The Legend of Bald

Relazione di progetto

Team di sviluppo
Università di Bologna

29 novembre 2025

Sommario

Questa relazione documenta obiettivi, requisiti, progettazione e implementazione del videogioco *The Legend of Bald*, sviluppato in Java (Gradle) con interfaccia grafica basata su Swing/AWT. Il documento funge da riferimento tecnico e come base per la discussione d'esame.

Indice

1 Introduzione

1.1 Contesto e obiettivi

- **Genere:** RPG 2D con combattimento in tempo reale.
- **Tecnologie:** Java 21, Gradle, Swing/AWT (rendering 2D), eventuale gestione risorse (sprites, tilemap).
- **Obiettivi didattici:** progettazione orientata agli oggetti, gestione del *game loop*, collisioni, AI semplice, UI di gioco.

1.2 Ambito del progetto

Il progetto "The Legend of Bald" si configura come un videogioco d'azione 2D in stile *Arcade*, con l'obiettivo di superare una serie di livelli sconfiggendo nemici fino al boss finale. La versione consegnata implementa un ciclo di gioco completo, includendo le seguenti funzionalità chiave:

- **Movimento e Mappe:** Il personaggio principale, Bald, può muoversi nelle quattro direzioni all'interno di mappe predefinite caricate da file di testo. Il gioco è strutturato su più piani, ognuno con una propria mappa, nemici e oggetti.
- **Nemici e Combattimento:** Sono stati implementati diversi tipi di nemici con comportamenti di base e un boss finale. Il sistema di combattimento permette al giocatore di attaccare utilizzando diverse armi, come spada e ascia, ciascuna con animazioni dedicate.
- **Oggetti e Interazione:** Il giocatore può interagire con l'ambiente per raccogliere oggetti. Le casse (*chest*) possono essere aperte per ottenere monete o pozioni (salute e forza). È presente anche uno *shop* dove è possibile acquistare nuove armi utilizzando le monete raccolte.
- **Inventario e HUD:** Inventario visualizzabile durante il gioco, il sistema gestisce internamente le monete e le pozioni raccolte. L'interfaccia utente (HUD) mostra informazioni essenziali come la vita del giocatore e un timer per la *run* corrente, visibile a seconda delle impostazioni scelte.
- **Gestione della Partita:** Il gioco implementa un menu principale completo di impostazioni (video, audio, controlli personalizzabili), una classifica locale (*leaderboard*) per le migliori *run* e la gestione del ciclo di vittoria o sconfitta, che riporta al menu principale.

L'ambito del progetto si è concentrato sulla creazione di un'esperienza di gioco completa e rigiocabile, dalla schermata iniziale alla sconfitta del boss finale, enfatizzando la logica di gioco, la gestione delle entità e l'interazione con l'ambiente.

2 Requisiti

2.1 Requisiti funzionali

Esempio (adattare alla vostra build):

- RF1: Il giocatore può muoversi nelle quattro direzioni sulla mappa.
- RF2: Il sistema gestisce collisioni con muri, ostacoli e nemici.
- RF3: Il giocatore può attaccare; i nemici subiscono danno e possono essere eliminati.
- RF4: HUD mostra punti vita (e altre statistiche rilevanti).
- RF5: È presente un menu principale e una schermata di pausa.

2.2 Requisiti non funzionali

- RNF1: Il gioco gira a framerate stabile (target 60 FPS) sul target hardware.
- RNF2: Struttura modulare del codice, separazione logica tra model, view, input.
- RNF3: Build automatizzata con Gradle, eseguibile *jar* generato.

3 Progettazione

3.1 Architettura generale

Il progetto segue il pattern architetturale **Model-View-Controller (MVC)** per separare la logica di gioco dalla sua rappresentazione e dall'interazione con l'utente. I principali sottosistemi sono:

- **Game Core (Model/Controller):** Gestisce il ciclo di gioco principale (*game loop*), che si occupa di aggiornare lo stato del mondo (*update*) e di disegnarlo a schermo (*render*). Questo sottosistema coordina la transizione tra i vari stati dell'applicazione, come il menu principale, la schermata di gioco, l'inventario, le impostazioni e la leaderboard. La logica è implementata nel package `model.system` e la gestione degli stati in `controller.navigation`.

- **World/Map (Model):** Responsabile della gestione delle mappe di gioco. Le mappe sono caricate da file di testo (`.txt`) che definiscono la disposizione delle *tile* e la loro collisione. Anche la posizione degli oggetti interattivi, come casse e shop, viene definita tramite file di testo dedicati, garantendo modularità e facilità nella creazione di nuovi livelli.

- **Entity System (Model):** Modella tutte le entità presenti nel gioco. È presente una gerarchia di classi che parte da un'entità base e si specializza nel personaggio giocante (`Bald`), nei vari tipi di nemici e nel boss finale. Ogni entità gestisce il proprio stato, posizione, animazioni e logica di comportamento (es. IA basilare per i nemici). Il sistema di combattimento, definito nel package `combat`, gestisce le interazioni offensive tra le entità.

- **Collision System (Model):** Implementa la logica per il rilevamento delle collisioni. Viene utilizzato un sistema basato su *Axis-Aligned Bounding Box (AABB)* per gestire le interazioni tra le entità (giocatore, nemici, proiettili) e tra le entità e l'ambiente di gioco (muri, oggetti).

- **Item/Object Management (Model/Controller):** Gestisce il ciclo di vita completo di tutti gli oggetti interattivi presenti nel mondo di gioco (`GameItem`). Questo sottosistema implementa un **Manager** (*ItemManager*) per l'aggiornamento e l'interazione degli oggetti con il giocatore. La gerarchia si specializza in oggetti utilizzabili (*Potion*) e trappole (*Trap*), sfruttando il **Polimorfismo** per delegare l'effetto specifico al momento della collisione con il personaggio giocante.

- **Input (Controller):** Il sottosistema di input, localizzato in `controller.listeners`, cattura gli eventi generati da tastiera e mouse. Permette la gestione dei movimenti del personaggio, delle azioni di gioco (attacco, apertura inventario) e dell'interazione con i menu. Il sistema supporta la personalizzazione dei tasti (*key bindings*), le cui configurazioni vengono salvate e caricate.

- **UI/HUD (View):** Si occupa della costruzione e della visualizzazione di tutti gli elementi dell'interfaccia grafica. Include il menu principale, le schermate delle impostazioni e della leaderboard, l'inventario e l'HUD di gioco, che mostra informazioni vitali come la vita del giocatore e il timer della run. La UI è realizzata con componenti Swing personalizzati, come si vede nei package `view.component` e `view.panel`.

- **Audio System:** Gestisce la riproduzione degli effetti sonori associati alle interazioni dell'utente con l'interfaccia (es. click dei bottoni). Sebbene la struttura sia predisposta, la componente musicale di sottofondo è ancora in fase di sviluppo, come indicato nel `README.md`.

3.2 Game loop e gestione stati

Il cuore del gioco è rappresentato da un *game loop* implementato in una classe dedicata che implementa l'interfaccia `Runnable`, permettendone l'esecuzione in un thread separato. Questo ciclo è responsabile dell'aggiornamento continuo dello stato del gioco e del suo rendering a schermo.

Il loop adotta un approccio a **timestep fisso** per garantire che la logica di gioco (fisica, movimento, IA) sia indipendente dalla velocità di esecuzione della macchina su cui gira. L'obiettivo è fissato a un numero predefinito di aggiornamenti al secondo (UPS). All'interno del ciclo, viene calcolato il tempo trascorso dall'ultimo aggiornamento; se questo intervallo supera la soglia definita per un singolo *tick* (es. 1/120 di secondo), viene invocato il metodo `update()`.

Il metodo `update()` è il punto di ingresso per tutta la logica di gioco, gestita principalmente dal package `model`. Esso si occupa di aggiornare la posizione del giocatore, l'intelligenza artificiale dei nemici, verificare le collisioni e gestire le interazioni.

Il rendering, invece, viene eseguito il più frequentemente possibile, ma limitato a un massimo di FPS (Frames Per Second) per non sprecare risorse. La chiamata al metodo `render()` (o `repaint()` in Swing) ridisegna l'intera scena di gioco, utilizzando le informazioni aggiornate dal modello.

La **gestione degli stati** non è implementata tramite una classica *State Machine* con classi dedicate (es. `GameState`, `MenuState`), ma attraverso un sistema di navigazione centralizzato nel package `controller.navigation`. Questo sistema gestisce quale pannello (`JPanel`) è attualmente visualizzato nella finestra principale (`JFrame`). La transizione da uno stato all'altro avviene sostituendo il pannello correntemente attivo nella finestra principale.

3.3 Architettura del Sistema Entità

La base di tutto il sistema è la classe astratta Entity, che definisce il contratto e lo stato fondamentale di un attore nel mondo di gioco.

3.3.1 La Classe Base Astratta: *Entity*

La classe `Entity.java` costituisce la radice della gerarchia per tutti gli oggetti dinamici del gioco (Player, Nemici, NPC, Proiettili). Essendo una classe concreta (non abstract nel codice, ma concettualmente astratta nel dominio), definisce lo stato minimo comune richiesto dal motore di gioco per gestire aggiornamenti e rendering.

Gestione dello Stato Spaziale La classe mantiene le proprietà fisiche fondamentali:

- **Posizione e Dimensione:** Le coordinate `x`, `y` e le dimensioni `width`, `height` sono memorizzate come interi. Sebbene le sottoclassi come `Bald` utilizzino calcoli a

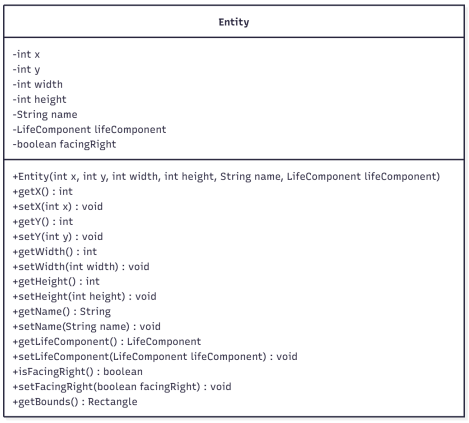

Figura 1: Classe Entity.java

virgola mobile per il movimento, `Entity` fornisce la rappresentazione "finale" discreta necessaria per il rendering su griglia e la rilevazione delle collisioni base.

- **Direzione:** Il campo booleano `facingRight` memorizza l'orientamento dell'entità. Questo stato è cruciale per il metodo `render` delle sottoclassi, che lo utilizzano per decidere se specchiare (flip) orizzontalmente la sprite.

Pattern Composizione per la Logica Vitale Una decisione architetturale chiave è stata l'adozione della **Composizione** invece dell'Ereditarietà per la gestione della salute. Invece di estendere una ipotetica superclasse *LivingEntity*, `Entity` possiede un riferimento a un'istanza di `LifeComponent`.

- **Incapsulamento:** La logica di danno, cura e morte è isolata nel componente, prevenendo il "bloating" (eccessiva crescita) della classe base.
- **Esposizione Controllata e SpotBugs:** Il metodo `getLifeComponent()` restituisce il riferimento all'oggetto originale, non una copia. Questa scelta è intenzionale e documentata tramite l'annotazione `@SuppressFBWarnings("EI_EXPOSE_REP")`. L'esposizione del riferimento è strettamente necessaria per permettere alla GUI (nello specifico `LifePanel`) di registrarsi come *PropertyChangeListener* e reagire in tempo reale ai cambiamenti di salute (Pattern Observer).

Interfaccia Fisica Il metodo `getBounds()` fornisce l'implementazione di default per la fisica, restituendo un `Rectangle` basato esattamente sulla posizione e sulle dimensioni della sprite (x, y, w, h). Le sottoclassi specializzate (come `Bald` e `FinalBoss`) effettuano l'*override* di questo metodo per implementare hitbox ridotte e disaccoppiate, mentre entità più semplici (come i proiettili) possono utilizzare questa implementazione base.

Questa classe è il genitore di tutte le entità dinamiche e gestisce le proprietà spaziali e vitali comuni, viene estesa dalle seguenti classi:

- **Bald.java**: gestisce l'input utente e introduce logiche cruciali per l'interazione fisica.
- **DummyEnemy**: base per i nemici comuni e introduce il controllo dello stato per le animazioni.
- **FinalBoss**: Il Boss è l'entità più complessa, introducendo una logica basata sulle fasi.

