Dragon & Dungeon

Alex Marchiorello , Carlo Maggiolo, Matteo Padrin, Dante Geretto

1 Introduzione

Il gioco Dragon & Dungeon è un gioco single-player sviluppato in Java cui obiettivo è muoversi nella mappa per arrivare alle stanze, sconfiggere i mostri nelle stanze, potenziarsi. Una volta sconfitti tutti i mostri sarà possibile combattere il Drago al fine di uscire dalla caverna salvi.

2 Specifiche tecniche

- Versione Java: 21
- Versione GSON: 2.11
- Versione JUnit: 4

3 Grafica

Per l'implementazione grafica è stata utilizzata la libreria `Java swing`. Per la creazione del frame si è fatto uso di uno strumento drag and drop. Come segnalato nel `Javadoc` alcune parti di codice sono quindi black box.
L'interfaccia di gioco, a partita avviata, si presenta come segue:

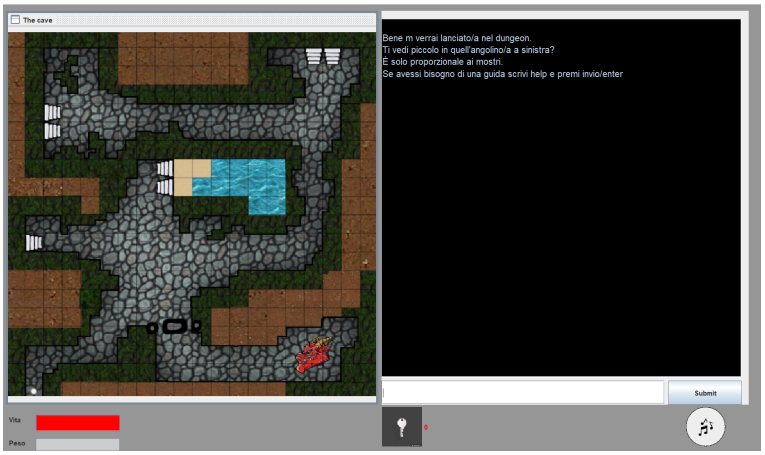

Figura 1: L'interfaccia di gioco

Per digitare i comandi bisogna scriverli al premendo su *enter your answer*. Nella parte in basso a sinistra sono riportate tre barre, una che mostra la vita del nemico, mentre le altre sono per il player e indicano il peso totale degli oggetti trasportati e la vita.
Nella parte in basso a destra, in ordine, il pulsante che permette di riprodurre la colonna sonora e un pulsante per salvare il gioco.

4 Caricamento mappa

Per caricare la mappa è possibile creare un documento `.txt` o previo un file `xml`

Sintassi del file txt

Ogni riga all'inizio deve contenere due valori interi separati da virgola che rappresentano la posizione e.g. `5,2`.
Superati tutti i nodi della mappa deve essere presente una riga `"Connessioni"` (senza le virgolette)
Dopo la riga `"Connessioni"` scrivere i collegamenti tra i nodi secondo questo formato

$$x_A, y_A > x_B, y_B : D_A, D_B$$

Esempio `1,19>8,13:N,S` che significa che il nodo (1,19) si collega procedendo verso `Nord` al noto (8,13) arrivando verso la sua parte `Sud`.

Nella fase di implementazione della mappa è stata utilizzata la libreria esterna `JGraph` Link alla pagina [i]

5 Entity

Questa classe rappresenta un entità nel gioco ovvero rappresenta il generico comportamento di player e nemici, in particolare gestisce:

- cambio della vita (changeHealth)
- getter del attacco, difesa, vita, storia, nome
- metodi per incrementare in modo sicuro armor, health e damage

  NB: sfrutta la classe "StorageManagement" per la gestione di tutti gli inventari. StorageManagement permette la generazione di "item" tramite l'utilizzo di "ItemFactory".

6 Monster

Questa classe estende **Entity**, sfrutta **l'observer pattern** per cambiare il livello:

- i mostri sono identificati da un tipo
- il livello viene aggiornato automaticamente(si adatta a quello del player)
- la creazione di un mostro va fatta usando la Factory

NB: l'implementazione di ciascun mostro è fatta in classi separate poiché ciascuno può avere dei valori diversi (es. history)

[i] https://jgrapht.org/

7 Player

Questa classe estende **Entity**, sfrutta **l'observer pattern** per informare i mostri che il player ha cambiato il suo livello:

- nel caso di aumento del livello informa tutti gli observer
- gestisce gli item nel inventario (il player ha un peso massimo quindi gli oggetti che può portare sono limitati)
- per default il player può portare al massimo un peso di 10

8 GameEngine

Questa classe gestisce la logica nel gioco, in realtà sfrutta il **facade pattern** per gestire in modo semplice player, nemici e mappa:

- attacco del player
- attacco del mostro
- informazioni sulle direzioni disponibili di player
- informazioni sugli item contenuti nelle chest delle stanze
- gestisce armi e difese (spada, armatura...)
- gestisce pozioni, (aumento attacco a player, danno istantaneo...)

9 Prompt e Menù

Classe intermediaria tra la grafica ed il classe che si occupa di mettere in comunicazione il gameEnegine con la grafica ed il salvataggio attraverso l'uso degli adapter. Lo scopo principale del prompt è quindi quello di rendere indipendenti diverse parti del progetto.
Lo scopo della classe è eseguire le azioni previste dalle altre classi una volta ricevuto un input tramite riga di comando.
Per favorire la lettura di un così vasto sistema si è deciso di applicare un command pattern per ogni comando utilizzabile, che quando viene riconosciuto si occuperà del necessario, oltre che al controllo se il numero di valori passati è corretto (alcune classi hanno accesso agli adapter per un funzionamento più agevole).
Il caricamento di questi comandi viene fatto dalla classe Menù, che si occupa, quando istanziati, di caricare i comandi dai file xml, metterli dentro ad una HashMap (più comoda per un accesso diretto) e collegarli direttamente ad un oggetto che implementa l'interfaccia Command.
NB: per aggiungere un nuovo comando, è necessario inserirlo sia nel file `xml`, sia inizializzarlo nella classe Menù. Nel caso che manchi uno dei due verrà lanciata una eccezione. Non può accadere quindi che esista l'alias di un comando nel file `xml` senza che sia presente il relativo oggetto istanziato.

10 Classi di utilità

Sono state realizzate una serie di classi per facilitare l'implementazione:

- `ParsePath` gestisce i percorsi per recuperare file di configurazione ed immagini
- xmlReader gestisce la lettura di file xml per configurare item, nemici, mappa
- altro(strutture come RoomValues, MonsterValues...) per memorizzare le informazioni di configurazioni lette dai file xml

11 Salvataggio

Per poter salvare lo stato di gioco e caricare i parametri fondamentali su Amazon AWS (viene usato un bucket S3) vengono usate le seguenti classi:

- `awsClient` si occupa di gestire la connessione con il bucket tramite le API di AWS (AWS SDK 2.0)
- `JsonParser` usa la libreria GSON (2.11) per serializzare gli oggetti (Player e MapGraph) in un file json che verrà usato come file di salvataggio
- `SaveAdapter` segue le specifiche di pattern **Adapter** per mascherare le due classi precedenti e permettere di accedere facilmente ai metodi di salvataggio

Link jira: https://studenti-team-dqs4ulpl.atlassian.net/jira/software/projects/SCRUM/boards/1/backlog