicazione degli stili grafici corretti e della notifica dei controller quando una schermata viene

visualizzata, evitando dipendenze dirette tra i vari controller della UI.

[Codice su GitHub: `SceneRouter.java`]

In questo modo, la gestione dello stato del player (morte, reset, ritorno al menu) e la navigazione tra schermate sono state integrate in un flusso complessivo uniforme e robusto, garantendo un'esperienza utente fluida ma anche tecnicamente corretta.

3.3 Problemi affrontati durante lo sviluppo

Durante lo sviluppo sono emerse difficoltà tipiche dei progetti di videogiochi:

- bilanciamento delle statistiche e delle mosse;
- gestione di stati globali senza errori di sincronizzazione;
- coordinamento tra UI e logica di gioco;
- integrazione di asset grafici con dati numerici.

Questi problemi sono stati affrontati mediante iterazioni progressive, test manuali e semplificazioni mirate, mantenendo la coerenza del prototipo.

3.4 Valutazione complessiva dello sviluppo

In conclusione, lo sviluppo di *Abyss Climber* ha permesso di realizzare un prototipo completo e giocabile, in linea con i requisiti iniziali. Il codice è stato organizzato in modo modulare, le principali meccaniche di gioco sono state implementate con successo e l'interfaccia utente offre un'esperienza coerente.

Il lavoro svolto costituisce una solida base per eventuali estensioni future, che potrebbero includere:

- ampliamento del catalogo di abilità e nemici;

- introduzione di status effect e meccaniche avanzate;
- miglioramento delle animazioni e del comparto audio;
- implementazione completa di un sistema di testing automatico.

4 Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Andrea Ciani

In generale sono contento del lavoro svolto, anche se con un po di tempo un piú si sarebbe potuto sviluppare una applicazione generale migliore, magari con aggiunta di suoni ed imprevisti durante la run. Il gruppo é sempre stato disponibile e nessuno si é mai lamentato del proprio lavoro. Mi sono però spesso trovato ad aggiungere in corso d'opera alcune funzioni che in origine non erano state pensate, come ad esempio il moltiplicatore delle statistiche per ogni piano. Questo implica che si sarebbe dovuta fare una analisi piú approfondita in tempi precedenti. Sempre per motivi di tempo non sono riuscito ad implementare delle cose che mi sarebbe piaciuto inserire :

- oggetti con effetti attivi per rendere il combattimento piú dinamico; da qui l'esistenza del paramentro effect ridotto purtroppo ad una mera stringa;
- combinazioni tra oggetti per generare effetti unici;
- tracciamento degli oggetti scoperti tra una run e l'altra; esiste infatti la variabile `discovered` appartenente agli oggetti ridotta anche lei ad un semplice booleano.

4.1.2 Artur Turchyn

Dal punto di vista del combattimento, il lavoro svolto ha permesso di implementare un sistema stabile e funzionante, ma ancora suscettibile di ampliamenti. Il combattimento a turni è coerente con il genere, e il sistema di stamina introduce una forma di gestione delle risorse che impedisce l'abuso di mosse potenti. L'elemento strategico è rafforzato dalle debolezze elementali e dalla possibilità di selezionare le abilità iniziali, aspetti che rendono ogni partita diversa dalle altre.

Punti di forza

- la chiarezza del ciclo di turni;
- la logica di danno basata su statistiche, critici e precisione;
- la presenza di un meccanismo elementale coerente.

Sviluppi futuri

- introduzione di status alterati (veleni, stun, buff, debuff, aumento temporaneo di statistiche);
- ampliamento dell'AI con comportamenti diversificati tra nemici;
- maggior bilanciamento tra mosse e classi per garantire una difficoltà più controllata;
- nuove classi per il giocatore.

Punti deboli

- il combattimento è funzionale ma semplice;
- mancanza di animazioni durante il combattimento;

- mancanza di effetti sonori.

È stata riscontrata difficoltà nella creazione del .jar eseguibile in combinazione di Gradle + JavaFx.

4.1.3 Matteo Morellini

Mi ritengo soddisfatto del lavoro svolto. Nella fase iniziale del progetto è stato talvolta complesso procedere in modo perfettamente sincronizzato, poiché alcune parti del mio contributo dipendevano dalle scelte implementative dei miei compagni, e viceversa. Nonostante queste difficoltà iniziali, il gruppo è riuscito a trovare rapidamente un buon ritmo di lavoro, portando a termine il progetto in modo efficace.

