Relazione OOP

Gruppo-OOP

2025/26

Contents

1 Analisi

1.1 Descrizione e requisiti

Il software è una riproduzione del videogioco Rogue del 1985, rivisitato in chiave moderna. Il gioco si basa sull' esplorazione di più dungeon a livelli, generati proceduralmente (generati algoritmicamente in modo sempre diverso), formati da corridoi e stanze al cui interno si possono trovare mostri e oggetti preziosi tra cui armature, pergamene, pozioni e armi che il giocatore potrà usare a proprio favore per fronteggiare la crescente difficoltà dei livelli. Il sistema di gioco è a turni per cui ogni azione compiuta dal giocatore determina l'avanzamento dello stato del gioco permettendo ai nemici di eseguire le proprie mosse. L'obiettivo finale è quello di raggiungere l'ultimo livello e ottenere l'amuleto di Yendor. Il punteggio è basato sull'ottenimento di monete, trovate randomicamente dentro le varie stanze e ottenute uccidendo i nemici, tuttavia il punteggio non verrà registrato se non si riescono a terminare tutti i livelli.

Requisiti funzionali

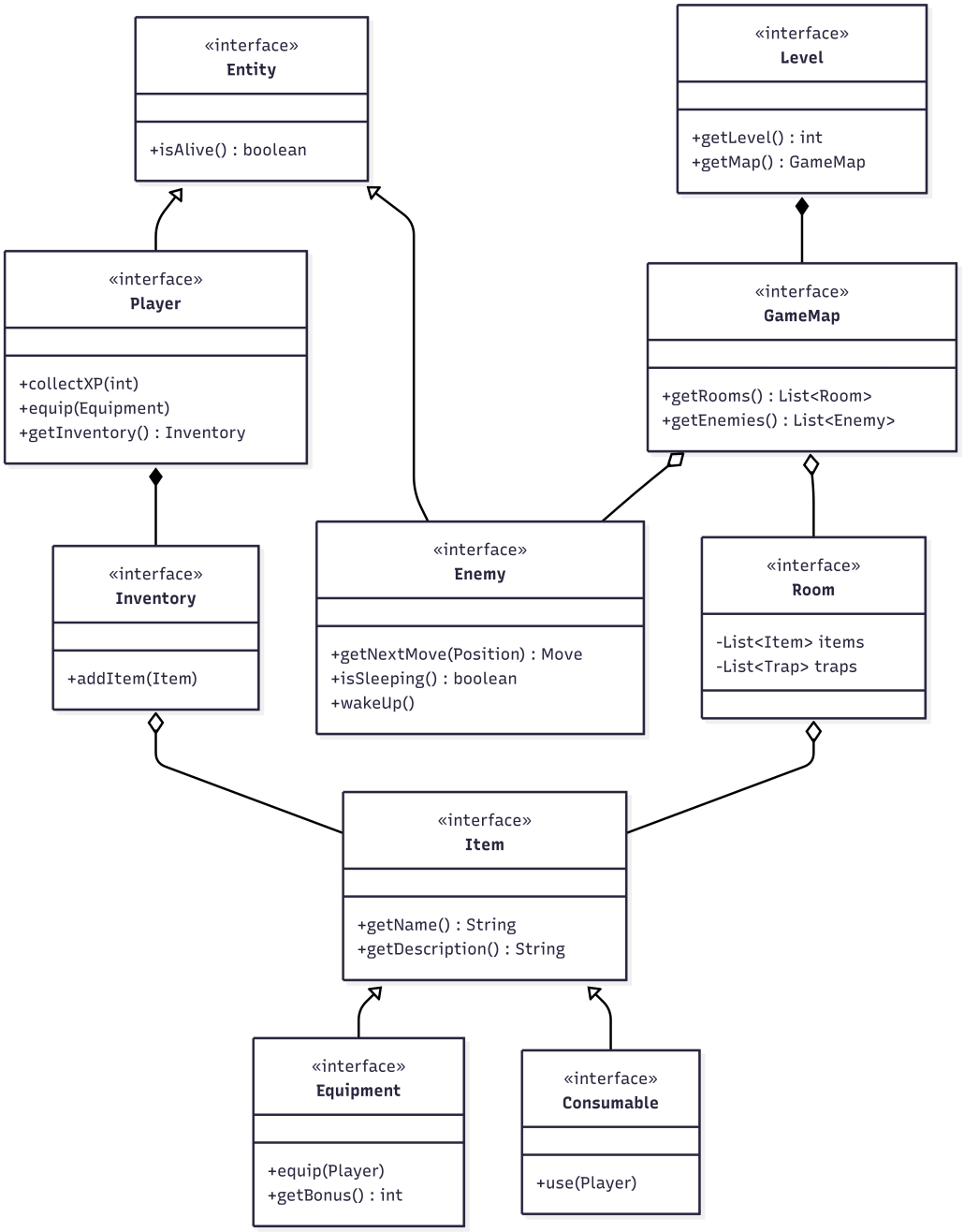
- Il software genera dei livelli, diversi ad ogni partita, composti da stanze connesse da corridoi in cui l'utente si potrá muovere
- Il giocatore deve poter muovere il personaggio sulla mappa tramite comandi da tastiera
- Il dungeon deve contenere diversi tipi di nemici, con comportamenti diversi, che devono poter infliggere danni al giocatore
- Il sistema deve generare randomicamente oggetti raccoglibili e utilizzabili dal giocatore
- Il programma gestisce il sistema di gioco a turni, per cui l'avanzamento dello stato del gioco sará determinato da ogni azione compiuta dal giocatore. In caso di morte la partita ripartira' da zero.
- La partita termina alla morte del personaggio o una volta raccolto l'amuleto di Yendor

Requisiti non funzionali

- Il sistema deve essere facilmente estensibile per quanto riguarda l'aggiunta di nemici e oggetti

1.2 Modello del dominio

Il dominio del gioco è basato su tre macro-parti principali: Entità, Mondo di gioco e Oggetti.
Le entità rappresentano gli elementi attivi del mondo di gioco ossia il giocatore (Player) e i nemici (Enemy). Le entitá possono muoversi all'interno del mondo di gioco, sono dotate di punti vita e possono interagire tra loro.
L'entità del giocatore è l'attore attraverso il quale l'utente interagisce con gli altri elementi del gioco, un giocatore è dotato di un inventario (Inventory) che rappresenta lo spazio in cui vengono conservati gli oggetti trovati.
Il mondo di gioco è modellato come un Dungeon costituito da più livelli. A ogni livello deve essere associata una mappa (Map) che rappresenta l'ambiente in cui nemici e giocatore interagiscono, ciascuna mappa è strutturata come un insieme di stanze (Room) e corridoi (Hallway) che forniscono collegamenti tra le stanze. Le stanze possono contenere trappole (Trap) ed oggetti, i quali vengono generati in modo casuale all'inizio di ciascun livello insieme alla mappa.
Al termine di ogni livello il giocatore troverà una scala che consente l'accesso al livello successivo. Al progredire dei livelli la difficoltà dei nemici aumenta fino al raggiungimento dell obiettivo.
All'interno di ogni stanza è possibile trovare oggetti (Items) posizionati in modo randomico all'interno delle stanze del dungeon, la loro funzionalità principale è quella di interagire con il giocatore influenzando le sue caratteristiche o le dinamiche di combattimento. Essi si dividono in due categorie: Consumabili (Consumable) (Pozioni, cibo, pergamene, frecce) ed indossabili (Equipment) (armature, armi, anelli).La prima categoria dona bonus temporanei, la seconda conferisce bonus permanenti finché indossati. Ciascun item in base alla tipologia ed effetto specifico apporterà delle modifiche alle statistiche del giocatore e/o modificherà alcuni aspetti di combattimento.


