1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

Relazione per il Progetto del Corso di Programmazione e Modellazione ad Oggetti

Elia Renzoni, Annarosa Clemente, Eloi Ricci, Giuseppe Benedetti

25/05/2024

Indice

1 Analisi

1.1 Analisi dei Requisiti

Labyrinth Legends è un gioco a singolo utente multi-livello in cui il giocatore deve uscire dal labirinto in cui è intrappolato. La mappa in cui si muove è delimitata da mura ed è cosparsa di ostacoli (quali bombe, NPC e bucce di banana) e premi (come le casse contenenti monete e l'incantesimo di immunità).
Il giocatore deve raccogliere più monete possibili in un tempo ragionevole; al termine della partita questi due valori concorreranno alla creazione di una classifica nella quale saranno visibili i record ottenuti dalle varie partite.
Il giocatore potrà personalizzare il suo game-play scegliendo un nome utente, un livello di difficoltà tra EASY, NORMAL e HARD e il personaggio con cui giocare; i personaggi sono due, un ragazzo e una ragazza: Linda e Fonzie.
All'aumentare del livello di difficoltà aumenta anche il numero di ostacoli.

1.2 Lista dei Requisiti:

Requisiti funzionali:

- Implementazione degli ostacoli:
  - Bomba: se colpita fa perdere delle monete al giocatore. Il numero di monete perse dipende dal livello di difficoltà della partita.
  - NPC: se toccati uccidono il personaggio, terminando così la partita.
  - Banana: se presa fa retrocedere il personaggio di cinque posizioni nel caso di Fonzie, altrimenti di sette posizioni nel caso di Linda.
- Implementazione dei premi:
  - Incantesimo di immunità: se preso rende immune il personaggio. L'effetto avrà durata non oltre i successivi tre spostamenti effettuati dal giocatore.
  - Cassa: contiene delle monete il cui numero dipende dal livello di difficoltà della partita. Se colpita il giocatore riscuote le monete contenute.
- Implementazione delle mura: bloccano il passaggio del giocatore.
- Implementazione dei personaggi:
  - Linda: la più veloce, ogni movimento la fa spostare di due posizioni nella mappa.
  - Fonzie: ha una vita extra, muore la seconda volta che colpisce un NPC.
- Il programma deve eseguire i movimenti dell'utente.
- Il programma deve permette all'utente di scegliere un nome, un personaggio e una modalità di gioco.
- A fine partita deve essere visualizzata una classifica delle dieci migliori partite, ordinata in base al minor tempo impiegato per uscire dal labirinto e al maggior numero di monete ottenute.
- Il programma deve creare dinamicamente e casualmente la mappa.

- La partita va gestita applicando le regole descritte qui sopra.

Requisiti non funzionali:

- Gestione dei suoni.
- Ottimizzazione delle texture fatta tramite l'ausilio della bufferizzazione delle immagini.
- Gestione e salvataggio dei dati di gioco su un file.

1.3 Modello del Dominio

Labyrinth Legends deve organizzare il gioco in base al livello di difficoltà selezionato, questo è rappresentato dall'entità **GameDifficult**, la quale determina il numero e tipo di oggetti - ovvero premi e ostacoli - che popoleranno la mappa. Gli oggetti sono rappresentati dall'entità **Element**.
I principali attori coinvolti in una partita sono:

- L'utente, rappresentato dall'entità **IUser**, la quale permette di impostare un nome utente.
- Il personaggio, rappresentato dall'entità **Player** la quale contiene le caratteristiche principali dei personaggi, come il nome (Linda o Fonzie), la velocità e la vita.
- La mappa, rappresentata dall'entità **IGameMap**, che deve essere generata in base al livello di difficoltà scelto.
- Il risultato finale rappresentato dall'entità **IResult**. Questa crea un record in cui il nome del giocatore è associato al personaggio scelto, alle monete raccolte e al tempo impiegato.
- La classifica, rappresentata dall'entità **IRanking**, la quale aggiunge i record dei giocatori in un file e li ordina in base al tempo impiegato e alle monete ottenute.
- Il movimento del personaggio, gestito dall'entità **Movement**, la quale gli consente di muoversi attraverso la mappa, lo blocca quando incontra delle mura, modifica la sua posizione se inciampa su una banana e controlla per quanto tempo l'incantesimo immunità avrà ancora effetto.