3.3.2 La Classe *Bald* (Player)

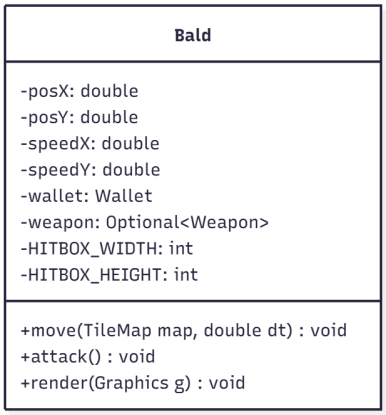

Figura 2: Classe Entity.java

La classe `Bald` rappresenta l'entità protagonista del gioco ed è il punto di convergenza tra l'input dell'utente e la logica del mondo di gioco. È una classe concreta che estende la classe astratta `Entity` e implementa l'interfaccia `Combatant`, definendo le meccaniche di movimento, combattimento e interazione.

L'implementazione di `Bald` si distingue per una gestione avanzata della fisica e del rendering, mirata a risolvere le problematiche tipiche dei giochi 2D *top-down*.

Fisica a Doppia Precisione Sebbene il rendering su schermo avvenga su pixel discreti (interi), la logica di movimento di `Bald` opera su valori a virgola mobile (`double`) per garantire fluidità e indipendenza dal frame rate.

- **Coordinate `posX`, `posY`:** Mantengono la posizione "reale" dell'entità con precisione sub-pixel. Ad ogni ciclo di `update`, la nuova posizione viene calcolata sommando il vettore velocità (`speedX`, `speedY`) moltiplicato per il *delta time* e un moltiplicatore di velocità (`SPEED_MULTIPLIER`).
- **Sincronizzazione:** Le coordinate intere ereditarie (`x`, `y`), utilizzate dalla superclasse per il disegno, vengono aggiornate solo al termine del calcolo fisico tramite un arrotondamento matematico (`(int) (val + 0.5)`), prevenendo il *jittering* visivo durante movimenti lenti.

Disaccoppiamento Hitbox e Rendering Uno degli aspetti critici risolti in questa classe è la discrepanza tra l'area visiva della sprite e l'area logica di collisione.

- **Sprite vs Hitbox:** La classe gestisce risorse grafiche di dimensione 50×50 pixel (`FRAME_WIDTH/HEIGHT`), ma definisce una *hitbox* di collisione molto più piccola, di 15×25 pixel (`HITBOX_WIDTH/HEIGHT`).
- **Allineamento al "Suolo":** Il metodo `getBounds()` sovrascrive quello della classe padre per calcolare un rettangolo di collisione traslato rispetto alla sprite. Mentre l'offset orizzontale centra la hitbox, l'offset verticale è calcolato come:

  $$Offset_Y = \texttt{ENTITY\_SIZE} - \texttt{HITBOX\_HEIGHT}$$

  Questo "ancora" la hitbox alla base della sprite (i piedi del personaggio), garantendo che le collisioni con muri e nemici avvengano in modo prospetticamente corretto per una visuale dall'alto, evitando che la testa del personaggio collida con ostacoli che dovrebbero essere "dietro" di lui.

Gestione degli Stati e Concorrenza La classe gestisce internamente macchine a stati implicite per le azioni del giocatore:

- **Attacco:** La gestione dell'attacco è supportata da una `Map` che associa il nome dell'arma (es. "sword", "axe") a specifici array di `BufferedImage`. Quando il giocatore attacca, il metodo `render` interrompe il ciclo di animazione della corsa per riprodurre la sequenza di attacco specifica dell'arma equipaggiata (`Optional<Weapon>`).
- **Immobilizzazione Asincrona:** Per gestire effetti di stato come lo stordimento o il blocco temporaneo, `Bald` utilizza un `ScheduledExecutorService` statico dedicato. Il metodo `immobilize(long duration)` imposta un flag booleano che inibisce il movimento nel metodo `move()` e pianifica il ripristino dello stato tramite un thread daemon, evitando di bloccare il *Game Loop* principale con operazioni di `Thread.sleep()`.

Composizione delle Risorse In linea con l'architettura del progetto, `Bald` agisce come un container composito per sottosistemi specifici:

- `Wallet`: Gestisce la valuta di gioco raccolta.
- `StatusEffectManager`: Un componente dedicato che calcola i modificatori temporanei alle statistiche (come l'`attackPower`).
- `LifeComponent`: Ereditato logicamente tramite composizione dalla superclasse, gestisce la salute e la morte.

3.3.3 La Classe *DummyEnemy* (Nemico Base)

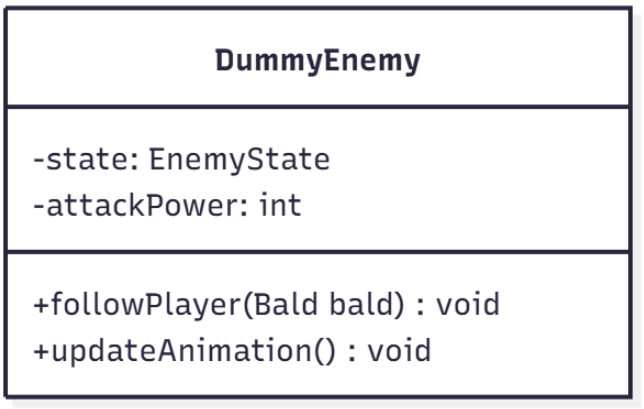

Figura 3: Classe Enemy.java

La classe `DummyEnemy` rappresenta l'antagonista standard del gioco. Estende `Entity` e implementa `Combatant`, ma a differenza del giocatore, la sua logica di controllo non deriva dall'input esterno ma da una gestione autonoma basata su stati.

Macchina a Stati Finiti (FSM) Il comportamento del nemico è governato da una Macchina a Stati Finiti interna, definita dall'enumerazione `EnemyState`. Questo approccio garantisce che l'entità possa eseguire una sola azione logica alla volta, prevenendo conflitti (es. muoversi mentre si subisce danno).

- `RUNNING`: Lo stato di default. Il nemico insegue attivamente il giocatore.
- `HURT`: Stato temporaneo attivato quando il nemico subisce danno ma sopravvive. Interrompe il movimento per fornire un feedback visivo ("hit stun").
- `DYING`: Stato di transizione attivato quando la salute scende a zero.
- `DEAD`: Stato finale inerte.

Ciclo di Vita e Animazioni Il metodo `updateAnimation()` agisce come il motore della FSM, gestendo le transizioni tra gli stati in base al completamento dei cicli di animazione:

- **Reazione al Danno:** Se colpito (`takeDamage`), l'entità passa a `HURT`. Una volta terminata la sequenza di frame dell'animazione "ferito", lo stato ritorna automaticamente a `RUNNING`, riattivando l'IA.
- **Gestione della Morte:** Se la salute si esaurisce, si passa a `DYING`. Al termine dell'animazione di morte, lo stato diventa `DEAD`.

- **Rimozione Ritardata:** Per evitare che il nemico scompaia istantaneamente alla morte, la classe distingue tra `isAlive()` (che ritorna *false* appena inizia l'animazione di morte) e `isRemovable()` (che ritorna *true* solo nello stato `DEAD`). Il *Game Loop* rimuove l'entità dalla lista di gioco solo quando `isRemovable()` è vero, garantendo la piena visibilità dell'animazione di sconfitta.

Intelligenza Artificiale (IA) e Movimento L'IA implementata nel metodo `followPlayer` è di tipo *Reactive*: calcola il vettore distanza tra il nemico e il giocatore (`Bald`) ad ogni frame.

- **Inseguimento Condizionato:** Il movimento avviene solo se lo stato è `RUNNING`. Questo crea un gameplay tattico dove il giocatore può "bloccare" l'avanzata dei nemici colpendoli (stunlock).
- **Pathfinding Diretto:** L'algoritmo muove l'entità sugli assi X e Y per minimizzare la distanza, utilizzando il sistema di collisione basato su tile (`moveWithCollision`) per navigare intorno agli ostacoli semplici.

3.3.4 La Classe *FinalBoss* (Boss Finale)

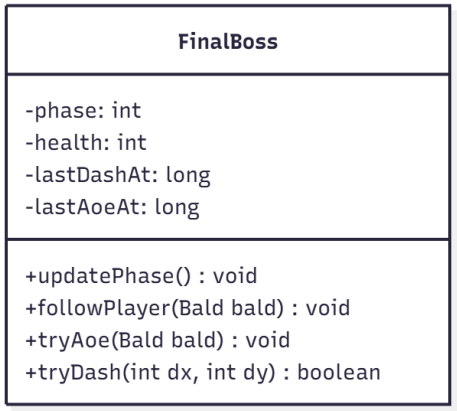

Figura 4: Classe Boss.java

La classe `FinalBoss.java` rappresenta l'antagonista principale e costituisce l'entità più complessa del sistema di gioco. Estende *Entity* e implementa *Combatant*, ma si distingue per l'adozione di una logica comportamentale non lineare basata su fasi progressive.

Sistema a Fasi Dinamiche (Multi-Stage Logic) A differenza dei nemici standard, il comportamento del Boss non è statico ma evolve in base allo stato della battaglia. Il metodo `updatePhase()` monitora costantemente il rapporto tra salute corrente e massima:

- **Soglie di Attivazione:** Sono definite due soglie critiche (`PHASE2_THRESHOLD` al 66% e `PHASE3_THRESHOLD` al 33%).

- **Scaling delle Statistiche:** Il passaggio di fase innesca un aumento immediato della difficoltà. Il metodo `getAttackPower()` applica un moltiplicatore (`PHASE_ATK_MULT`, fino a $1.5x$ nella fase 3), mentre `getCurrentSpeed()` incrementa la velocità di movimento (da 2 a 4 pixel/frame).

Intelligenza Artificiale e Abilità L'IA implementata in `followPlayer` utilizza un albero decisionale prioritario basato sulla distanza dal giocatore (`dist`):

1. **Aggro Check:** L'entità rimane inattiva finché il giocatore non entra nel raggio visivo (`AGGRO_RANGE_PX`).
2. **Combattimento Ravvicinato:** Se in range *melee* (< 48px), il Boss tenta un attacco base (soggetto a `MELEE_COOLDOWN_MS`) e, simultaneamente, un attacco ad area (`tryAoe`). L'AOE infligge danni extra e punisce il giocatore che rimane troppo vicino troppo a lungo.
3. **Ingaggio a Distanza (Dash):** Se il giocatore è lontano ma visibile, il Boss tenta di eseguire un *Dash*. Questa abilità proietta l'entità velocemente verso il bersaglio ignorando la velocità di movimento base, riducendo rapidamente il divario (gap-closer).
4. **Inseguimento Standard:** Se nessuna abilità è disponibile, il Boss utilizza il movimento standard con collisioni sui tile per raggiungere il giocatore.

Rendering Multi-Asset Per fornire un feedback visivo chiaro sulla progressione dello scontro, la classe gestisce tre set distinti di risorse grafiche (`phase1Frames`, `phase2Frames`, `phase3Frames`). Il metodo `render` non si limita a disegnare il frame corrente, ma seleziona dinamicamente l'array di sprite corretto interrogando la fase attiva (`getActiveFrames()`). Questo permette al Boss di cambiare aspetto (es. diventare più minaccioso) man mano che la sua salute diminuisce, senza necessitare di classi separate.

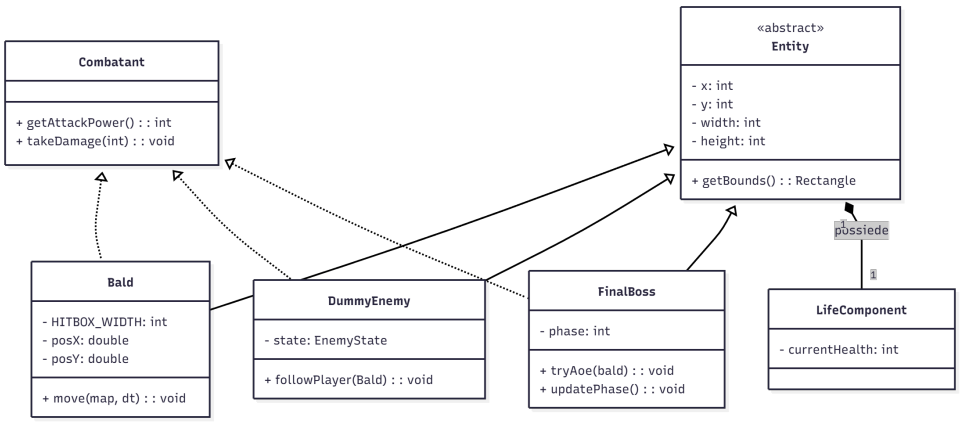

Figura 5: UML fatto con MermaidChard.