«interface»
Entity
+isAlive() : boolean
«interface»
Level
+getLevel() : int
+getMap() : GameMap
«interface»
Player
+collectXP(int)
+equip(Equipment)
+getInventory() : Inventory
«interface»
GameMap
+getRooms() : List<Room>
+getEnemies() : List<Enemy>
«interface»
Inventory
+addItem(Item)
«interface»
Enemy
+getNextMove(Position) : Move
+isSleeping() : boolean
+wakeUp()
«interface»
Room
-List<Item> items
-List<Trap> traps
«interface»
Item
+getName() : String
+getDescription() : String
«interface»
Equipment
+equip(Player)
+getBonus() : int
«interface»
Consumable
+use(Player)

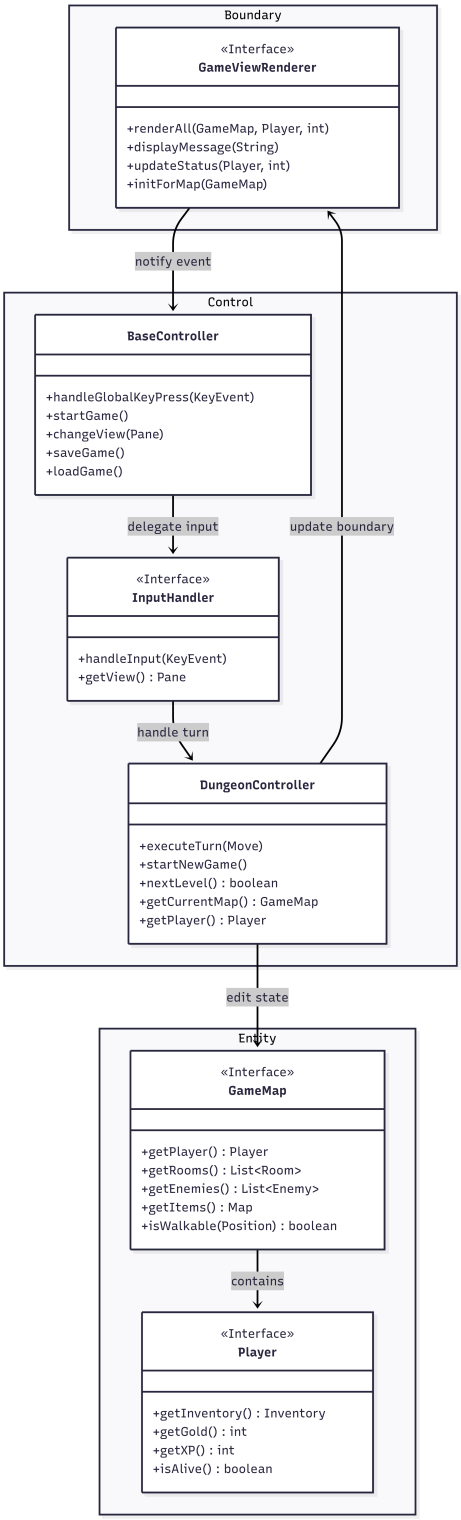
2 Design

2.1 Architettura

Per lo sviluppo di questo gioco abbiamo scelto di adottare il pattern architetturale Entity-Control-Boundary (ECB). Questo ci consente di staccare la logica del dominio (il funzionamento del gioco) dall'interfaccia grafica e dalla gestione dell'input utente. Le entity rappresentano i dati persistenti (quindi Player, Enemy), quest'ultime sono "passive". Non sanno come vengono gestiti i suoi dati, non sanno di essere visualizzati su una GUI. Le classi Entity non hanno dipendenze verso Control o Boundary e per questo sono altamente riutilizzabili. Le classi Control sono il cervello dell'applicazione, non contengono loro stessi i dati di gioco ma sanno come interfacciarsi per ottenerli e come elaborarli. Il sistema a turni del gioco viene gestito dai controller. Le classi Boundary sono invece il punto di contatto tra l'utente e il software. Queste classi agiscono per gestire la logica dell'input e passarla ai Controller e per ricevere dai Controller le informazioni per renderizzare la scena. Le classi Boundary quindi si interfacciano direttamente con le classi Control ma non si interfacciano con le classi Entity per gestire i dati di gioco.
Gestione Evento:

1. L'utente preme un tasto di movimento sulla tastiera
2. La classe Boundary "GameController" cattura l'evento e invocherà il metodo di movimento nel DungeonController
3. La classe DungeonRenderer aggiorna la Map e la renderizza nuovamente
4. Il Controller notifica al Boundary (View) che la Mappa e' cambiata
5. Il Boundary renderizza la scena nuovamente

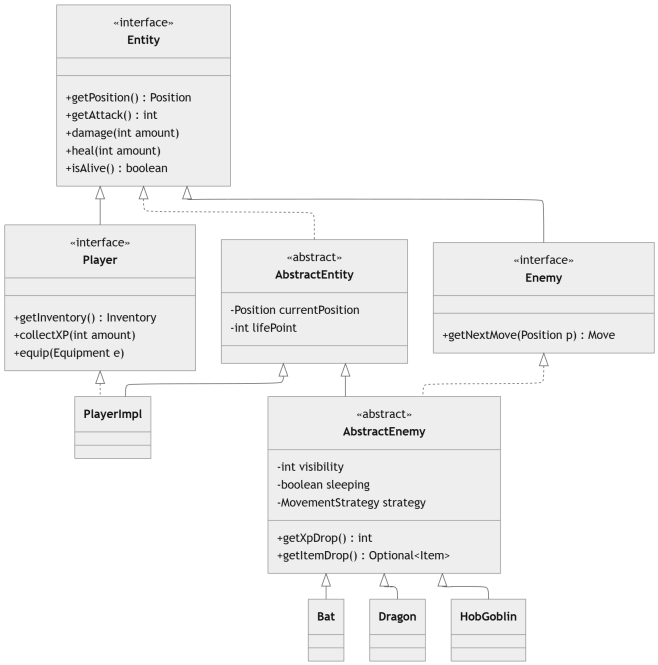
Grazie a questa architettura uno switch tra Swing/JavaFX a un interfaccia testuale richiederebbe solo la riscrittura delle classi Boundary. La modifica dell'algoritmo di generazione della mappa richiederebbe una modifica solo alla classe Control senza modificare le classi Entity e Boundary.


Boundary
«Interface»
GameViewRenderer
+renderAll(GameMap, Player, int)
+displayMessage(String)
+updateStatus(Player, int)
+initForMap(GameMap)
notify event
Control
BaseController
+handleGlobalKeyPress(KeyEvent)
+startGame()
+changeView(Pane)
+saveGame()
+loadGame()
delegate input
update boundary
«Interface»
InputHandler
+handleInput(KeyEvent)
+getView() : Pane
handle turn
DungeonController
+executeTurn(Move)
+startNewGame()
+nextLevel() : boolean
+getCurrentMap() : GameMap
+getPlayer() : Player
edit state
Entity
«Interface»
GameMap
+getPlayer() : Player
+getRooms() : List<Room>
+getEnemies() : List<Enemy>
+getItems() : Map
+isWalkable(Position) : boolean
contains
«Interface»
Player
+getInventory() : Inventory
+getGold() : int
+getXP() : int
+isAlive() : boolean