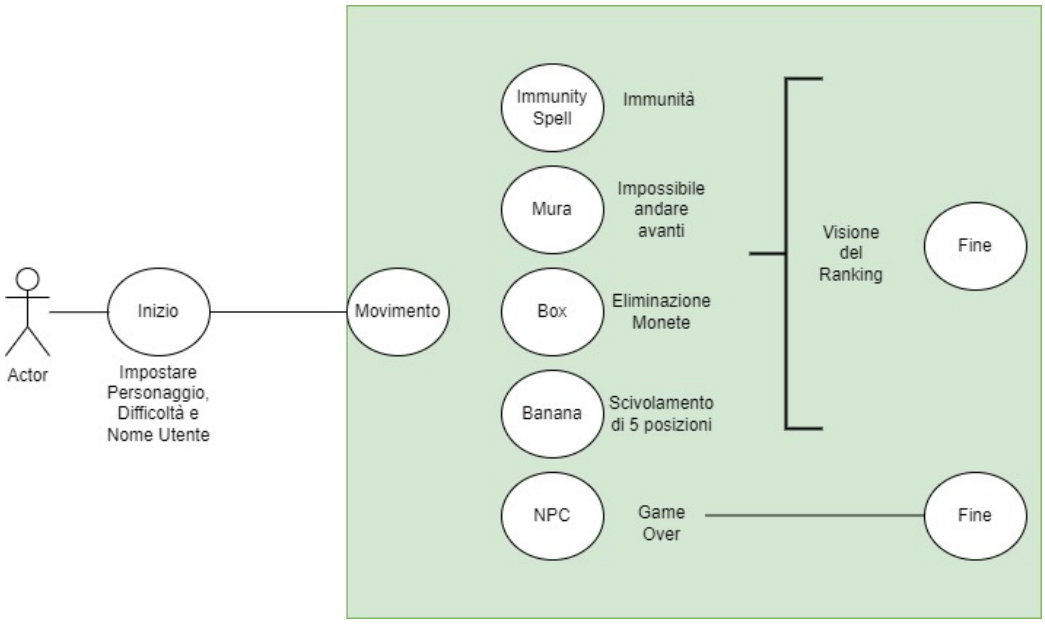


Figura 1: Diagramma d'uso

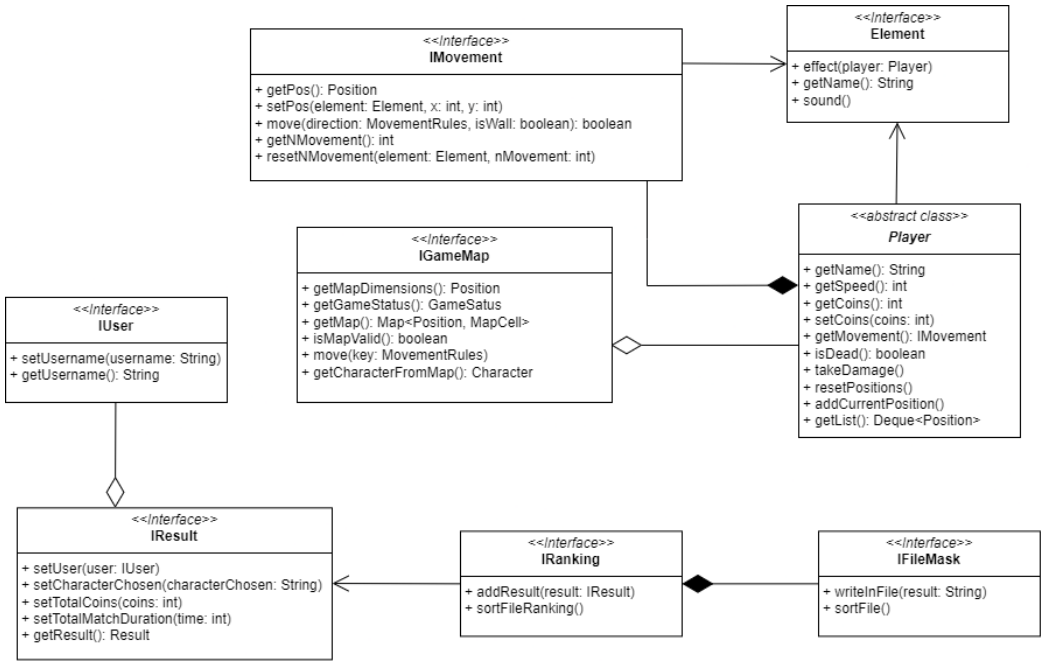


Figura 2: Modello del dominio

La **figura 1** mostra i casi d'uso del software. Ogni partita inizia con l'inserimento del proprio nome utente, la selezione del personaggio e del livello di difficoltà. Una volta avviata la partita il giocatore dovrà muoversi nella mappa. L'incontro con i vari ostacoli e premi modifica l'esito del gioco. A fine partita, il risultato dell'utente è salvato nella classifica e ne vengono mostrati i migliori dieci risultati se, invece, l'utente incontra un NPC, il gioco termina mostrando la schermata di Game-Over e il suo risultato, mentre la classifica non viene stampata.