Personalmente ho apprezzato molto lavorare alla mia parte, in quanto riguarda componenti centrali del sistema: la gestione della navigazione tra le schermate e il coordinamento dello stato globale del gioco. Si tratta di aspetti fondamentali per garantire un flusso di gioco coerente e privo di incongruenze. Il lavoro è stato impegnativo, ma allo stesso tempo stimolante e formativo.

Punti di forza

- flusso di navigazione chiaro e lineare;
- buona organizzazione delle schermate;
- gestione centralizzata degli asset;
- gestione coerente dello stato di partita in caso di morte o vittoria;
- reset corretto della run al restart o al ritorno al menu.

Sviluppi futuri In prospettiva futura, è possibile immaginare diversi miglioramenti:

- introduzione di un sistema audio completo, con effetti sonori in combattimento e colonna sonora;
- transizioni più dinamiche tra le schermate;
- ottimizzazione grafica per supportare risoluzioni diverse;
- salvataggio dello stato della run per permettere di riprendere una partita in un secondo momento.

Questi interventi contribuirebbero a migliorare l'impatto visivo e la qualità percepita del progetto, avvicinando il prototipo a un prodotto più rifinito.

4.2 Difficoltà incontrate e commenti per i docenti

Lo sviluppo di un videogioco, anche in forma di prototipo, comporta sfide multidimensionali. Nel caso di *Abyss Climber*, le principali difficoltà possono essere riassunte in tre macro-aree: bilanciamento, integrazione e gestione dell'interfaccia.

Bilanciamento e complessità del combattimento Il bilanciamento del combattimento è risultato uno degli aspetti più complessi. Garantire che le mosse siano bilanciate, che l'AI sia credibile ma non frustrante, e che le ricompense siano proporzionate al rischio ha richiesto diverse iterazioni. Inoltre, il sistema elementale aggiunge un ulteriore livello di complessità: bisogna evitare che un solo elemento o una sola combinazione di mosse risulti dominante.

Integrazione tra moduli Un secondo punto critico riguarda l'integrazione. Il progetto coinvolge moduli diversi (UI, core, model, dati esterni), ciascuno con responsabilità specifiche. Coordinare queste parti richiede attenzione

per prevenire incoerenze tra stato interno e interfaccia visiva. Ad esempio, la progressione del giocatore deve riflettersi immediatamente sull'HUD, e la transizione tra stanze deve conservare lo stato corretto della run.

Gestione della UI L'interfaccia grafica, pur essendo un elemento spesso considerato secondario nello sviluppo di prototipi, è invece fondamentale per la fruibilità. La difficoltà principale è stata garantire un equilibrio tra semplicità di utilizzo e completezza informativa, evitando di sovraccaricare lo schermo con troppe informazioni o controlli.

Considerazioni sui vincoli di tempo Come accade in molti progetti accademici, il tempo limitato ha imposto compromessi. Alcune funzionalità opzionali (audio, animazioni avanzate, modalità di input alternative) non sono state implementate, pur essendo previste in fase di analisi. Questo non pregiudica la completezza del prototipo, ma evidenzia la necessità di una pianificazione realistica e di una chiara definizione delle priorità.

4.3 Riflessioni generali sul progetto

Nel complesso, il progetto ha raggiunto l'obiettivo di creare un prototipo di videogioco giocabile e coerente. La struttura modulare, la separazione tra logica e interfaccia e l'uso di dati esterni dimostrano un approccio maturo allo sviluppo software.

Tra i principali risultati ottenuti si possono citare:

- un ciclo di gioco completo;
- un sistema di combattimento funzionante e strategico;
- un'interfaccia intuitiva;

- una base solida per future estensioni.

Dal punto di vista didattico, il progetto ha permesso di affrontare temi fondamentali dell'ingegneria del software: progettazione modulare, gestione dello stato, interfacce utente e integrazione di componenti eterogenei. L'esperienza evidenzia come lo sviluppo di videogiochi, anche in scala ridotta, richieda competenze trasversali e la capacità di integrare design e implementazione.

4.4 Proposte di estensione future

Per concludere, si riportano alcune proposte di estensione che potrebbero essere sviluppate in una fase successiva:

- **Ampliamento del contenuto:** nuovi nemici, nuove mosse, nuove classi e nuovi oggetti con effetti speciali.
- **Meccaniche avanzate:** introduzione di status effect, abilità passive e combo di abilità.
- **Sistemi di progressione a lungo termine:** tracciamento degli oggetti scoperti, achievement, modalità di sblocco.
- **Miglioramenti audiovisivi:** musica dinamica, effetti sonori e animazioni più ricche.
- **Accessibilità e input alternativi:** supporto per controller o opzioni di personalizzazione dei comandi.