2.2 Design avanzato

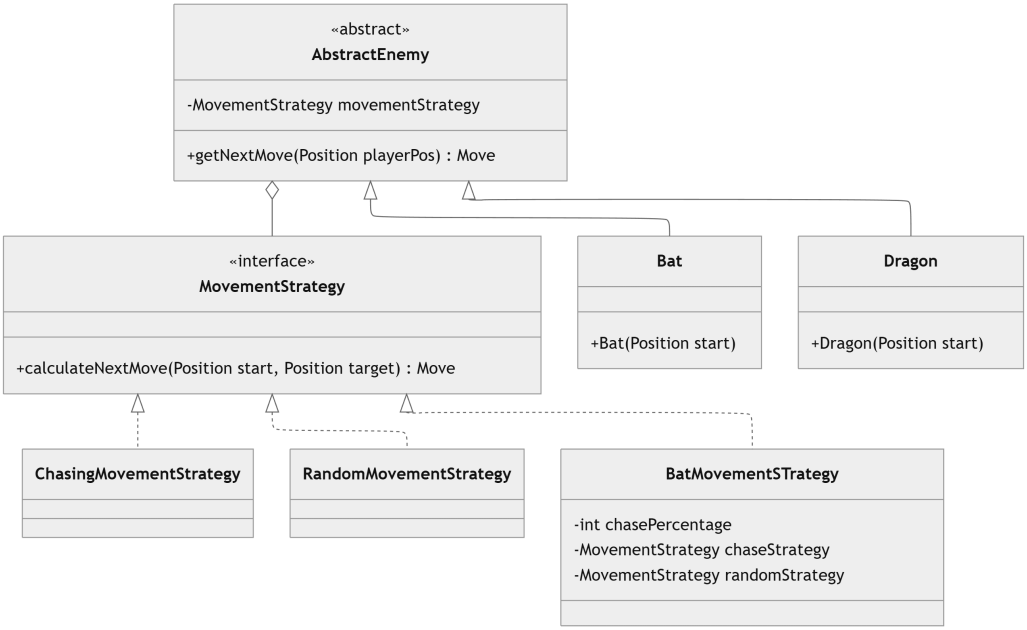
2.2.1 Simone Salucci

Astrazione e gerarchia delle entitá



Problema: Strutturazione delle entitá, tutte le entita condividono proprietá comuni (hp, armor class, ecc...) ma differiscono nelle interazioni con l'utente, il giocatore viene ad esempio controllato dall'utente mentre i nemici vengono controllati da un IA in base alla posizione del giocatore.
Soluzione: É stata adottata una struttura a livelli basata sull'ereditarietá che permette di massimizzare il riuso del codice rispettando il principio DRY "Don't repeat yourself". Abbiamo un interfaccia principale Entity che definisce il contratto minimo per ogni entitá e una classe astratta AbstractEntity che implementa Entity definendo tutta la logica comune tra le varie entitá; Le interfacce Player ed Enemy estendono Entity specializzando il contratto con metodi specifici permettendo ai controller di eseguire operazioni generiche su entitá senza specificare se si tratta di player o enemy. Inoltre per gestire le differenze di comportamento e statistica tra i nemici è stato introdotto un secondo livello di astrazione tramite la classe AbstractEnemy che specializza ulteriormente il comportamento dei nemici permettendoci di aggiungere diversi nemici con statistiche differenti mantenendo un comportamento di base indifferito tra i vari tipi di mostri, questo ci permette di rispettare l'Open/Closed principle.

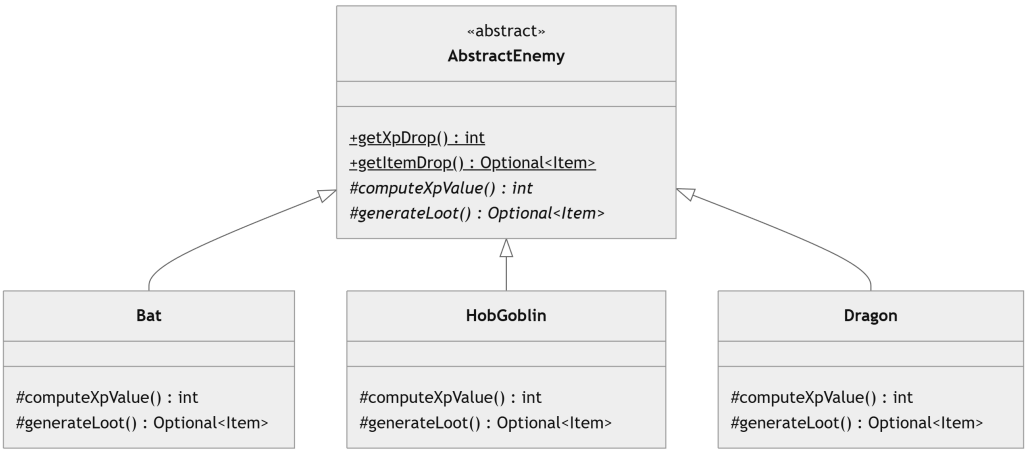
Gestione dei comportamenti dinamici



Problema: Implementazione dei comportamenti dinamici dei nemici, ogni tipo di nemico ha un comportamento differente rispetto ad un altro, ad esempio un drago si sposterá in modo piú aggressivo rispetto ad un pipistrello andando dritto verso il giocatore.

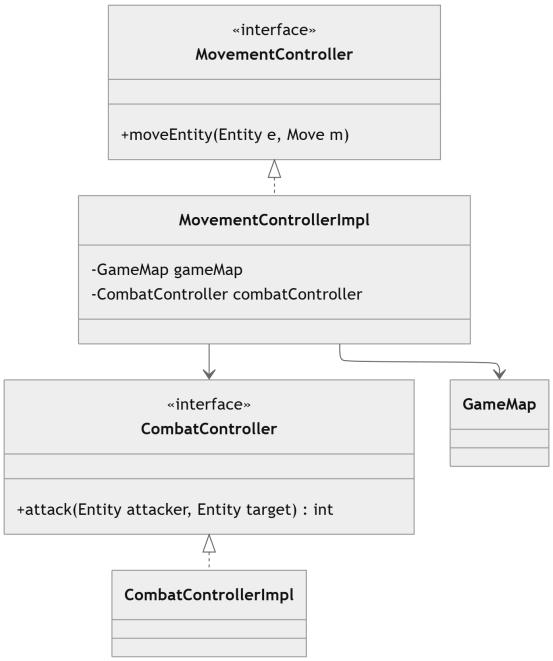
Soluzione: É stata separatá la logica di calcolo del movimento dell'entitá dall'entitá stessa attraverso il pattern Strategy implementato tramite l'interfaccia MovementStrategy. Invece di definire come il mostro deve muoversi, la classe AbstractEnemy possiede un riferimento all'interfaccia MovementStrategy su cui viene richiamato il metodo della strategy che gli ritorna la prossima mossa da eseguire, in questo modo è possibile aggiungere nuovi comportamenti senza modificare niente in AbstractEnemy o nei Controller, stiamo rispettando quindi l'Open/Closed Principle poiché il sistema é aperto alle estensioni ma chiuso alle modifiche. In questo modo la responsabilitá di decidere la mossa è assegnata alla strategie che estendono MovementStrategy che si occupano solo di quello, mentre il nemico si occupa solo di gestire i suoi stati, quindi possiamo dire che stiamo rispettando il principio "SRP"(Single Responsibility Principle).

Sistema di ricompense e generazione loot



Problema: Gestione dell'uccisione di un nemico che deve innescare il rilascio di punti esperienza (xp) e di oggetti che insieme vanno a costruire il "loot" di quel nemico, il quale deve essere diverso per ogni tipo di nemico.
Soluzione: É stato applicato il pattern template method all'interno della classe AbstractEnemy. La classe base definisce lo scheletro dell'algoritmo di rilascio del loot tramite getXpDrop() e getItemDrop() che sono dichiarati final e sono i Template Method, questi metodi eseguono dei controlli e poi invocano metodi primitivi computeXpValue() e generateLoot() dichiarati astratti che le classi concrete (Bat, HobGoblin, Dragon) sono obbligate ad implementare. Questo ci permette di personalizzare il valore del drop a seconda del nemico sconfitto rispettando il principio "DRY".

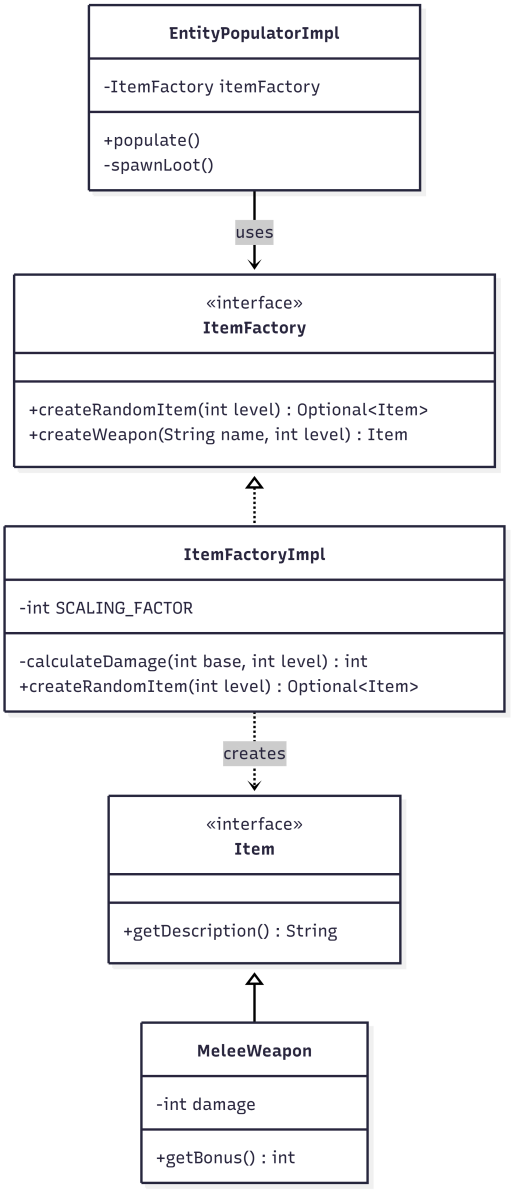
Logica di movimento e combattimento



Problema: Gestione della logica di movimento delle entitá nella mappa e gestione del combattimento tra due entitá se l'entitá che attacca si é spostata contro l'entita che viene attaccata
Soluzione: Le entitá sono diventate oggetti di dati intelligenti che gestiscono solamente il proprio stato interno senza prendere decisioni sul mondo esterno, a questo scopo sono stati implementati dei Controller come MovementControllerImpl e CombatControllerImpl che agiscono da arbitri e hanno la funzione di regolare il cambiamento di stato di una data entitá. In questo modo i parametri delle entitá sono separati dalla logica di business del gioco permettendoci di eseguire eventuali modifiche sulla logica di business senza intaccare le entitá.