La **figura 2** mostra il modello del dominio rappresentato con le interfacce e una classe astratta. Si possono vedere fin da subito le relazioni tra le diverse entità: le più importanti sono quelle di aggregazione e composizione che riguardano le entità **Player**, **IGameMap**, **IResult** e **IRanking**; è presente anche la relazione di utilizzo.

2 Design

2.1 Architettura

L'architettura del software si basa sul pattern MVC (Model - View - Controller). Il Controller funge da disacoppiatore, in quanto elimina ogni interazione diretta tra Model e View passando i riferimenti del primo alla View e utilizzando i metodi pubblici delle entità del modello per modificare i dati a fronte di ogni evento generato dal viewer.
Inoltre, abbiamo integrato il pattern Observer, secondo cui ci sono dei generatori di eventi e degli osservatori di quest'ultimi: i primi si trovano nella view, mentre i secondi sono nel modello e vengono attivati dal controller, il quale fa da intermediario.

2.2 Design Dettagliato

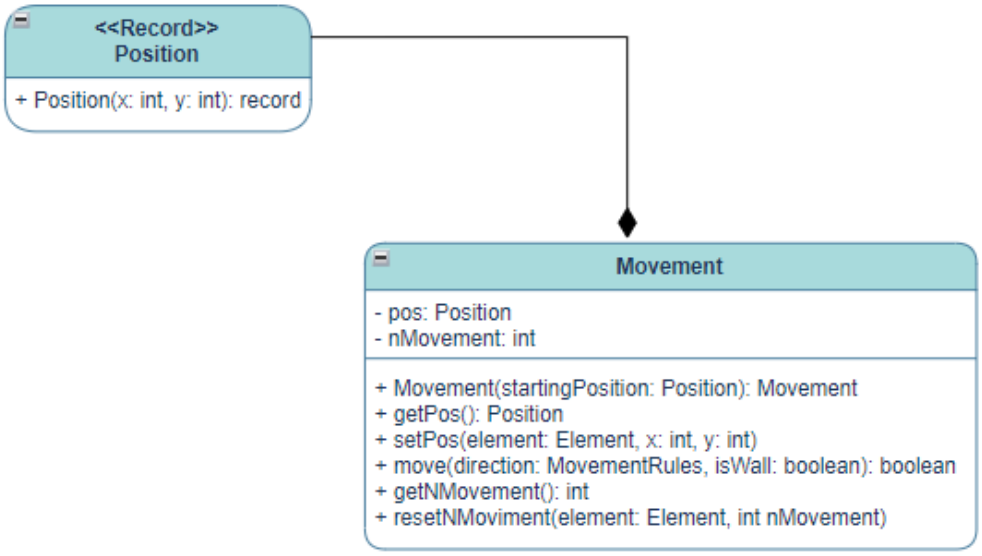
In questa sezione abbiamo riportato i diagrammi UML con le relative descrizioni delle entità che rappresentano le macro componenti del nostro progetto. Oltre all'impostazione architetturale del software secondo i pattern MVC e Observer, abbiamo utilizzato il design pattern creazionale **Singleton** laddove non era necessario avere più istanze della stessa entità. Per realizzarlo abbiamo reso privato il costruttore e aggiunto un metodo statico e pubblico che lo chiama restituendo la nuova, o la precedente, istanza dell'entità.
Le entità che utilizzano il design pattern Singleton sono le seguenti:

- Controller, appartenente al controller.
- Fonzie, appartenente al modello.
- Linda, appartenente al modello.
- FileMask, appartenente al modello.
- Ranking, appartenente al modello.
- Result, appartenente al modello.
- User, appartenente al modello.
- GameOverFrame, appartenente alla view.
- InputPlayer, appartenente alla view.
- MatrixPrinter, appartenente alla view.

Abbiamo inoltre deciso di utilizzare il pattern comportamentale **Template Method** che consente alle entità Fonzie e Linda di personalizzare l'effetto della banana in base al personaggio.