3.3.5 Architettura del Sottosistema di Controllo e Servizi

La gestione del ciclo di gioco (Game Loop) e l'elaborazione degli input sono state completamente delegate a classi specializzate per garantire una rigida adesione al principio di **Separazione delle Responsabilità (SRP)** e disaccoppiare il Model dalla View.

1. La Classe `GamePanel` (Orchestratore della Vista) Il ruolo del `GamePanel` è stato ridefinito. Non gestisce più logica interna, ma funge da **View Orchestrator**. La sua responsabilità è avviare il `GameEngine` (implementato su un thread separato), gestire la composizione dei pannelli UI/HUD e supervisionare il rendering. Delega tutte le decisioni di stato e di input ai rispettivi controller e manager.

2. La Classe `LevelManager` (Gestore dello Stato del Mondo) La classe `LevelManager` è l'orchestratore dello stato del livello. Essa incapsula e coordina tutte le componenti relative al mondo di gioco: mantiene i riferimenti alla `TileMap`, al `CombatManager` e alle liste degli attori. È l'unica entità responsabile della logica complessa di **transizione tra mappe** (`changeAndLoadMap`), gestendo il riposizionamento del giocatore, lo spawn degli attori e l'aggiornamento degli oggetti ad ogni cambio di livello.

3. La Classe `InputController` (Traduttore Input) La classe `InputController` implementa lo strato **Controller**. Non contiene logica di movimento o di gioco, ma è responsabile unicamente di catturare gli eventi del mouse e della tastiera tramite `InputMap` e `ActionMap`. Il suo compito è tradurre questi eventi in chiamate API sul Model:

- Calcola il vettore velocità (`speedX/Y`) e la direzione del giocatore (`Bald`).
- Delega gli attacchi al `CombatManager`.
- Attiva le interazioni con gli oggetti e i toggle dei menu.

Questo disaccoppiamento permette di cambiare il tipo di input (es. controller di gioco) senza dover modificare le classi del Model.

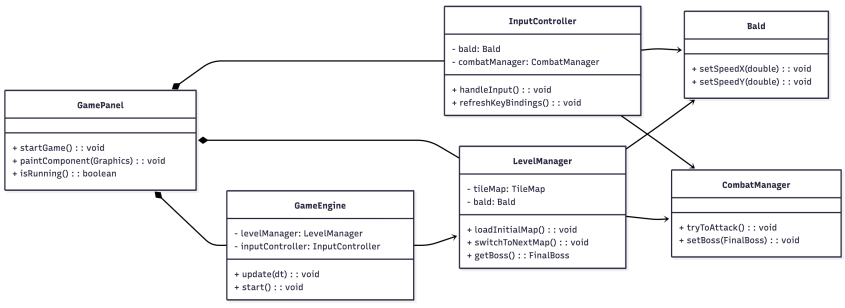

Figura 6: UML fatto con MermaidChard.

3.4 Gerarchia delle Interfacce per le View

Questo diagramma UML illustra la gerarchia delle interfacce *View* di un'applicazione, mostrando come il design segua i principi della programmazione orientata agli oggetti per la separazione delle responsabilità e l'estensibilità.

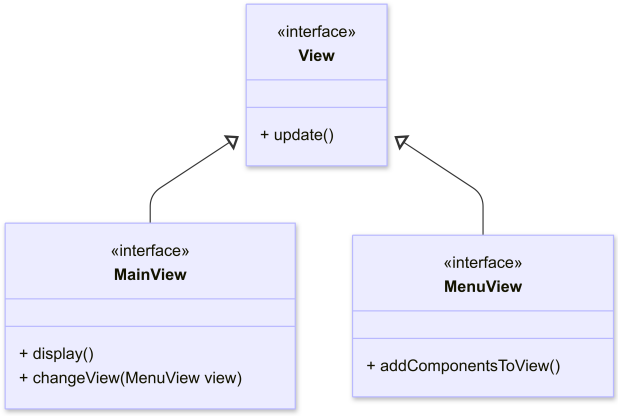

Figura 7: UML fatto con MermaidChard.

Il design si basa sul principio della generalizzazione, dove l'interfaccia *View* definisce il contratto comune con il metodo *update()*, tipico del design pattern **Observer**. Le interfacce *MainView* e *MenuView* ereditano da *View*, specializzando il suo comportamento con metodi specifici come *display()* e *addComponentsToView()*, rispettivamente. Questa architettura consente una gestione modulare e flessibile delle diverse componenti dell'interfaccia utente.

3.5 Architettura della Gestione delle View

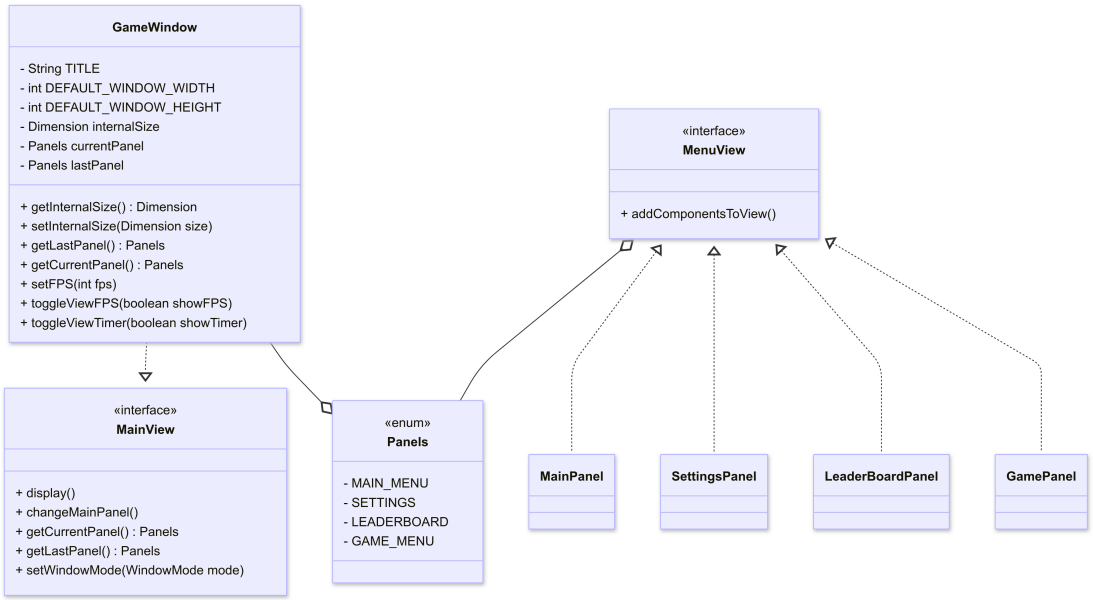

Figura 8: UML fatto con MermaidChard.

3.5.1 Principi di Design e Pattern Applicati

Il design della gestione delle interfacce utente (view) del progetto si basa su diversi principi e pattern architetturali per garantire modularità, flessibilità e manutenibilità.

- **Inversion of Control (IoC) e Dependency Inversion Principle (DIP)**: Per disaccoppiare la classe *GameWindow* dalle implementazioni concrete delle diverse viste (come *MainPanel*, *SettingsPanel*, ecc.), si fa uso di interfacce astratte. Nello specifico, *GameWindow* interagisce con l'interfaccia *MainView*, delegando la gestione delle viste specifiche a classi che la implementano. Questo riduce le dipendenze dirette e rende più semplice aggiungere o modificare nuove viste senza impattare la classe principale.

- **Strategy Pattern**: Questo pattern è applicato per incapsulare i diversi algoritmi di visualizzazione. Ciascuna classe di vista (ad esempio *MainPanel*, *GamePanel*) rappresenta una "strategia" specifica per come l'interfaccia utente dovrebbe essere mostrata e gestita. La classe *GameWindow* agisce come "contesto" che, attraverso l'interfaccia *MainView*, può cambiare dinamicamente il comportamento di visualizzazione scegliendo tra le diverse strategie disponibili (cambiando la *currentPanel*).

- **State Pattern**: Sebbene il diagramma non lo illustri esplicitamente, la logica di cambio di stato tra le diverse viste (*MAIN_MENU*, *SETTINGS*, *GAME_MENU*, ecc.) gestita dall'enumerazione *Panels* può essere vista come un'applicazione del **State**

Pattern. La classe *GameWindow* si trova in uno stato specifico (es. *MAIN_MENU*) e il suo comportamento (la view che mostra) cambia in base a tale stato, con la transizione da uno stato all'altro (es. da *MAIN_MENU* a *GAME_MENU*) che avviene in risposta ad azioni dell'utente.

3.6 Effetti di Stato

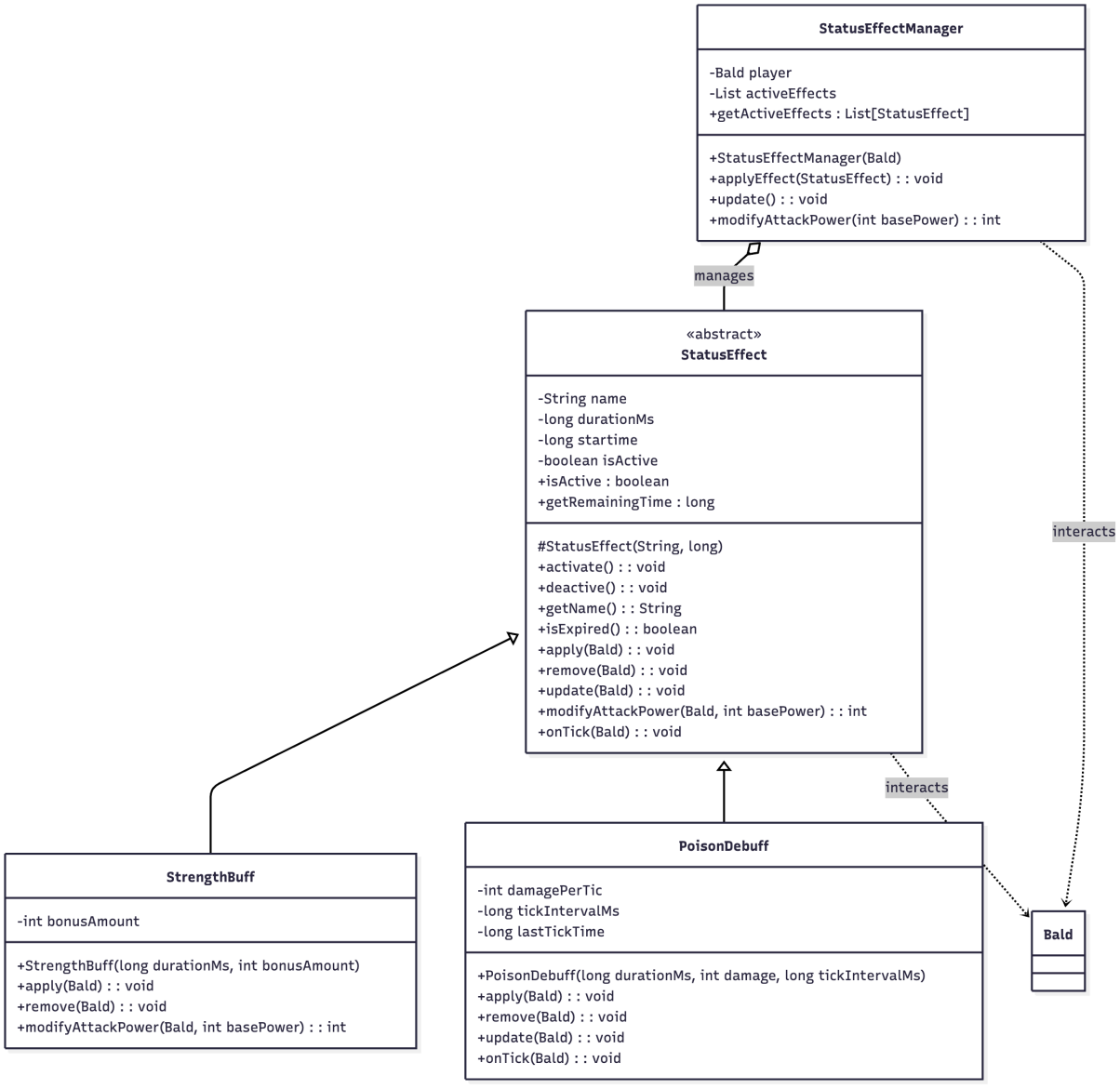

Figura 9: UML fatto con MermaidChard.