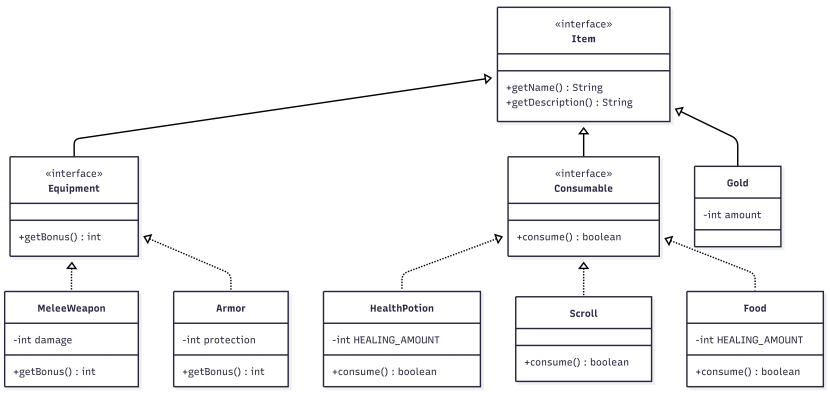
2.2.2 Cristian Costrut

Generazione Procedurale e Bilanciamento degli Oggetti



Problema: Durante la progettazione, è emersa la necessità di popolare il Dungeon con una vasta varietà di oggetti (armi,armature,consumabili). L'approccio iniziale, basato sull'istanziazione diretta delle classi concrete

all'interno alla logica di generazione della mappa, presentava due problemi: violava il principio di Separazione della Responsabilità, unendo la struttura del livello al contenuto degli item, e rendeva complessa la gestione del bilanciamento della difficoltà, cioè la generazione di oggetti sempre più potenti con l'avanzare dei livelli.
Soluzione: Per risolvere questo problema è stato utilizzato il Factory Method Pattern , introducendo l'interfaccia ItemFactory e la sua implementazione ItemFactoryImpl. Questa architettura permette all'EntityPopulator di delegare la logica della creazione degli item interamente alla factory, che andrà a creare items bilanciati in base al livello corrente. All'interno di ItemFactoryImpl l'algoritmo di generazione sfrutta dei metodi dedicati (come calculateProtection e calculateDamage) basati su formule matematiche deterministiche. Per esempio il danno di un'arma viene calcolato sommando il baseDamage dell'arma (che varia se è un pugnale, una spada o una pala), a un fattore di crescita growth ottenuto dividendo il livello corrente per una costante SCALING FACTOR e per ultimo applicando una varianza casuale (da -1 a 1). Infine, per garantire la riproducibilità dei livelli essenziale per i testing e il debugging, la factory utilizza una classe wrapper statica GameRandom e non la classica Random, che permette di impostare un seed globale all'avvio del gioco,assicurando che la generazione procedurale sia identica a parità di seed.



Polimorfismo e Gerarchia degli oggetti

Problema: Il gioco richiede la gestione di oggetti molto differenti tra loro: una pozione che cura, una pala per combattere o una pergamena per scoprire i segreti di un anello hanno comportamenti, statistiche e modalità di utilizzo molto differenti. Gestire queste differenze tramite un'unica grande classe concreta attraverso lunghe catene di controlli avrebbe violato i principi della programmazione a oggetti, rendendo il sistema rigido e difficile da mantere.

Soluzione: È stata definita una rigorosa gerarchia di interfacce per sfruttare il polimorfismo e rispettare l'Open Closed Principle.

- **Item (interfaccia base)**: definisce il contratto comune tra tutti gli oggetti (getName() e getDescription()).
- **Equipment(sotto-interfaccia)**: Estende Item e rappresenta gli oggetti indossabili(armi e armature).Aggiunge metodi specifici come getBonus() per calcolare l'apporto alle statistiche del giocatore.
- **Consumable(sotto-interfaccia)**: Estende Item e rappresenta gli oggetti di singolo utilizzo(pozioni,cibo,pergamene). Aggiunge metodi specifici come heal() che conferiscono al player un effetto istantaneo e vengono rimossi dopo l'uso.
- **Resources(sotto-interfaccia)**: Estende Item e rappresenta valute o materiali. Momentaneamente risulta implementata solo dalla classe Gold, ma si è deciso di mantenere questa interfaccia per garantire l'estensibilità futura senza dover fare particolari modifiche al sistema.

L'InventoryManager interagisce principalmente con l'intefaccia Item, utilizzando il controllo dei tipi (tramite instanceof) in modo mirato solo nel momento dell'utilizzo effettivo.

Rendering Condizionale e Feedback Visuale
Problema: La visualizzazione dell'inventario pone una sfida di design legata alla sovrapposizione degli stati. Ogni slot della griglia deve comunicare simultaneamente diverse informazioni all'utente: se la cella contiene un oggetto, se quell'oggetto è attualmente equipaggiato dal giocatore e se è la cella correttamente selezionata dal cursore. Questa complessità rispondeva a uno dei requisiti non funzionali prima del progetto: migliorare la User Experience rispetto l'interfaccia originale di Rogue, evitando il più possibile la confusione visiva che era causata in precedenza.
Soluzione: All'interno della classe InventoryGUI è stato implementato un algoritmo di rendering basato su priorità visuali che viene eseguito ad ogni aggiornamento della vista. Come evidenziato nel diagramma UML, la GUI non mantiene uno stato interno della selezione, ma interroga InventoryManager ad ogni ciclo di aggiornamento, garantendo la sincronizzazione perfetta con il modello.

- **Priorità alta (selezione):** Se la cella è selezionata, il bordo diventa rosso per indicare la posizione del cursore.