2.3 Design del Modello

2.3.1 Gestione delle Posizioni e del Movimento



Problema

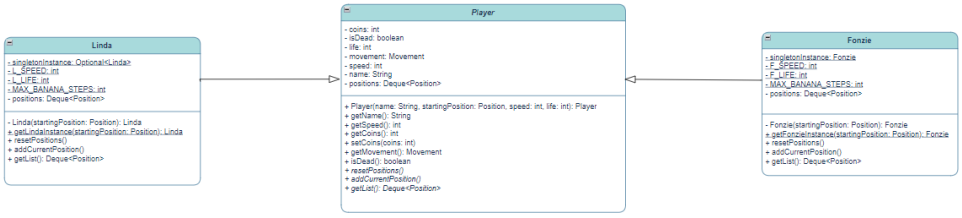
Una parte fondamentale del gioco è il movimento. L'implementazione deve tenere conto del fatto che se l'utente si trova nella posizione `(x,y)`, allora un movimento in avanti o indietro deve produrre rispettivamente un incremento o un decremento del valore della coordinata `row`. Viceversa, ogni spostamento verso sinistra o destra deve comportare rispettivamente un incremento o un decremento del valore della coordinata `column`.

Soluzione

Date le premesse del problema, è stato necessario legare il movimento del personaggio alla struttura della mappa, dato che questa si compone di un numero finito di celle identificabili con una coppia di coordinate. Il movimento, pertanto, va a modificare i valori delle coordinate della posizione attuale in base a dove l'utente intende spostarsi. Per implementare queste funzionalità abbiamo creato le entità **Position** e **Movement** che gestiscono rispettivamente le posizioni e il movimento. La prima rappresenta le due coordinate

x e y che identificano la posizione del personaggio sulla mappa; la seconda è in relazione di aggregazione con Position in quanto viene inizialmente creata un'istanza di quest'ultima per impostare la posizione di partenza, che sarà poi modificata in seguito a ogni spostamento che il giocatore intende effettuare. Inoltre Movement fa uso dell'enumerato **MovementRules** che stabilisce le regole di movimento. Questo, a seconda della direzione desiderata, genera una coppia di coordinate da sommare a quelle attuali consentendo così lo spostamento. Se, ad esempio, sono in posizione `row = 4` e `column = 3`, in caso di spostamento a sinistra MovementRules restituisce la coppia (0, -1) che, sommata alla coppia di coordinate della posizione attuale, fa diminuire il valore di `column` di 1.
Movement è in relazione di composizione con l'entità **Player**, mentre Position è in relazione di composizione con **GameDifficulty** e di aggregazione con **GameMap**

2.3.2 Creazione del Personaggio



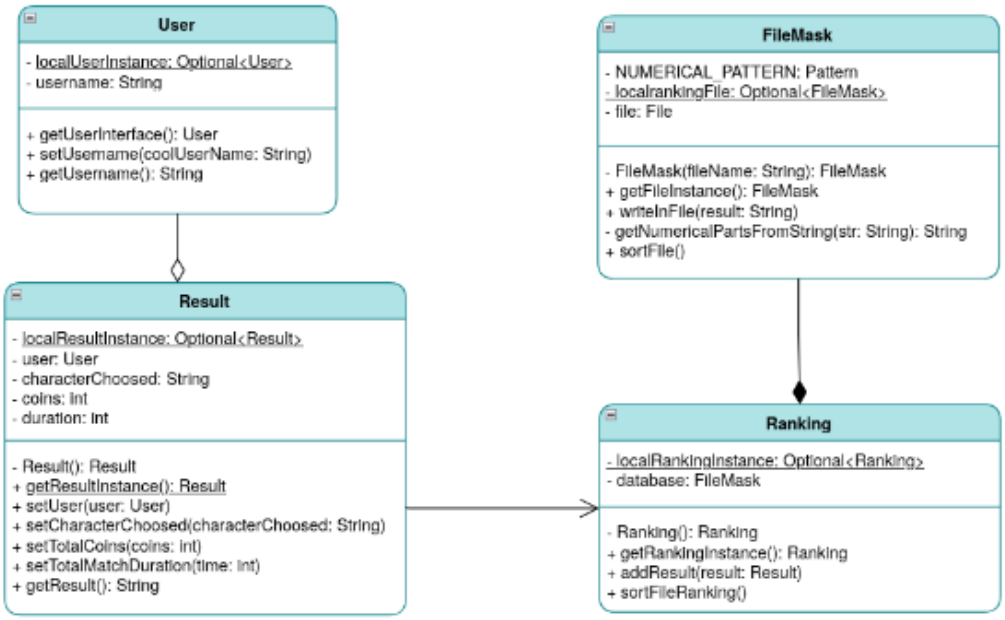
Problema

L'utente gioca con un personaggio a scelta tra Linda e Fonzie. Le informazioni importanti da memorizzare legate al personaggio sono la vita (fondamentale in caso di uccisione da parte di un NPC), la velocità (serve per impostare la lunghezza dei movimenti) e il numero di monete raccolte (utile per ottenere la posizione nel ranking).
Un altro aspetto importante da gestire è dato dall'effetto dell'ostacolo banana, che fa retrocedere di cinque posizioni il giocatore, nel caso in cui venga usato il personaggio Fonzie, di sette nel caso in cui si stia giocando con Linda.