3.6.1 Pattern Ereditarietà e Specializzazione

Il design del sistema degli effetti di stato si basa su una chiara **gerarchia di ereditarietà** per definire il comportamento di base.

- **Classe Astratta StatusEffect**: Serve come radice della gerarchia (visibile nella Figura 9). Definisce uno **scheletro comune** per tutti gli effetti, definendo la logica concreta del ciclo di vita (es. `activate()`, `isExpired()`, gestione del tempo di `durationMs`).

- **Specializzazione del Comportamento (Polimorfismo)**: Per i comportamenti che variano, la classe *StatusEffect* definisce metodi astratti come `apply(Bald)` e `remove(Bald)`, che fungono da **Metodi Primitivi**. Questo garantisce che ogni sottoclasse si specializzi implementando:

  - La logica di Danno nel Tempo (DoT) tramite *PoisonDebuff*.
  - La modifica passiva delle statistiche tramite *StrengthBuff*.

- **Vantaggio del Design**: L'ereditarietà garantisce che tutte le classi derivate (Buff o Debuff) ereditino e utilizzino lo stesso **meccanismo di gestione del tempo** e di attivazione/disattivazione, riducendo la duplicazione del codice e centralizzando la logica di base degli effetti.

3.6.2 Pattern Aggregazione e delegazione

La classe **StatusEffectManager** implementa il pattern di **Aggregazione** mantenendo una collezione (`List<StatusEffect>`) di effetti attivi, non li crea ma li gestisce.

- **Aggregazione**: La classe *StatusEffectManager* è l'unica classe per la gestione logica e rimozione degli effetti di stato.

- **Delega (Delegation)**: La Composizione (o Aggregazione) permette allo *StatusEffectManager* di **delegare** le complesse logiche di funzionamento e di calcolo dei parametri ai singoli oggetti *StatusEffect* (es. *PoisonDebuff* o *StrengthBuff*). Invece di gestire i timer di tick, i calcoli del Danno nel Tempo (DoT) o la modifica delle statistiche del giocatore direttamente all'interno della classe *StatusEffectManager*, questa delega tale responsabilità all'effetto specifico attraverso i metodi polimorfici come `update()` e `modifyAttackPower()`.

- **Single Responcibility Principle - SRP**: Il design iniziale prevedeva di mantenere una lista di effettivi attivi dentro l'Entità Bald.Tuttavia con la creazione della classe *StatusEffectManager* si è potuta spostare la responsabilità degli effetti di stato.

3.7 Gestione e Architettura della Tilemap

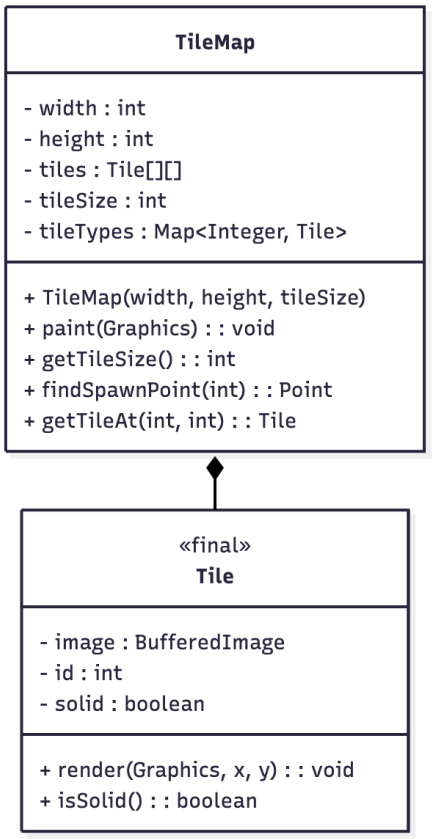

Figura 10: UML fatto con MermaidChard.

3.7.1 Design della Tilemap

Questa sottosezione illustra la **struttura dati fondamentale** e il modello di Tile. La classe `TileMap` gestisce la mappa come una matrice bidimensionale di oggetti `Tile` (`Tile[][] tiles`). Ogni oggetto `Tile` (classe `Tile`) incapsula le proprietà del singolo elemento del mondo, inclusi l'ID (da `ID_EMPTY` a `ID_NEXT_MAP_TRIGGER`), l'immagine di base, una potenziale immagine di *overlay* e i metadati fondamentali (`isSolid`, `isWalkable`, `isSpawn`).

Strumenti e Formato Dati. La creazione delle mappe è basata su un approccio di **level editing testuale**. La mappa viene salvata in file **CSV/TXT** (e.g., `map_1.txt`) all'interno della cartella `/resources/map`. Il formato di esportazione è una semplice matrice di **ID interi** separati da spazi o virgole. L'implementazione evita l'uso di editor esterni complessi, gestendo il *parsing* dei file TXT direttamente tramite `BufferedReader` e `InputStreamReader` nella funzione `loadMapFromFile`, garantendo un caricamento snello e aderente alla struttura `java.awt`.

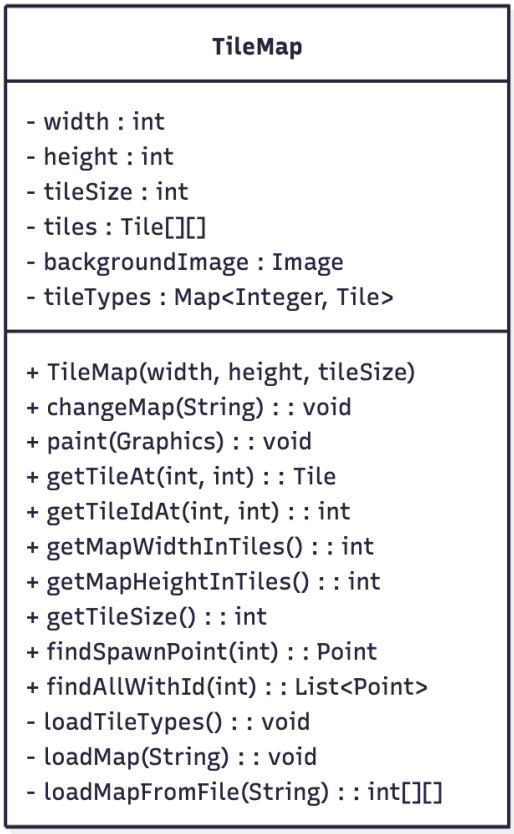

Figura 11: UML fatto con MermaidChard.

3.7.2 La Classe Tile

La classe `Tile` è un'entità **final** e immutabile che rappresenta il blocco costruttivo fondamentale della mappa. Non è pensata per essere estesa, garantendo coerenza e sicurezza dei dati.

Proprietà e Metadati. Ogni istanza di `Tile` incapsula le proprietà necessarie per il rendering e la logica di gioco:

- **Immagine:** Contiene l'`Image` di base e un'immagine opzionale di *overlay* (`overlayImage`), gestite come `BufferedImage`. Il costruttore gestisce il **ridimensionamento** e la **copia difensiva** (`deepCopy`) delle immagini per garantire l'incapsulamento e l'immutabilità.
- **Identificativo:** L'ID (`id`) numerico viene utilizzato dalla `TileMap` per mappare i dati grezzi del file TXT al tipo di tile.
- **Logica:** Metadati boolean essenziali per le interazioni, ovvero `solid` (bloccante), `walkable` (attraversabile) e `isSpawn` (punto di generazione).

Rendering. Il rendering del singolo tile è delegato al metodo `render(Graphics g, int x, int y)`, che disegna l'immagine di base e, successivamente, l'eventuale *overlay image* alle coordinate specificate.

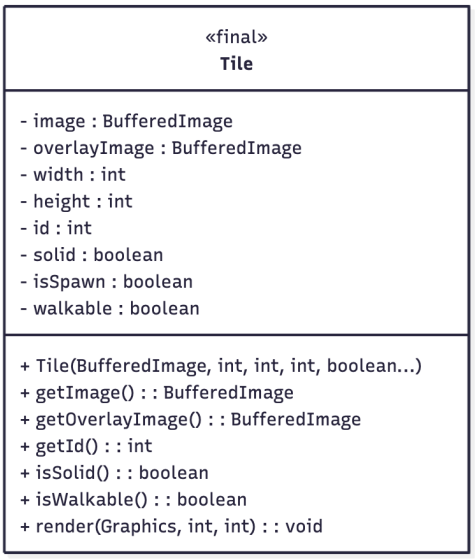

Figura 12: UML fatto con MermaidChard.

3.7.3 Caricamento e rendering della Mappa

Il processo di caricamento si articola in due fasi principali. Inizialmente, la funzione `loadTileTypes` precarica le **immagini base** dei tile (`BufferedImage`) tramite `ImageIO.read` e popola una `HashMap<Integer, Tile> tileTypes` che funge da *factory* per i tile. Successivamente, la funzione `loadMapFromFile` legge la matrice di ID dal file di testo e istanzia la matrice `tiles` clonando le proprietà dai `tileTypes` precaricati.

Il rendering è gestito dal metodo `paint` della `TileMap`, che opera in un contesto `Graphics`. Il processo è sequenziale: prima viene disegnata l'eventuale **immagine di sfondo** (`backgroundImage`), quindi la mappa viene renderizzata iterando riga per riga e colonna per colonna (*row-by-row, column-by-column*). La classe `Tile` gestisce il disegno dell'immagine base e dell'*overlay* separatamente, senza implementare tecniche avanzate di *culling* (poiché il rendering è basato sul contesto `AWT/Swing` e disegna l'intera matrice visibile).

Il Pannello di Debug GridPanel La classe `GridPanel` estende `JPanel` ed è una componente **final** e puramente di *view*, responsabile unicamente del disegno di una griglia di overlay. Il suo scopo è fornire un ausilio visivo (tipicamente per il debug) che mostra i confini esatti dei tile. Il rendering è implementato nel metodo `drawGrid(Graphics g)` che utilizza un colore semitrasparente. Nella `GamePanel`, viene disegnata subito dopo la `TileMap` (`tileMap.paint(g2d)`).

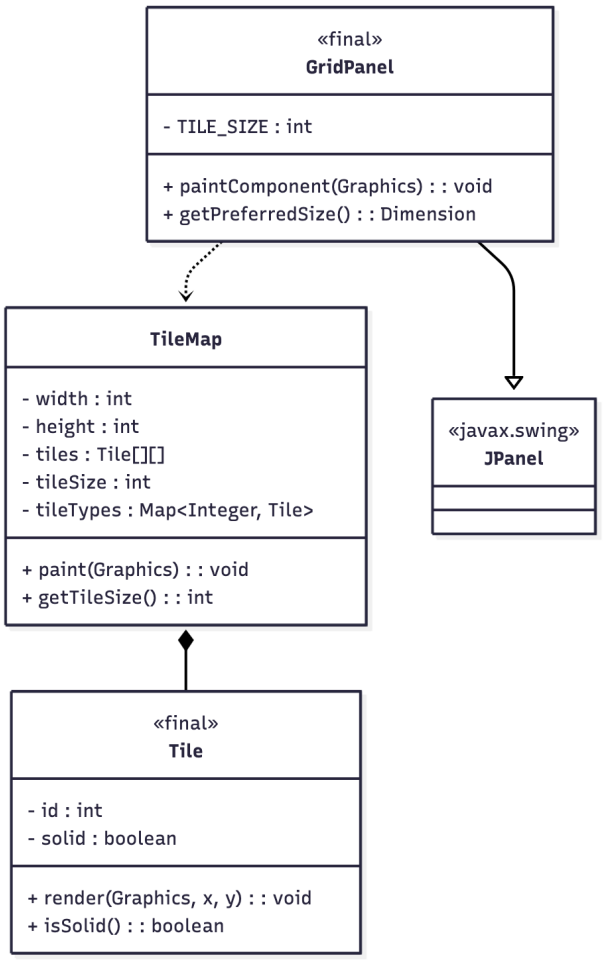

Figura 13: UML fatto con MermaidChard.

3.7.4 Collisioni con la Tilemap

Il sistema di collisione e navigabilità è basato interamente sui **metadati binari** contenuti nell'oggetto `Tile`.