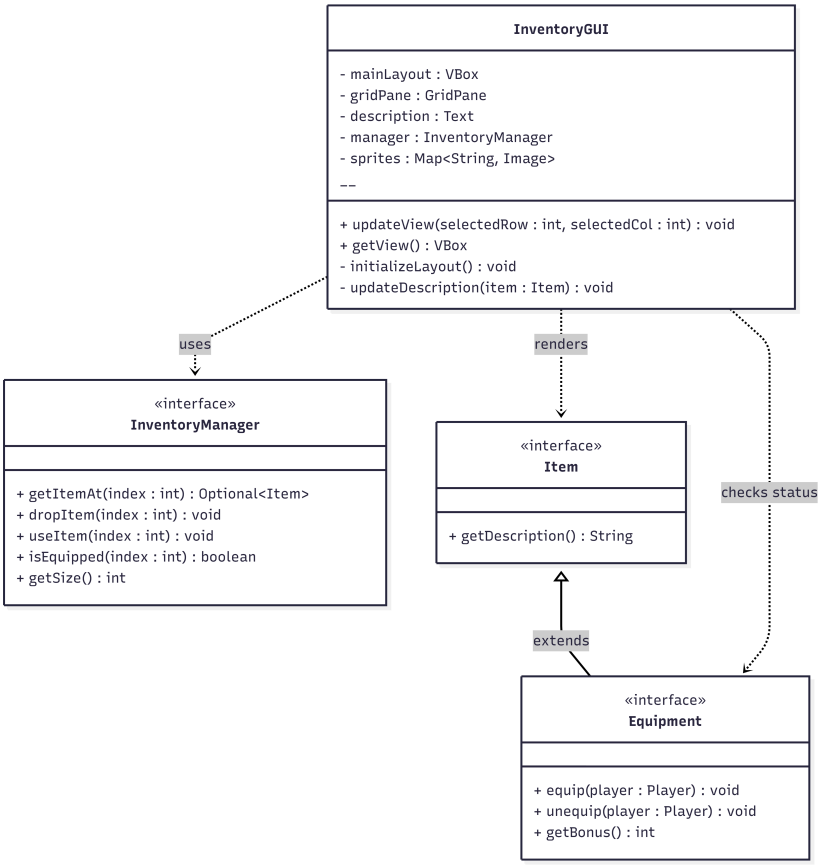
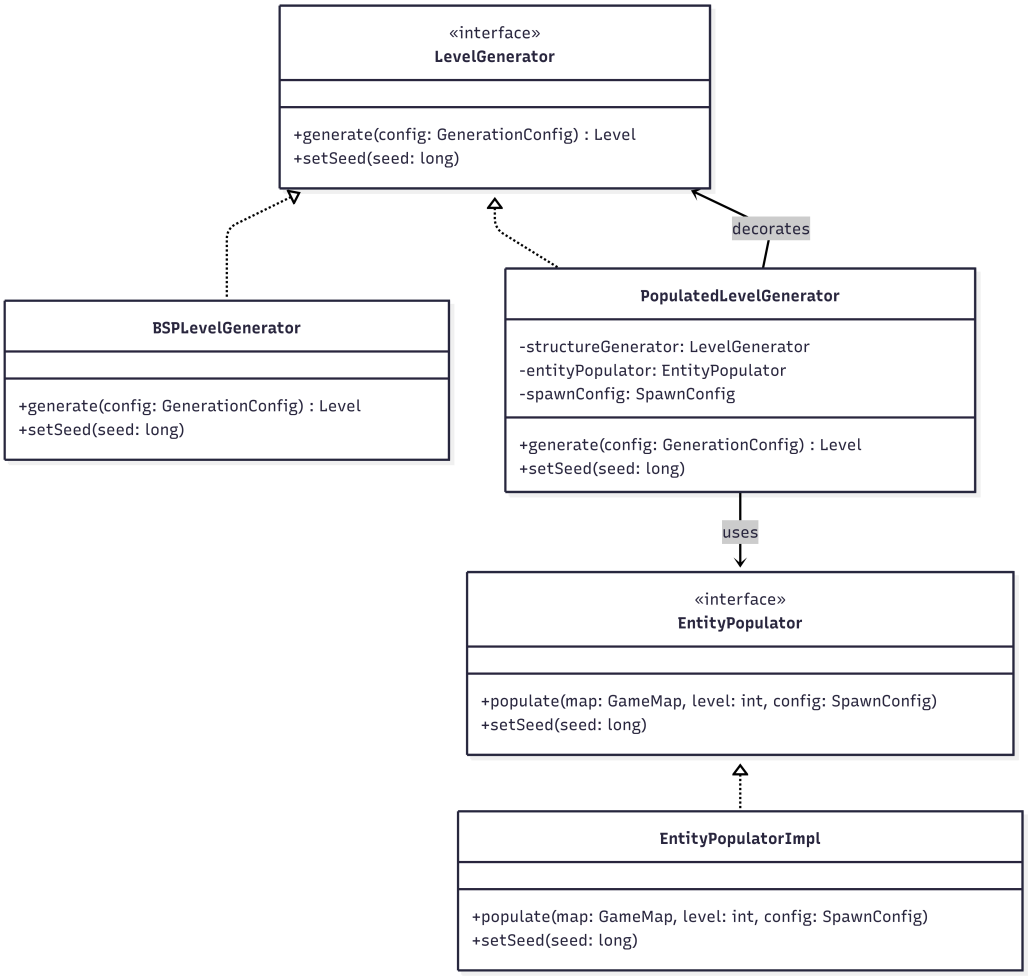


Figure 1: Schema UML del Dominio

- **Priorità media (equipaggiamento):** se l'oggetto è equipaggiato, il bordo è blu. Tuttavia, poiché la selezione ha priorità grafica sull'equipaggiamento, è stata introdotta un'icona Badge "E" in alto a destra. Questa appare sempre sugli oggetti in uso, garantendo che l'informazione sia visibile anche quando l'oggetto è selezionato.
- **Priorità bassa (Default):** altrimenti il bordo è neutro.

2.2.3 Romeo Biancalani

Generazione procedurale dei livelli



Problema: Il gioco richiede la generazione procedurale dei livelli del dungeon. Un livello è composto da piu Stanze che sono collegate tra loro da Corridoi. Nelle stanze è possibile trovare Nemici e Items che sono generati randomicamente. La generazione del livello e lo spawn di Nemici e Item sono divise per permettere di poter variare l'algoritmo di layout senza cambiare le regole di spawn e viceversa.
Soluzione: L'interfaccia LevelGenerator definisce il metodo generate che deve implementare qualsiasi algoritmo di generazione. In aggiunta c'è anche un metodo setSeed che deve essere usato per garantire la riproducibilità della generazione (vedi Save e Load). BSPLevelGenerator implementa Lev-

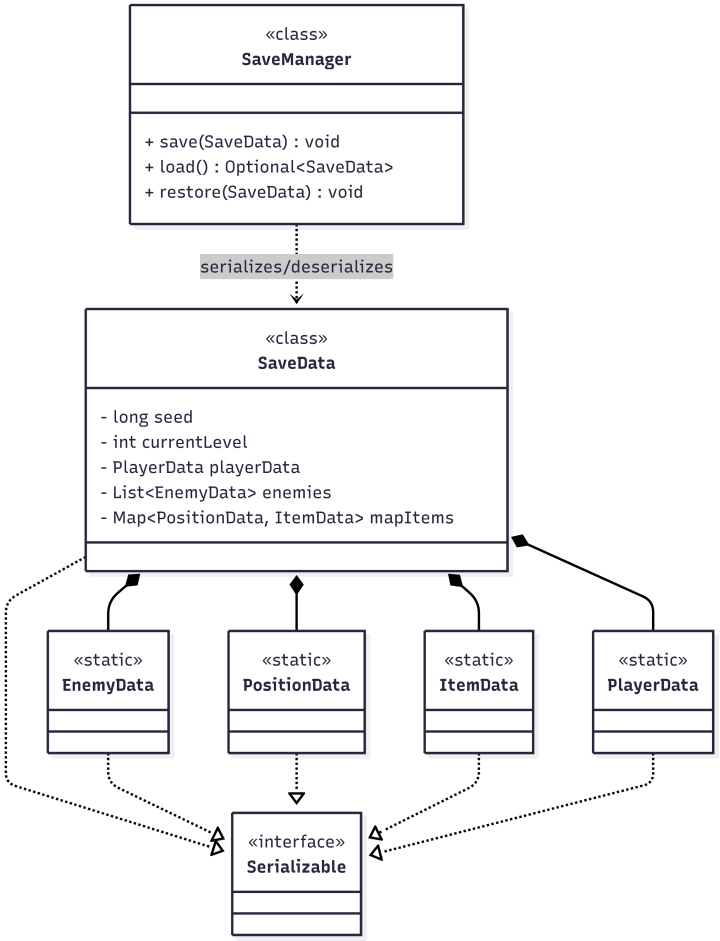
elGenerator tramite un algoritmo di Binary Space Partitioning (`https://en.wikipedia.org/wiki/Binary_space_partitioning`), questo algoritmo viene usato da altri videogiochi famosi come Doom e genera un livello dividendo ricorsivamente lo spazio disponibile in partizioni. Una volta generato un albero vengono generate le stanze nelle foglie dell'albero e vengono collegate tra loro con dei corridoi.
La classe PopulatedLevelGenerator agisce da Decorator, wrappa una qualsiasi istanza di LevelGenerator e aggiunge il popolamento delle entita tramite EntityPopulator che utilizza le classi Factory per generare Item e Mostri.
Questa architettura garantisce un'ampia flessibilità permettendo di cambiare l'algoritmo di generazione senza impattare il popolamento (e di testare entrambi indipendentemente).
Pattern utilizzati:

- Decorator = PopulatedLevelGenerator (Wrappa LevelGenerator aggiungendo EntityPopulator)
- Strategy = LevelGenerator (Consente piu algoritmi tramite una classe comune di comportamenti)

Per rendere la generazione piu chiara ed efficiente tutti i parametri sono stati spostati in due Record: GenerationConfig e SpawnConfig che gestiscono rispettivamente i parametri di generazione del livello e i parametri di spawn di oggetti e entita. Entrambi forniscono dei factory methods:

- GenerationConfig.withDefaults(width, height, level, seed)
- SpawnConfig.defaults()