Questi miglioramenti dimostrano che il prototipo sviluppato costituisce una base solida e significativa, ma anche aperta a possibili evoluzioni, come tipico dei progetti di software interattivo.

4.5 Conclusione

In conclusione, *Abyss Climber* rappresenta un progetto completo e coerente, in grado di soddisfare i requisiti principali definiti in fase di analisi. La modularità del codice, la chiarezza del design e la presenza di un ciclo di gioco completo dimostrano che il prototipo è stato sviluppato con un approccio rigoroso. Pur essendo possibile introdurre numerose estensioni, il risultato ottenuto è già sufficiente per rappresentare un valido prodotto didattico, capace di mostrare competenze di progettazione, sviluppo e integrazione software.

5 Appendice

5.1 Guida utente

Il gioco ci pare intuitivo tranne per 2 cose :



Figura 8: Creazione del personaggio.

La scelta del tipo indica a quale elemento si sarà più vulnerabili da-

gli attacchi nemici durante tutta la run. La scelta della classe invece aumenterà determinate statistiche base del personaggio; Knight modifica principalmente l'attacco fisico e la difesa fisica, Mage modifica principalmente l'attacco magica e la difesa magica, Soldier é la via di mezzo.

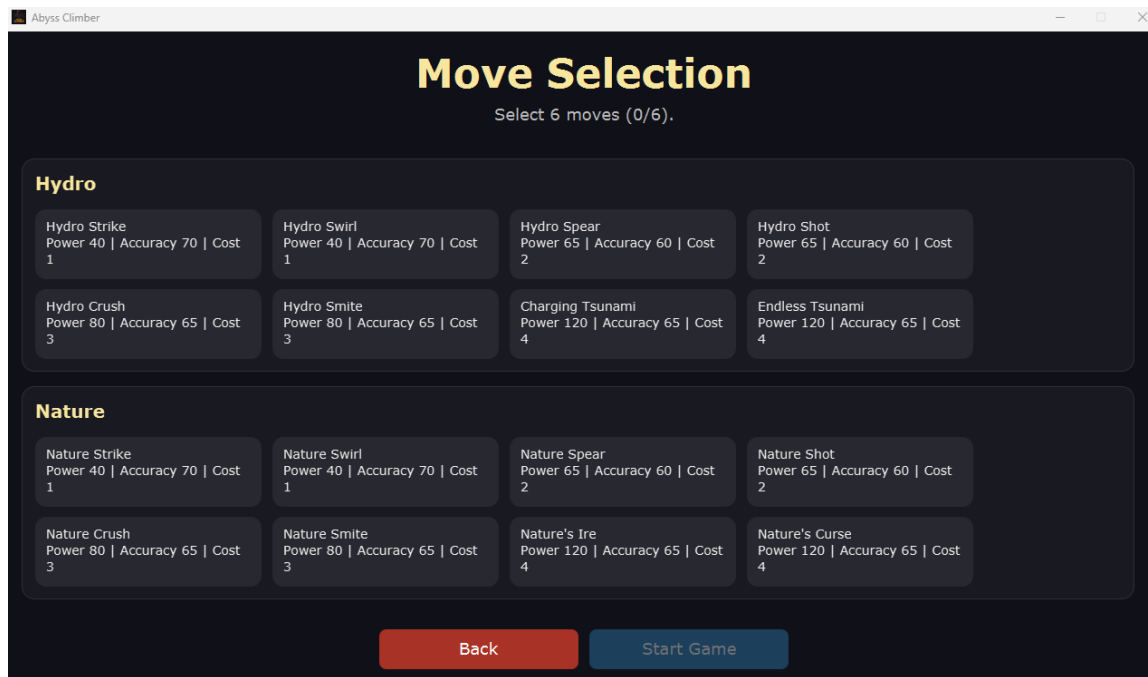


Figura 9: Scelta delle mosse

Anche i mostri sono vulnerabili a certi tipi di mosse; un nemico fatto di fuoco sarà più vulnerabile a mosse di tipo acqua. Conviene quindi scegliere tipi di mosse diverse per rendere la run più facile. Ci siamo ispirati al sistema dei Pokemon.