- **Solidità/Blocco:** La proprietà `isSolid()` è impostata a `true` per i tile come `ID_WALL`. Questa proprietà viene interrogata dalle entità di gioco tramite `getTileAt(x, y).isSolid()` per prevenire il movimento in aree bloccate.
- **Navigabilità:** La proprietà `isWalkable()` definisce se il tile può essere attraversato.

L'interazione è gestita tramite la funzione `getTileAt(x, y)`, che restituisce l'oggetto `Tile` nella posizione di griglia desiderata, consentendo al sistema di gioco (non implementato in questa classe) di eseguire *query* precise sulle proprietà di collisione prima di aggiornare la posizione dei personaggi. La `Tilemap` fornisce anche metodi per trovare punti di interesse (e.g., `findSpawnPoint` per `ID_SPAWN` o `findAllWithId` per gli oggetti).

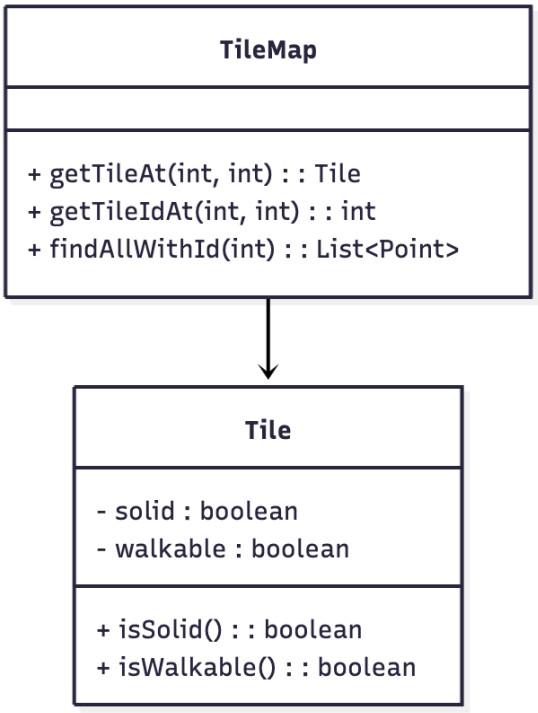

Figura 14: UML fatto con MermaidChard.

3.8 Gestione degli Oggetti di Gioco

Il diagramma UML allegato illustra la struttura delle classi per la gestione degli oggetti di gioco, applicando diversi **design patterns** per creare un'architettura flessibile ed estendibile.

3.8.1 Pattern Ereditarietà e Specializzazione

Il design si basa su una **gerarchia di ereditarietà**. La classe astratta *GameItem* serve come radice, definendo un oggetto generico nel gioco.Da essa abbiamo varie classi astratte, come *Potion e Trap*, le quali definiscono proprietà e comportamenti comuni per le loro sottoclassi.

Specializzazione

- **Classe Astratta Potion** : La classe astratta *Potion* implementa l'interfaccia *UsableItem*. Questo design stabilisce un **contratto** che obbliga tutte le pozioni concrete a definire la propria logica di utilizzo tramite il metodo *applyEffect(Bald player)*.
- **Classi Concrete**: Le classi concrete come *HealthPotion* e *StrengthPotion* **ereditano da *GameItem* (tramite *Potion*)**, completando la catena di ereditarietà. Ciascuna

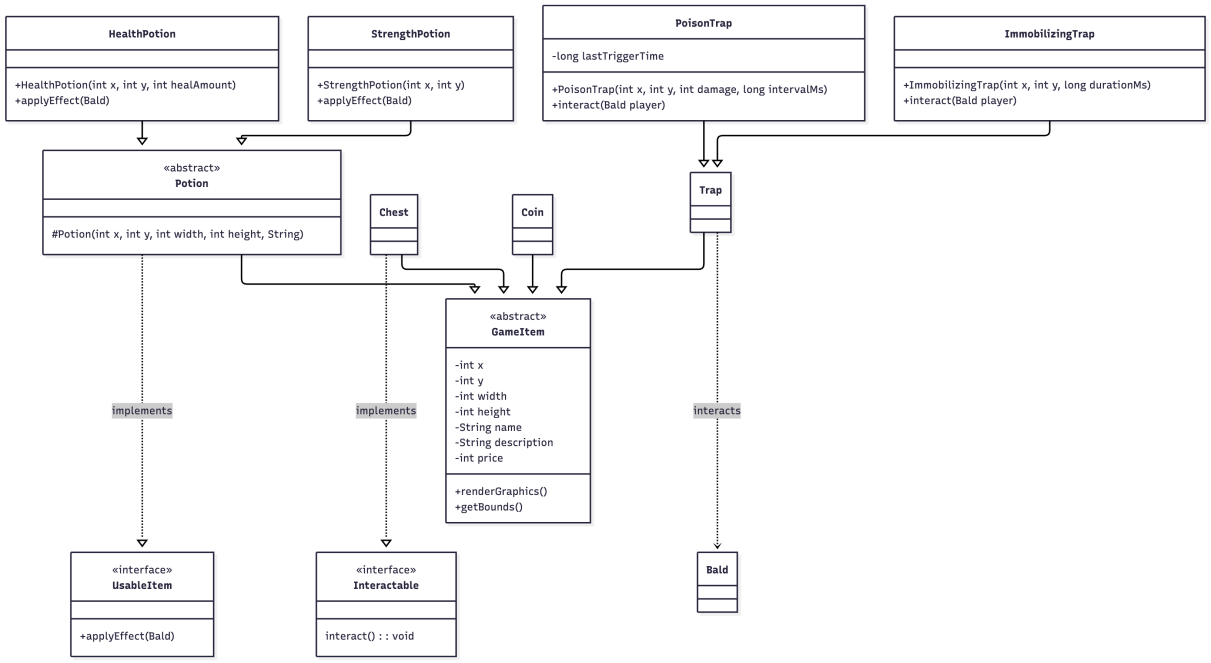

Figura 15: UML fatto con MermaidChard.

di esse fornisce i dettagli specifici dell'effetto e porta a termine l'implementazione del metodo *applyEffect*, definendo l'oggetto utilizzabile finale.

- **Classe Astratta Trap**: Definisce lo scheletro comune, gestendo proprietà come *triggered* e *removeOnTrigger*, e implementando metodi di utilità come *render()*. Il metodo astratto `public abstract void interact(Bald player)` definisce un **contratto** che obbliga tutte le specializzazioni della classe Trap ad implementare quel metodo.Le motivazioni della scelta del metodo astratto rispetto ad un'interfaccia è dovuta dall'implementazione della classe *ImmobilizingTrap* , dove viene chiamato i metodi *isTriggered() e SetTriggered()* della classe padre *Trap*.

- **Classe Concrete**: Le classi *PoisonTrap e ImmobilizingTrap* ereditano lo scheletro della classe *Trap* e fornisce l'implementazione concreta per il metodo *interact()* per l'interazione tra la trappola ed il giocatore.

- **Altre classi concrete**: *Chest* che implementa *Interactable* per definire un contratto per l'interazione con il giocatore.*Coin*,che è la valuta del gioco, riferendosi alla classe Wallet.

3.8.2 Pattern Composizione

- **Delegazione**: La relazione di gestione e controllo (Aggregazione) permette al controller di livello superiore (es. *GamePanel*) di **delegare** tutte le logiche degli oggetti al **ItemManager** (es. aggiornamento, rendering e gestione delle collisioni).

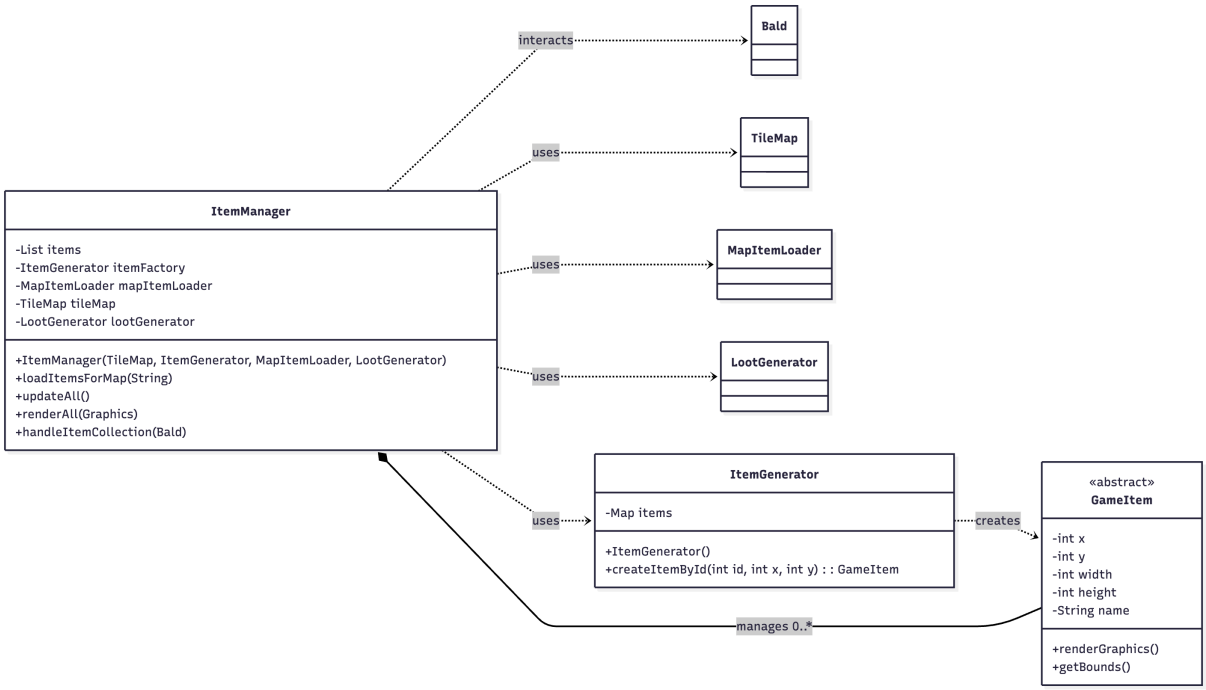

Figura 16: UML fatto con MermaidChard.

- **Separazione delle Responsabilità**: Il design iniziale è stato modificato per separare la gestione della logica degli oggetti dal *GamePanel*, creando la classe **ItemManager** in modo da approcciare il principio della **singola responsabilità** (Single Responsibility Principle - SRP).

3.8.3 Gestione della Generazione degli Oggetti di gioco nella mappa

La gestione della generazione degli oggetti di gioco nella mappa sfrutta tre classi : **MapItemSpawnData , MapItemLoader e MapItemGenerator**.

- **ItemSpawnData**: Agisce come un **Data Transfer Object (DTO)**. La sua unica responsabilità è incapsulare i dati grezzi (ID, riga, colonna) necessari per la creazione di un oggetto, mantenendo lo stato immutabile.

- **MapItemLoader**: È responsabile della gestione dell'I/O (Input/Output). La sua funzione è leggere e validare il file di testo in formato **TXT**, contenente id e coordinate della posizione in base ai file di testo della mappa originale, convertendo ogni riga in una lista di oggetti *ItemSpawnData*. Questo disaccoppia la logica di parsing dalla logica di creazione degli oggetti.

- **MapItemSpawner**: Agisce come il **Director** o **Builder** del processo di generazione. Riceve i dati dal *Loader* e, utilizzando l'istanza di *ItemGenerator* (Factory), calcola le coordinate finali e istanzia gli oggetti *GameItem* concreti nella mappa.

3.9 Gerarchia delle Armi e del Sistema di Combattimento

Il diagramma UML allegato illustra la struttura delle classi per la gestione delle armi e del combattimento, applicando diversi **design patterns** per creare un'architettura flessibile ed estensibile.