Persistenza dello stato di gioco

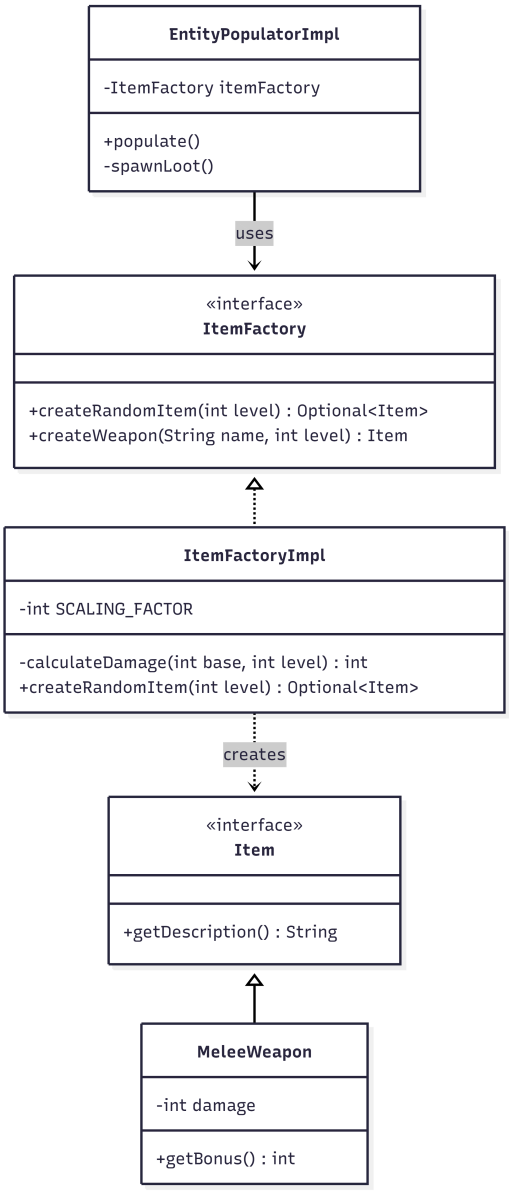


Problema Il gioco da la possibilita di salvare e ripristinare lo stato di una partita in corso. Il sistema permette il ripristino anche dopo il riavvio dell'applicazione salvando il file nella path "user.home/.jrogue/save.dat".
Soluzione: Il sistema si basa su due classi:

- SaveData: contiene alcune classi statiche (PlayerData, ItemData...) che sono responsabili del salvataggio di un tipo di dato. Tutte implementano Serializable.
- SaveManager: questa classe implementa e gestisce le operazioni di I/O. Offre metodi save(), load() e restore() che si occupa di ricostruire lo stato del gioco come era prima del salvataggio. Adottando una generazione deterministica tramite il seed sara' necessario salvare il seed per poter generare piu volte la stessa GameMap.

Per ridurre la dimensione del file di salvataggio infatti, non viene salvata tutta la griglia di Tile ma viene salvato il seed + le entita + l'inventario.

Rendering a livelli con canvas stratification



Problema: Per disegnare la mappa dobbiamo gestire il rendering di piu elementi: tile, oggetti, nemici, nebbia che si possono anche sovrapporre (un nemico si trova sopra una tile pavimento). Ridisegnare ogni frame tutti gli elementi sarebbe inefficiente, la mappa infatti non cambia durante un livello.
Soluzione: Viene implementato DungeonRenderer che estende la classe StackPane di JavaFX e si occupa di gestire tre Canvas sovrapposti:

- Canvas del Terreno: viene disegnato solo al caricamento del livello e non viene piu aggiornato.
- Item Canvas: Viene ridisegnato solo quando l'utente raccoglie un oggetto.
- Entity Canvas: Viene ridisegnato ad ogni turno, in questo Canvas troviamo il player che si muove e anche i nemici.

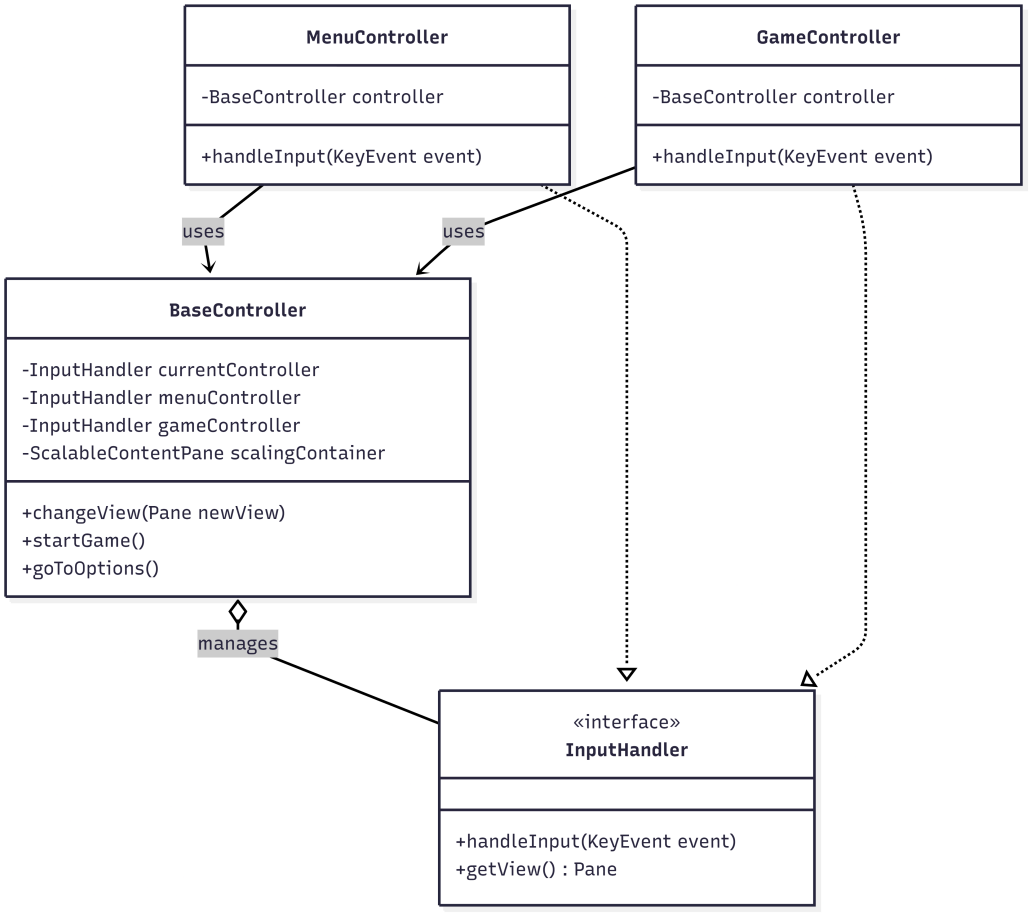
Gli sprite sono caricati una volta dalla funzione loadSprite e sono poi memorizzate in una Map<String, Image> evitando di ricaricarlo piu volte e permettendo la condivisione con altre classi Boundary (come la classe InventoryGUI che necessita delle immagini per renderizzare l'inventario).

2.2.4 Thomas Bosi

Problema: Navigazione coerente nel software
L'applicazione necessita di un sistema coerente per gestire la transizione tra le diverse fasi del gioco (Menu principale, Opzioni, Partita attiva, Game Over). L'obiettivo è garantire una separazione netta tra la logica di controllo e le interfacce grafiche, permettendo al contempo facilita nell'estensione, ad esempio per l'aggiunta futura di nuovi menu o stati di gioco.

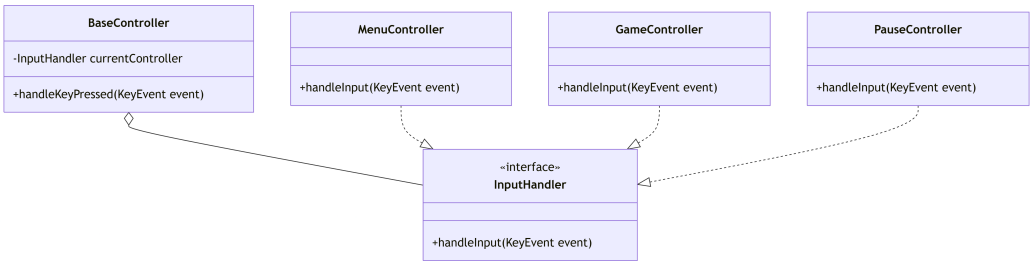
Soluzione:

Per risolvere questa problematica è stato implementato un sistema di controllo gerarchico basato su un Controller centrale. Il BaseController funge da gestore principale: detiene i riferimenti a tutti i sotto controller (MenuController, GameController, OptionsController, etc...) e gestisce lo scambio delle view. A loro volta ogni singolo controller sottostante a BaseController viene implementato con l'interfaccia InputHandler, questo impone i metodi handleInput e getView, fondamentali per cio che riguarda il cambiamento di boundary e di funzionalita specifiche di essa. Il BaseController riesce a delegare al controller di riferimento la gestione dell'associazione dei KeyEvent ad una funzionalita. Questa scelta implementativa ha come vantaggio l'estensbilita, ogni "schermata" ha la propria logica con controller e boundary dedicati.