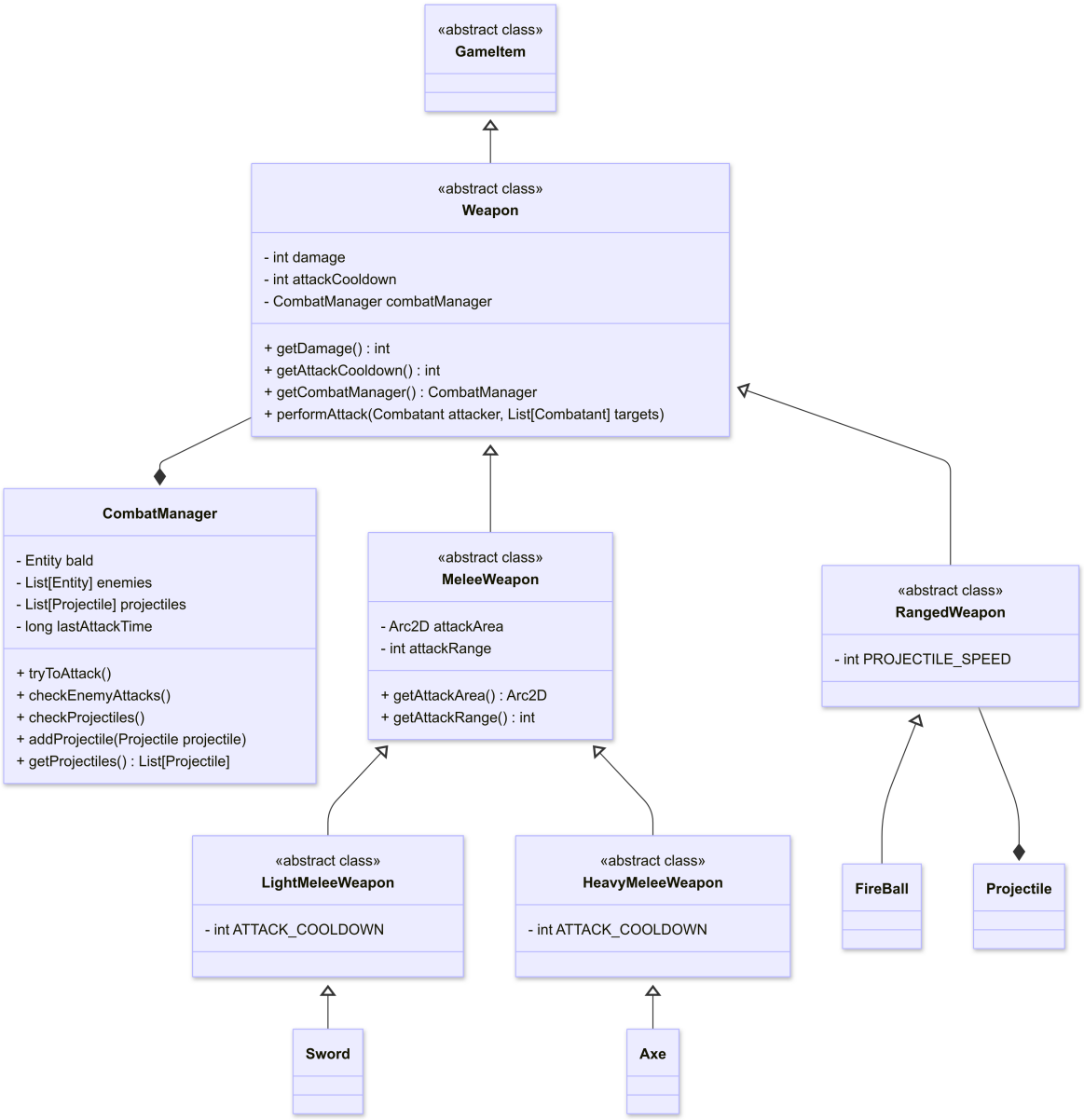

Figura 17: UML fatto con MermaidChard.

3.9.1 Pattern Ereditarietà e Specializzazione (Inheritance and Specialization)

Il design si basa su una **gerarchia di ereditarietà** chiara e funzionale. La classe astratta *GameItem* serve come radice, definendo un oggetto generico nel gioco. Da essa, la classe

astratta *Weapon* introduce le proprietà e i comportamenti comuni a tutte le armi, come il danno (*damage*) e il tempo di recupero tra gli attacchi (*attackCooldown*).

- **Specializzazione:** *Weapon* si specializza ulteriormente in *MeleeWeapon* e *RangedWeapon*, classi astratte che definiscono comportamenti specifici per armi da mischia e a distanza. Questo approccio non solo **riduce la duplicazione del codice** ma rende anche il sistema facile da estendere.
- **Classi Concrete:** Le classi concrete come *Sword*, *Axe*, *Fireball* e *Projectile* ereditano dalle rispettive classi astratte, implementando i dettagli specifici di ogni arma. L'ereditarietà *RangedWeapon* → *Fireball* è un esempio eccellente di come la classe *Fireball* possa essere trattata come un'arma a distanza.

Questo design segue il **principio Open-Closed**, che sostiene che le classi dovrebbero essere **aperte per l'estensione ma chiuse per la modifica**. Se si volesse aggiungere un nuovo tipo di arma, come una lancia o un'arma laser, lo si potrebbe fare semplicemente creando una nuova classe che eredita da *MeleeWeapon* o *RangedWeapon* senza dover alterare il codice esistente.

3.9.2 Pattern Composizione (Composition)

A differenza dell'ereditarietà che rappresenta una relazione "è un tipo di" (*is-a*), la **composizione** rappresenta una relazione "ha un" (*has-a*). La classe *Weapon* ha una relazione di composizione con la classe *CombatManager*, indicata dal rombo nero sul diagramma.

- **Delegazione:** La composizione permette a *Weapon* di **delegare** le complesse logiche di combattimento al *CombatManager*. Invece di gestire la logica dei proiettili o la verifica dei bersagli direttamente all'interno della classe *Weapon*, questa delega tale responsabilità al *CombatManager* attraverso il metodo *performAttack()*.
- **Separazione delle Responsabilità:** Questo approccio aderisce al principio della **singola responsabilità** (Single Responsibility Principle - SRP), dividendo il comportamento. La classe *Weapon* si concentra sui suoi attributi intrinseci (danno, cooldown), mentre *CombatManager* si occupa della logica di battaglia e dell'interazione con l'ambiente del gioco.

3.10 Struttura Principale dell'Inventario

Il diagramma UML illustra la struttura delle classi e interfacce per la gestione di un sistema di inventario in un'applicazione o videogioco, mettendo in evidenza l'utilizzo di specifici **design patterns** che promuovono la flessibilità e la separazione delle responsabilità.

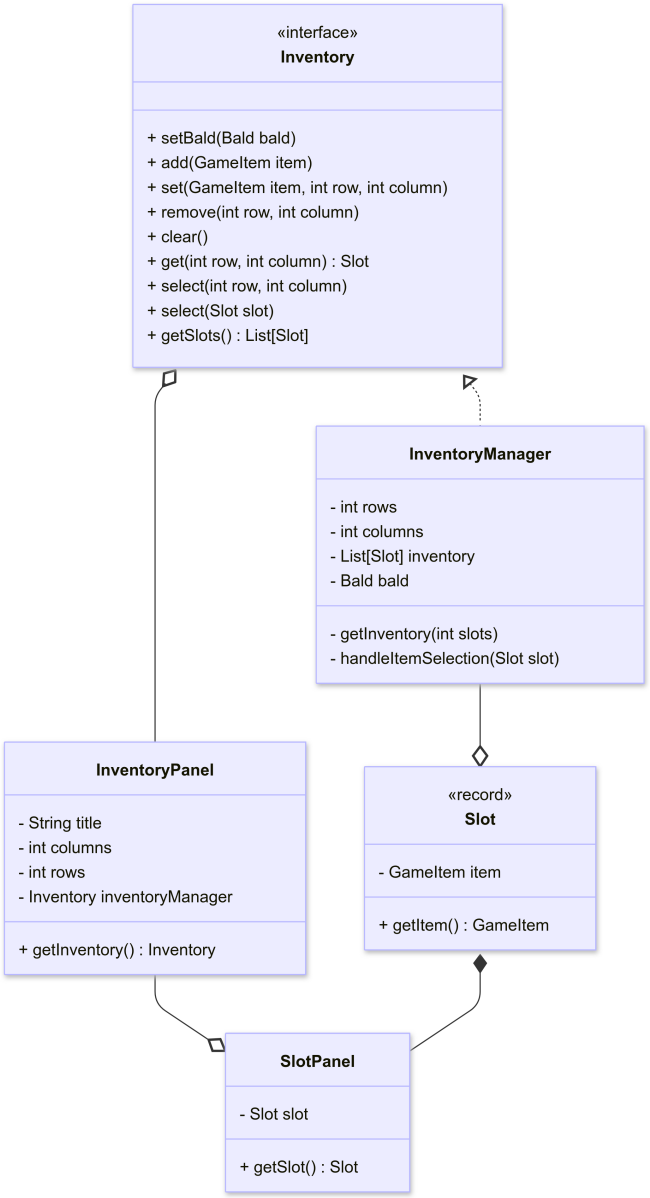

Figura 18: UML fatto con MermaidChard

3.10.1 Pattern di Progettazione

Questo sistema di inventario integra diversi principi di design.

- **Pattern Abstraction (Interfaccia)** Il design utilizza un'**interfaccia *Inventory*** per definire il **contratto** del comportamento di un inventario, specificando metodi come *add()*, *remove()*, *get()*, e *clear()*. Qualsiasi classe che implementi questa interfaccia deve fornire la propria implementazione di questi metodi. Questo approccio promuove il **disaccoppiamento** tra l'interfaccia (cosa un inventario fa) e la sua implementazione (come lo fa). Questo permette di scambiare l'implementazione dell'inventario in futuro, ad esempio, passando da un inventario basato su una lista a uno basato su un array, senza dover modificare il codice che interagisce con l'interfaccia.

- **Pattern di Composizione (Composition)** Il diagramma mostra diverse relazioni di **composizione**, indicate dal rombo nero, che rappresentano una relazione "ha un" (*has-a*).

  – *InventoryManager* ha una lista di *Slot*.

  – *InventoryPanel* ha un *Inventory*.

  – *SlotPanel* ha un *Slot*.

  La **composizione** è un'alternativa all'ereditarietà e viene utilizzata per costruire oggetti complessi a partire da oggetti più semplici. In questo caso, *InventoryPanel* non "è un tipo di" *Inventory*, ma "utilizza" un *Inventory* per delegare la gestione logica dell'inventario. Allo stesso modo, *SlotPanel* utilizza un *Slot* per rappresentare visivamente l'item contenuto al suo interno. Questo approccio rispetta il principio del **disaccoppiamento** e della **singola responsabilità** (Single Responsibility Principle - SRP), dove ogni classe ha un solo motivo per cambiare.

3.10.2 Implementazione del Pannello Negozio (ShopPanel)

La classe `ShopPanel` implementa l'interfaccia utente per il negozio, consentendo al giocatore di acquistare oggetti in cambio di monete. Questa classe agisce da mediatore tra il modello economico (`Wallet`) e il modello degli oggetti (`Inventory` e `GameItem`).

Inizializzazione e Stato. Il pannello viene istanziato nella `GamePanel` e mostrato come finestra modale `JOptionPane` quando il giocatore interagisce con un tile `ID_SHOP`. Lo stato *runtime* viene inizializzato in `initRuntimeState`, dove vengono precaricati gli oggetti acquistabili (attualmente `Sword`, `Axe`, `FireBall`) e stabilito il riferimento al `Wallet` e all'`Inventory` del giocatore.

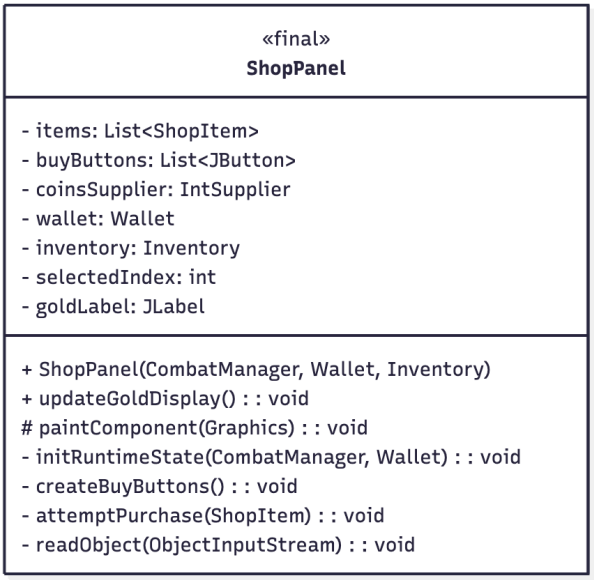

Figura 19: UML fatto con MermaidChard.

Logica di Acquisto. L'interazione è gestita tramite pulsanti "Buy Item" (`JButton`) associati a ciascun oggetto. Il metodo `attemptPurchase(ShopItem item)` gestisce la logica transazionale:

1. Verifica che il `Wallet` del giocatore contenga monete sufficienti.
2. In caso positivo, le monete vengono rimosse (`wallet.removeCoins(price)`).
3. L'oggetto viene aggiunto all'`Inventory` (`inventory.add(gameItem)`).

La visualizzazione dell'oro viene mantenuta aggiornata tramite `updateGoldDisplay` e un `JLabel`.

Rendering Custom. Il pannello estende `JPanel` e utilizza il metodo `paintComponent` per il rendering personalizzato (custom rendering). Questo include il disegno del titolo "Shop", la lista degli oggetti acquistabili con i loro prezzi, un effetto di selezione (`SELECT_COLOR`) basato sugli eventi del `MouseAdapter` e la descrizione dell'oggetto selezionato. L'uso di `Graphics2D` con `RenderingHints.KEY_ANTIALIASING` migliora la qualità visiva degli elementi disegnati.

4 Testing e validazione

4.1 Testing automatizzato

Diversi elementi del modello e della logica di gioco sono stati testati utilizzando JUnit per garantire il corretto funzionamento e la robustezza del codice. Di seguito sono elencate le principali classi di test e le loro finalità:

- **FinalBossTest:** È stata testata la corretta gestione del `FinalBoss`, incluse le transizioni di fase in base alla salute, la scalabilità della potenza d'attacco, la ricezione del danno e l'intelligenza artificiale di base per seguire il giocatore.
- **InventoryTest:** È stato verificato il corretto funzionamento dell'inventario, testando l'aggiunta, la rimozione e la gestione degli oggetti (`GameItem`) all'interno degli slot.
- **ItemTest:** È stata testata la creazione e l'utilizzo degli oggetti di gioco, assicurando che le loro proprietà ed effetti (come nel caso degli `UsableItem`) vengano applicati correttamente.
- **CombatTest:** Sono state verificate le meccaniche di combattimento fondamentali, come il calcolo del danno, l'applicazione degli attacchi tra entità e la gestione dei proiettili.
- **StatusEffectTest:** È stato testato il corretto funzionamento del sistema di effetti di stato (es. `PoisonDebuff`), verificando la loro applicazione, durata e l'impatto che hanno sulle statistiche dei personaggi.

5 Build & Run

5.1 Prerequisiti

Java 21 e Gradle wrapper incluso nel progetto.

5.2 Comandi

```
# Compilazione
./gradlew build

# Esecuzione (se il progetto espone un'app)
./gradlew run

# Esecuzione del jar (se configurato)
java -jar build/libs/the-legend-of-bald-<version>.jar
```

6 Commenti Finali e Riflessioni

6.1 Davide Magyari

Questo progetto è stata la mia prima esperienza di sviluppo software per un gioco, ho compreso che la coordinazione è fondamentale. Ci sono state fatiche e problematiche ma alla fine mi sono divertito, avrei preso altre scelte ma tutto sommato non rimpiango di aver partecipato a questo progetto. Ho imparato moltissimo a diversificare e rendere il codice più accessibile e manutenibile che penso sia inoltre una delle cose più importanti come idea da cui partire per iniziare a creare un gioco o proprio un software "grande" in generale. Ho legato moltissimo inoltre anche con i miei colleghi di progetto e ritengo che non sia una cosa scontata o per tutti.

6.2 Vincent Rey Ramos

Partecipare a questo progetto ha rappresentato la mia prima esperienza di sviluppo software in team e da zero.Nel corso del lavoro, ho compreso l'importanza cruciale della coordinazione e della comunicazione efficace per un ambiente di lavoro produttivo e armonioso. È stata un'esperienza formativa che mi ha permesso di identificare i comuni errori individuali che si possono commettere in un contesto collaborativo.Inoltre, mi ha fatto crescere notevolmente anche come sviluppatore. Sebbene progettare qualcosa ex novo risulti inizialmente complesso, ho imparato che, adottando le giuste metodologie di design e pratiche di sviluppo, l'implementazione successiva diventa significativamente più fluida.In sintesi, questa esperienza mi ha lasciato solide competenze nel lavoro di squadra e una chiara consapevolezza degli anti-pattern collaborativi che so di dover evitare nei futuri contesti di team.

6.3 Karim ElBerni

Lavorare in gruppo a un progetto così complesso si è dimostrato essere un'esperienza molto più complicata e motivante di quanto pensassi. Pur essendoci stati incidenti di percorso che hanno rallentato inevitabilmente i tempi, siamo comunque riusciti a raggiungere un punto di arrivo, del quale mi sento soddisfatto, soprattutto considerando che è stata la mia prima esperienza di gruppo nell'ambito del game development. Ad oggi, probabilmente cambierei molte cose che ho fatto nel progetto, forse a causa della mia inesperienza iniziale. Ma, in fondo, il punto cruciale è proprio questo: essere arrivati alla fine di un percorso con una conoscenza sulle spalle maggiore, un bagaglio che mi tornerà indubbiamente utile in occasioni future. In sintesi, al di là del risultato finale, il vero successo di questa esperienza risiede nel processo di apprendimento collettivo e individuale. Ogni errore commesso si è

trasformato in una lezione pratica e, guardando indietro, la sfida più grande è stata anche la ricompensa più grande in termini di crescita professionale.

6.4 Luca Dellasantina

Il presente progetto ha rappresentato la mia prima esperienza nello sviluppo di un software complesso in un contesto di team, rivelandosi una sfida estremamente complessa ma ricca di valore formativo. Fin dalle prime fasi, ho acquisito una profonda consapevolezza delle sfide intrinseche legate al lavoro in gruppo, in particolare l'importanza cruciale di una metodologia comunicativa rigorosa per garantire l'integrazione efficace di componenti sviluppati in parallelo. Il superamento delle problematiche riscontrate ha rafforzato la mia comprensione della necessità di sincronizzazione per evitare rallentamenti nelle tempistiche di consegna. Nonostante le imperfezioni residue, il risultato finale è personalmente soddisfacente, soprattutto in relazione all'impegno profuso. In conclusione, il vero successo di questa esperienza non risiede nel prodotto finale, ma nell'acquisizione di un solido patrimonio di competenze e di una maggiore consapevolezza progettuale che ritengo essenziali per il mio futuro professionale nello sviluppo software.

7 Appendice A
Guida Utente

Breve spiegazione dei comandi e delle dinamiche di gioco.

- Il menu principale permette di avviare la partita (cliccando il tasto **PLAY**) , modificare le impostazioni (visive, audio e gestione dei tasti), visualizzare la leaderboard delle migliori run locali o infine uscire dal gioco.
- Una volta avviata la partita si presenterà l'entrata del castello, dove ti dovrai dirigere verso la porta, tramite comandi WASD, per accedere al primo piano.
- Entrato nel primo piano, potrai muoverti attraverso la mappa tramite comandi WASD e inoltre attaccare tramite la *barra spaziatrice*.Cerca di evitare le trappole piazzate per terra e di sconfiggere i nemici attaccandoli.
- Nella mappa sono presenti casse, che possono essere aperte passandoci sopra con il personaggio,contenenti delle ricompense (come Pozioni di Vita,Pozioni di Forza e Monete) e se ci passi sopra ne otterrai l'effetto.
- Per passare al piano successivo ti dovrai dirigere verso la parete laterale bucata.

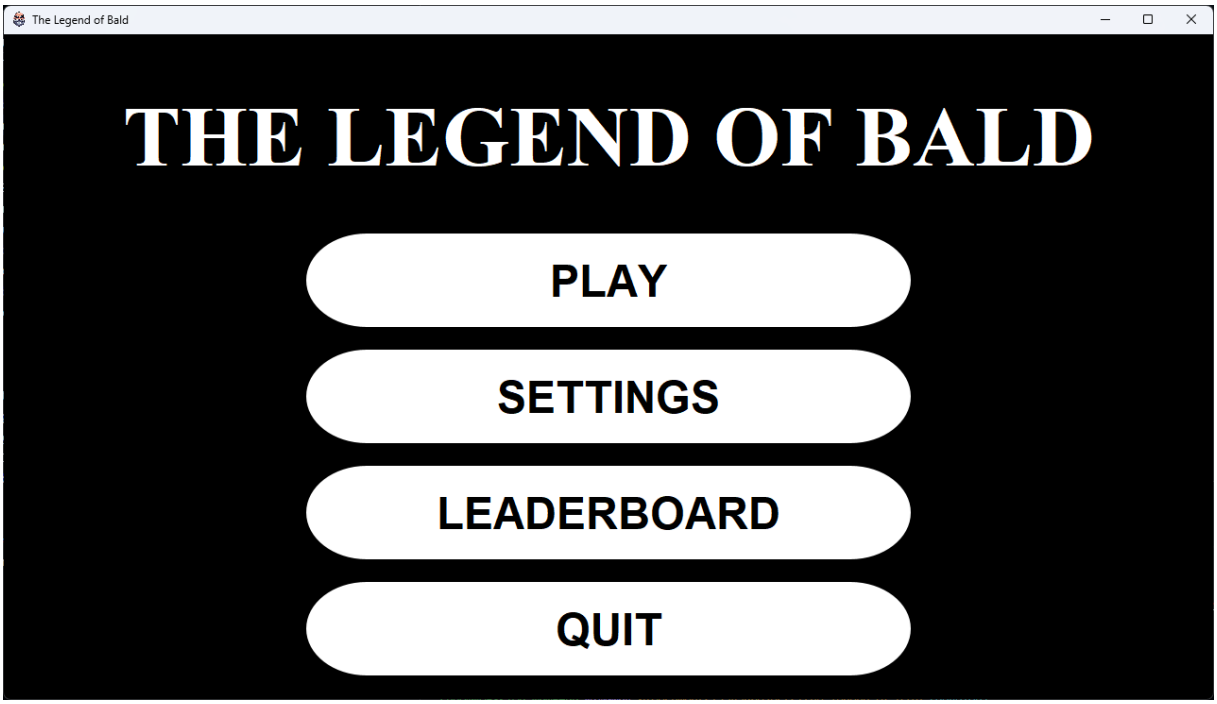

Figura 20: Schermata Principale.

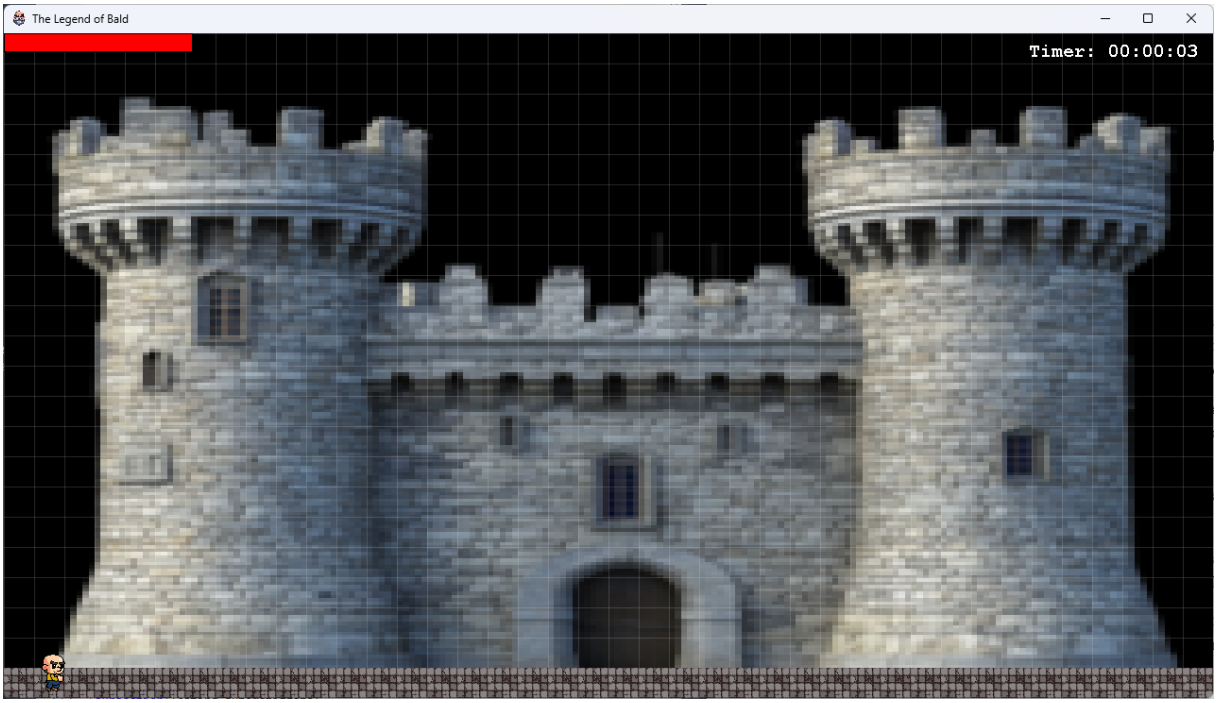

Figura 21: Entrata del castello.