MenuController
-BaseController controller
+handleInput(KeyEvent event)
GameController
-BaseController controller
+handleInput(KeyEvent event)
uses
uses
BaseController
-InputHandler currentController
-InputHandler menuController
-InputHandler gameController
-ScalableContentPane scalingContainer
+changeView(Pane newView)
+startGame()
+goToOptions()
manages
«interface»
InputHandler
+handleInput(KeyEvent event)
+getView() : Pane

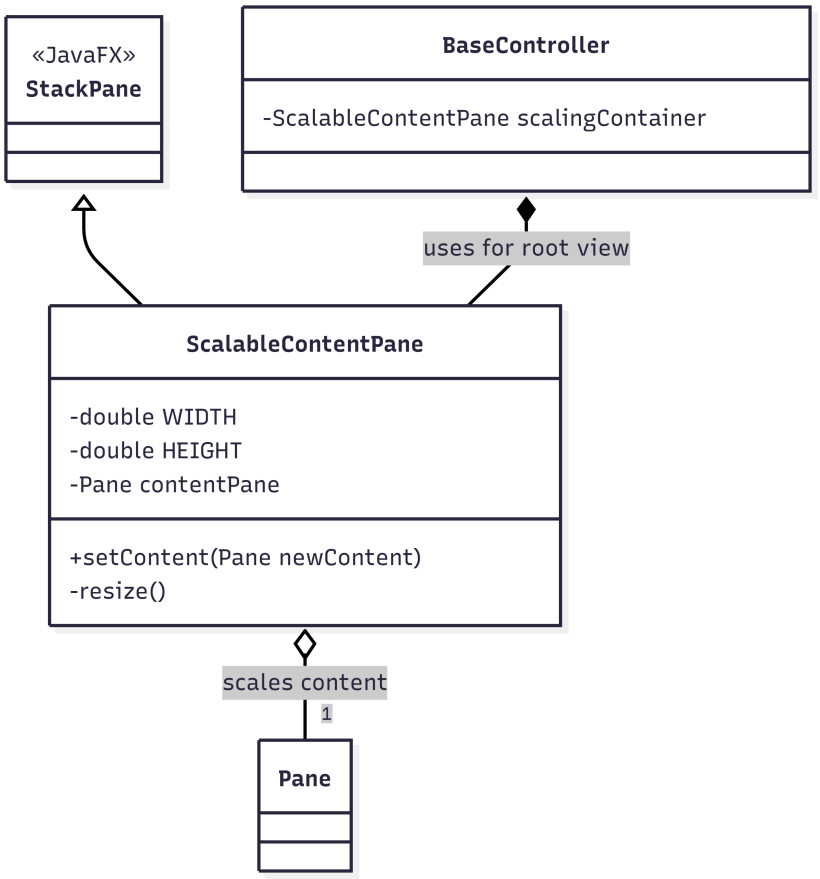
Problema: Gestione degli input da tastiera
L'input da tastiera dell'utente deve svolgere funzioni specifiche in base al contesto in cui si trova, ad esempio nel Menu il tasto "W" deve permettere di navigare verso l'alto mentre quando si e nel gioco deve far muovere il giocatore.

Soluzione:
E stata definita una interfaccia InputHandler che e stata poi implementata in tutti i controller specifici collegati al BaseController, il quale si occupa di ricevere l'input da tastiera KeyEvent e passarlo al controller specifico in modo tale che ogni controller possa mappare i propri tasti e funzioni specifiche.



Problema: Implementazione della scalabilita della risoluzione e aspect ratio
L'applicazione deve essere visualizzabile su schermi di dimensioni diverse e supportare un ridimensionamento della grafica mantenendo l'aspect ratio senza distorcere le proporzioni.Il problema principale si presenta nella situazione in cui la finestra o il monitor utilizzato abbia un aspect ratio diverso dal 16:9. Il sistema deve fare si che le GUI non "sappiano" se sono su un altro aspect ratio permettendo al team di lavorare tenendo come riferimento il formato 16:9 a 1920x1080 pixel.

Soluzione:
E stata implementata la classe ScalableContentPane applicando la tecnica del Letter Boxing per mantenere l'aspect ratio 16:9. L'idea e: "ascoltare" i cambiamenti della dimensione della finestra e ricalcola dinamicamente le dimensioni del contenuto, riempiendo di nero lo spazio tra contenuto e finestra.La classe estende StackPane e agisce come un contenitore radice. Internamente, definendo una risoluzione virtuale fissa (1920x1080) e tramite dei listener sulle proprietà di larghezza e altezza della finestra, viene calcolato un fattore di scala (scaleFactor) basato sulla dimensione minima tra i due assi. Viene applicata una classe Scale al contenuto. Se la finestra ha un rapporto diverso da 16:9, il ScalableContentPane centra il contenuto e riempie lo spazio rimanente con bordi neri, mantenendo l'integrità visiva del gioco.


«JavaFX»
StackPane
BaseController
-ScalableContentPane scalingContainer
uses for root view
ScalableContentPane
-double WIDTH
-double HEIGHT
-Pane contentPane
+setContent(Pane newContent)
-resize()
scales content
1
Pane

3 Sviluppo

3.1 Testing Automatizzato

Il progetto adotta una strategia di testing basata su framework JUnit5 per garantire la stabilità del codice a fronte di modifiche e per garantire la correttezza delle logiche di gioco. Nello specifico i test sono stati suddivisi nelle seguenti macro aree:
Entitá: Oltre alle statistiche e alla logica di base di ogni entitá (player, bat, hobgoblin e dragon), vengono testate le interazioni tra entitá come il calcolo dei danni, la gestione della visibilitá dei nemici e le strategie di movimento.
Inventario e Oggetti:I test riguardano principalmente il rispetto della capienza massima assicurando che l'inserimento in un inventario pieno venga rifiutato correttamente senza errori a runtime. Viene inoltre garantita la consistenza delle operazioni di aggiunta e rimozione e il corretto recupero di oggetti di diverse tipologie (Consumabili e Equipaggiabili)
Generazione procedurale: E' stato creato un file di test BSPGenerationTest che consente di testare se la generazione avvenga con successo.
Logica dei menu e input: Sono state testate le logiche di movimento all'interno dei menu, simulando un KeyEvent per spostarsi all'interno del menu. La maggior parte dei test sono tuttavia stati effettuati manualmente.

3.2 Note di sviluppo

3.2.1 Romeo Biancalani

Qui sotto ho scritto le feature avanzate di Java utilizzate durante lo sviluppo:
Record: Utilizzati per GenerationConfig `https://github.com/RomeoBiancalani/OOP25-j-rogue/blob/b85c3ad0bd6efddf70d636bfa74091e50872c631/src/main/java/it/unibo/jrogue/controller/generation/api/GenerationConfig.java#L16`
Pattern Matching: Ho utilizzato instanceOf per DungeonRenderer nelle funzioni getItemSprite e getEnemySprite. E' stato utilizzato anche in SaveManager.createItemData() per la serializzazione degli Items. `https://github.com/RomeoBiancalani/OOP25-j-rogue/blob/b85c3ad0bd6efddf70d636bfa74091e50872c631/src/main/java/it/unibo/jrogue/boundary/DungeonRenderer.java#L437-L456` `https://github.com/RomeoBiancalani/OOP25-j-rogue/blob/b85c3ad0bd6efddf70d636bfa74091e50872c631/src/main/java/it/unibo/jrogue/engine/SaveManager.java#L269-L288`
Optional: Ho utilizzato Optional in BSPNode per ottenere i figli opzionali

dell'albero BSP (infatti un nodo che e' foglia non ha figli). E' stato utilizzato anche in GameMap (come in getEntityAt(Position p) che restituisce un Entity se trovato alla posizione, altrimenti Optional.empty()) https://github.com/RomeoBiancalani/OOP25-j-rogue/blob/b85c3ad0bd6efddf70d636bfa74091e50872c631/src/main/java/it/unibo/jrogue/controller/generation/api/BSPNode.java#L40

Stream API e Lambda Expressions: Sono state utilizzate in SimpleGameMap per filtrare velocemente entita (getEnemies, getEntityAt). Sono state utilizzate le operazioni di Stream filter, map, anyMatch. https://github.com/RomeoBiancalani/OOP25-j-rogue/blob/b85c3ad0bd6efddf70d636bfa74091e50872c631/src/main/java/it/unibo/jrogue/entity/world/impl/SimpleGameMap.java#L81

Collections.unmodifiableXXX (map/list/set) in SimpleGameMap per creare copie immutabili e evitare modifiche dall'esterno. https://github.com/RomeoBiancalani/OOP25-j-rogue/blob/b85c3ad0bd6efddf70d636bfa74091e50872c631/src/main/java/it/unibo/jrogue/entity/world/impl/SimpleGameMap.java#L126

Path API: La classe SaveManager utilizza Path e Files.createDirectories(), Files.newOutputStream() e Files.newInputStream() sostituendo le api java.io.File consentendo una gestione piu semplice e robusta. https://github.com/RomeoBiancalani/OOP25-j-rogue/blob/b85c3ad0bd6efddf70d636bfa74091e50872c631/src/main/java/it/unibo/jrogue/engine/SaveManager.java#L91

Parti di codice da fonti esterne: Per la generazione del dungeon ho trovato questa repo Github (https://github.com/marukrap/RoguelikeDevResources?tab=readme-ov-file#procedural-map-generation) che fornisce alcune risorse per creare un gioco RogueLike da 0. In particolare ho letto https://www.roguebasin.com/index.php?title=Basic_BSP_Dungeon_generation, https://eskerda.com/bsp-dungeon-generation/, https://varav.in/archive/dungeon/, https://en.wikipedia.org/wiki/Binary_space_partitioning per capire meglio come utilizzare BSP per la generazione del livello. Tutti gli sprite usati nel progetto sono stati estratti dal pacchetto Dawnlike che fornisce numerose risorse. Il pacchetto e' scaricabile da https://opengameart.org/content/dawnlike-16x16-universal-rogue-like-tileset-v181 o https://github.com/hadean-mirrors/dawnlike.

3.2.2 Simone Salucci

In questa sezione vengono illustrate le feature avanzate di java utilizzate durante lo sviluppo:

Utilizzo di lambda expression e stream:

- https://github.com/RomeoBiancalani/OOP25-j-rogue/blob/b85c3ad0bd6efddf70d636bfa74091e50872c631/src/main/java/it/unibo/jrogue/controller/MovementControllerImpl.java#L221-L224
- https://github.com/RomeoBiancalani/OOP25-j-rogue/blob/b85c3ad0bd6efddf70d636bfa74091e50872c631/src/main/java/it/unibo/jrogue/entity/entities/impl/AbstractEnemy.java#L107
- https://github.com/RomeoBiancalani/OOP25-j-rogue/blob/b85c3ad0bd6efddf70d636bfa74091e50872c631/src/main/java/it/unibo/jrogue/controller/MovementControllerImpl.java#L143-L160

Utilizzo di Optional:

- https://github.com/RomeoBiancalani/OOP25-j-rogue/blob/b85c3ad0bd6efddf70d636bfa74091e50872c631/src/main/java/it/unibo/jrogue/controller/MovementControllerImpl.java#L219-L226

3.2.3 Thomas Bosi

In questa sezione vengono illustrate le feature avanzate di java utilizzate durante lo sviluppo di seguito un esempio per tipo:
Utilizzo di Lambda expression:

https://github.com/RomeoBiancalani/OOP25-j-rogue/blob/b85c3ad0bd6efddf70d636bfa74091e50872c631/src/main/java/it/unibo/jrogue/engine/ScalableContentPane.java#L27
Utilizzo di Optional:

https://github.com/RomeoBiancalani/OOP25-j-rogue/blob/b85c3ad0bd6efddf70d636bfa74091e50872c631/src/main/java/it/unibo/jrogue/entity/items/impl/Scroll.java#L22
Sono stati impiegati strumenti di supporto per compiere degli "How to" come per l'inserimento di Background preso dal sito geekforgeek:

https://www.geeksforgeeks.org/java/javafx-background-class/

3.2.4 Cristian Costrut

In questa sezione vengono illustrate le feature avanzate di java utilizzate durante lo sviluppo:
Uso di Stream e lambda-expressions:
https://github.com/RomeoBiancalani/OOP25-j-rogue/blob/b85c3ad0bd6efddf70d636bfa74091e50872c631/src/main/java/it/unibo/jrogue/entity/items/impl/SimpleInventory.java#L63

Uso di Optional in diverse parti ma queste sono solo alcuni esempi:
https://github.com/RomeoBiancalani/OOP25-j-rogue/blob/b85c3ad0bd6efddf70d636bfa74091e50872c631/src/main/java/it/unibo/jrogue/entity/items/impl/ItemFactoryImpl.java#L109

https://github.com/RomeoBiancalani/OOP25-j-rogue/blob/b85c3ad0bd6efddf70d636bfa74091e50872c631/src/main/java/it/unibo/jrogue/controller/InventoryManagerImpl.java#L85

https://github.com/RomeoBiancalani/OOP25-j-rogue/blob/b85c3ad0bd6efddf70d636bfa74091e50872c631/src/main/java/it/unibo/jrogue/entity/items/impl/Armor.java#L69

4 Commenti finali

4.1 Autovalutazione

4.1.1 Autovalutazione (Romeo Biancalani)

Credo di aver completato il lavoro che mi era assegnato con successo. Abbiamo concentrato buona parte del lavoro negli ultimi giorni sono riuscito a finire la mia parte e anche le funzionalita opzionali

4.1.2 Autovalutazione (Thomas Bosi)

Per essere la mia prima esperienza di lavoro di gruppo, con un progetto strutturato e di dimensioni discrete sono relativamente soddisfatto del mio operato. Alcune sezioni di codice sono un po troppo hardcoded che per mancanza o per cattiva gestione del tempo mi sono ritrovato a non poter ottimizzare e rendere più pulite. Alcune feature a cui tenevo come l'implementazione del supporto a 21:9 nativo non sono riuscito ad implementarlo, dovendo quindi limitarmi all utilizzo di Letter Boxing,pratica che avrei preferito evitare.Sento anche di non aver dato abbastanza importanza ai test e che sarebbero potuti tornare utili soprattutto nelle ultime fasi di sviluppo. Altra mia mancanza anche il non aver ralvato le references per il punto 3.2 note di implementazione per la relazione durante la stesura del codice. Tutto sommato si e rivelata essere una piacevole esperienza costruttiva su cosa non fare piu in futuro e cosa invece portarmi dietro, oltre che formativa nell'ambito della progettazione.

4.1.3 Autovalutazione (Cristian Costrut)

Sono abbastanza soddisfatto di quello che ho fatto io personalmente e penso che nel complesso abbiamo lavorato bene come gruppo, anche se forse non siamo riusciti a organizzarci perfettamente in quanto ci siamo ritrovati negli ultimi giorni con diverse cose da implementare e sistemare. Comunque siamo riusciti a implementare tutto quello che ci eravamo imposti all'inizio quindi mi ritengo contento del nostro progetto.

4.1.4 Autovalutazione (Simone Saluccii)

Sono soddisfatto del mio lavoro poiché ritengo di aver completato le parti assegnatemi raggiungendo gli obiettivi prefissati in fase di progettazione. Ci sono state problematiche riguardanti l'organizzazione del progetto dato che

la maggior parte del lavoro è stato fatto nell'ultima settimana ma siamo comunque riusciti a lavorare in team, suddividendo i compiti e riuscendo a portare a termine il lavoro. Sono particolarmente soddisfatto del lavoro finale e principalmente dell'implementazione dei movimenti dei nemici tramite il pattern strategy.

5 Appendice A

5.1 Guida Utente

Il gioco è controllato interamente tramite tastiera, di seguito sono elencati i comandi principali:

- **Movimento:** tasti **W,A,S,D** per muovere il personaggio nelle quattro direzioni.
- **Attacco:** per attaccare un nemico, è sufficiente tentare di muoversi nella casella da esso occupata.
- **Raccolta:** il personaggio raccogli automaticamente gli oggetti semplicemente passandoci sopra.
- **Scale:** posizionandosi sopra alla casella con le scale e premendo il tasto **E**, si passa al livello successivo.
- **Menù :** premere **ESC** per accedere al menu di pausa o **INVIO** per confermare le selezione nei menu.
- **Inventario:** per accedere all'inventario bisogna premere il tasto **Q**. All'interno di questa modalità i comandi cambiano:
  - **Navigazione:** usare **W,A,S,D** per spostare il cursore di selezione (indicato da un bordo rosso sulla cella)
  - **Usa:** premere **E** sull'oggetto selezionato, l'oggetto viene consumato e l'effetto applicato e rimosso dall'inventario.
  - **Equipaggia:** premere sempre **E** sull'oggetto selezionato, l'oggetto viene equipaggiato e un bordo blu con anche un badge "E" indicheranno che l'oggetto è attivo.
  - **Scarta:** Premere **R** per rimuovere permanentemente l'oggetto dall'inventario.