- Per aprire lo shop, devi dirigerti verso l'icona dello shop, dove verrà visualizzato un bottone per la visualizzazione di esso.L'utente in seguito potrai comprare i vari oggetti nel listino in base alle monete collezionate dalle casse.

Figura 22: Primo piano.

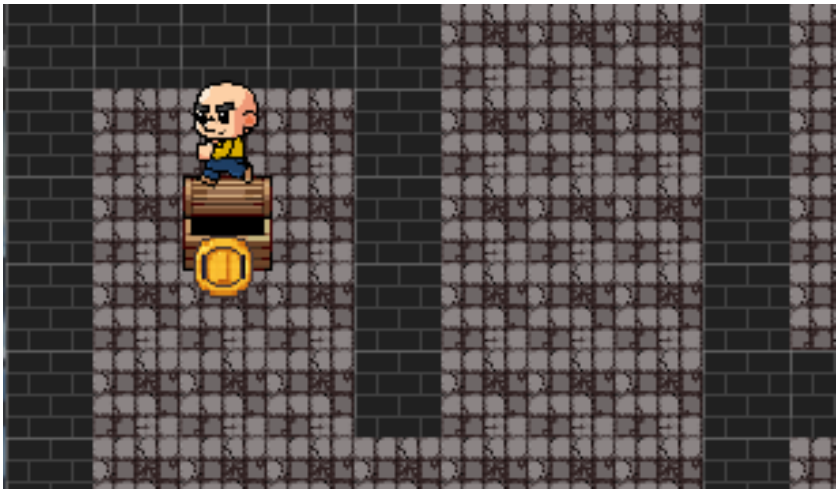

Figura 23: Apertura cassa.

- Per equipaggiare la nuova arma presa nello shop, puoi aprire l'inventario tramite il tasto *I*, dove basterà cliccare sopra la nuova arma per equipaggiarla.
- Se vieni ucciso da trappole o nemici, hai perso!
- Se invece raggiungi l'ultimo piano,sei arrivato alla tua battaglia finale!! Sconfiggi il boss per ottenere un posto nella classifica dei miglior giocatori, oppure se verrai sconfitto verrai riportato al menu principale.Buona fortuna pelato!.

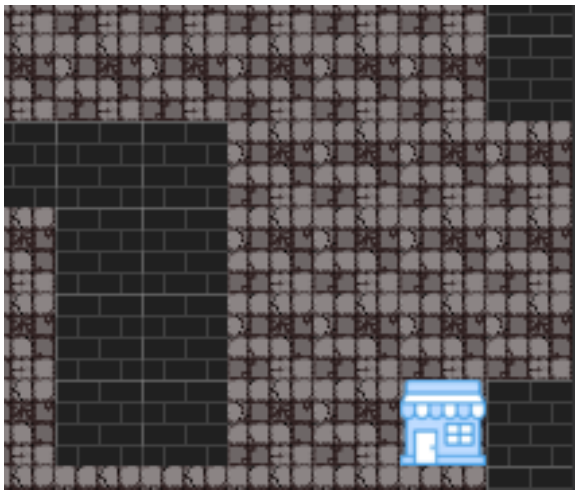

Figura 24: Porta verso il prossimo piano.

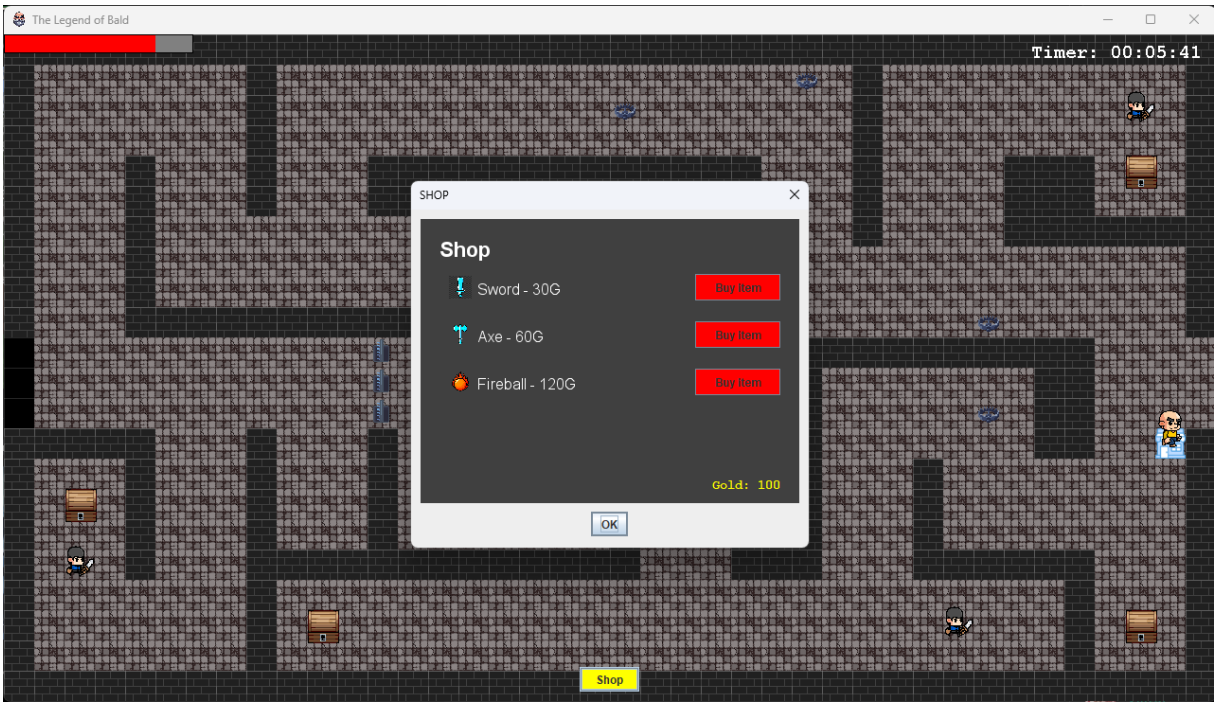

Figura 25: Schermata dello shop.

Riferimenti

- Repository: https://github.com/KarimElBerniUNIBO/the-legend-of-bald
- Documentazione Java SE: https://docs.oracle.com/javase/

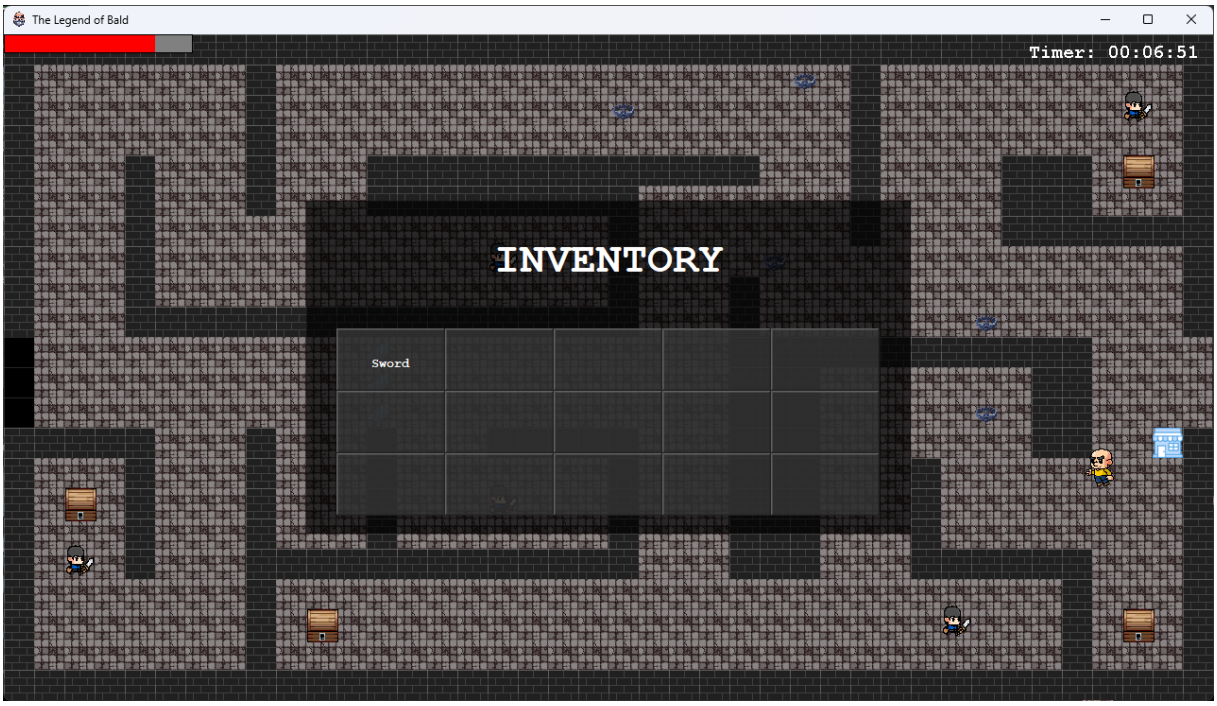

Figura 26: Schermata inventario.

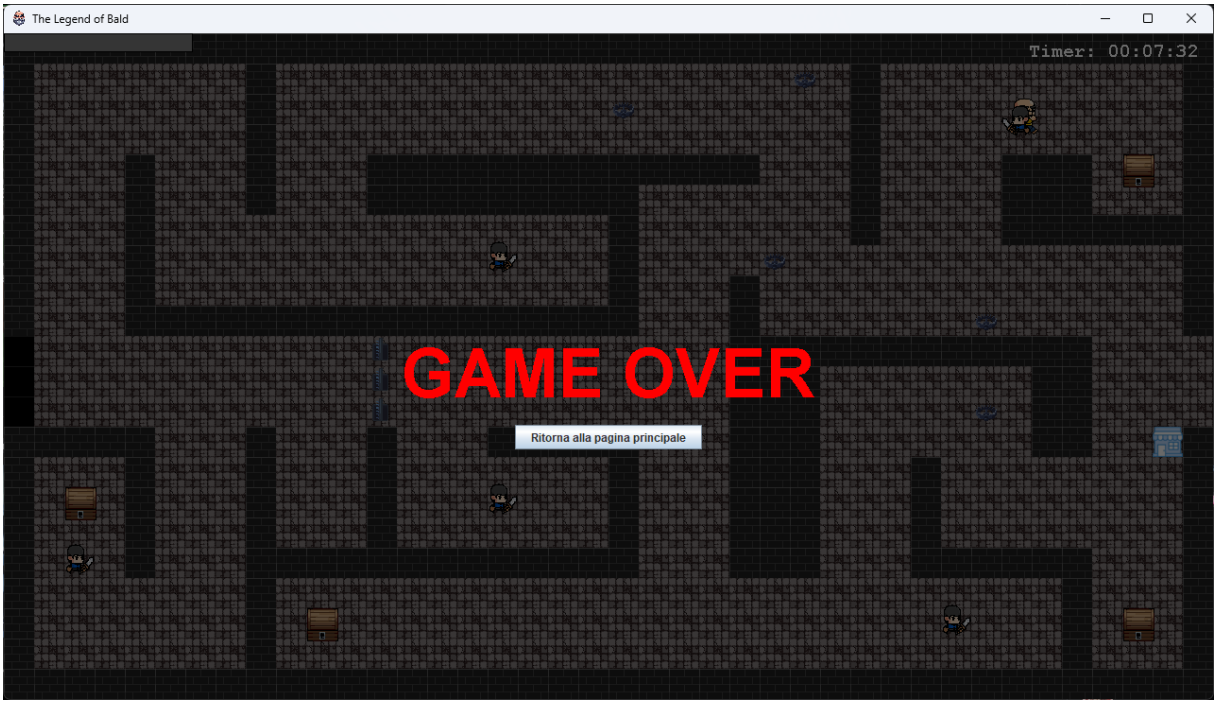

Figura 27: Schermata GameOver.

Figura 28: Ultimo piano con il boss.