Soluzione

Per implementare queste funzionalità abbiamo creato una classe astratta chiamata **Player**, che viene estesa dalle altre due rappresentanti i personaggi, quindi **Linda** e **Fonzie**. In questo modo si riducono le ripetizioni di codice dovute alle caratteristiche in comune tra i due personaggi, evitando così di violare il principio DRY.
Per risolvere la questione legata all'effetto della banana, abbiamo deciso di memorizzare le ultime posizioni in cui il personaggio è andato. Per questo motivo, Player, oltre ai getter e i setter, espone i metodi astratti `resetPosition()` e `addCurrentPosition()`: il primo svuota la lista di posizioni dopo che è stata toccata la banana, mentre il secondo salva le posizioni occupate in una coda, evitando che si ecceda la massima capacità prefissata, Da notare che l'effetto è attivato da altre entità, ma i metodi che permettono all'entità di subire l'effetto sono quelli sopra specificati (con l'aggiunta del metodo astratto `getList()`) poiché riguardano il personaggio. I metodi astratti dell'entità Player sono implementati dalle entità Fonzie e Linda, in quanto devono personalizzare il numero di passi indietro da effettuare quando viene colpita la banana.

2.3.3 Gestione dei Risultati e del Ranking



Problema

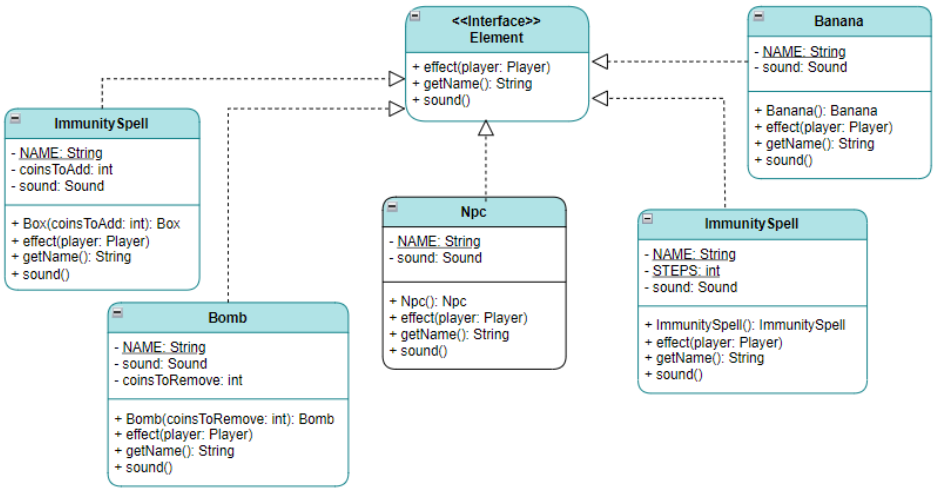
In caso di vittoria, al termine di ogni partita l'utente deve poter visualizzare il proprio risultato e una classifica contente i dieci migliori risultati, mentre in caso di sconfitta deve poter vedere solo il proprio. I risultati vanno salvati in un file a parte e ordinati in base al miglior tempo e numero di monetine (da qui devono essere presi i primi 10 da visualizzare in caso di vittoria).

Soluzione

Per implementare queste funzionalità vengono modellate le entità **User**, **Result**, **Ranking** e **FileMask**. User deve memorizzare il nome scelto dall'utente, Result, invece, deve contenere il risultato della partita appena terminata. Quest'ultimo è composto dal nome dell'utente - per cui risulta utile aggregare la classe User a Result - il numero di monete raccolte e la durata del game-play. Una volta impostati i campi compone il risultato.
L'entità Ranking deve occuparsi di scrivere dentro il file contente la classi-

fica e ordinarlo secondo il numero di monete ottenute e il tempo impiegato per terminare la partita. Queste attività di memorizzazione persistente e di ordinamento sono però affidate all'entità FileMask, la quale consente la scrittura dei risultati su un file di testo e il loro ordinamento confrontandone i parametri riga per riga.

2.3.4 Creazione dei Premi e degli Ostacoli



Problema

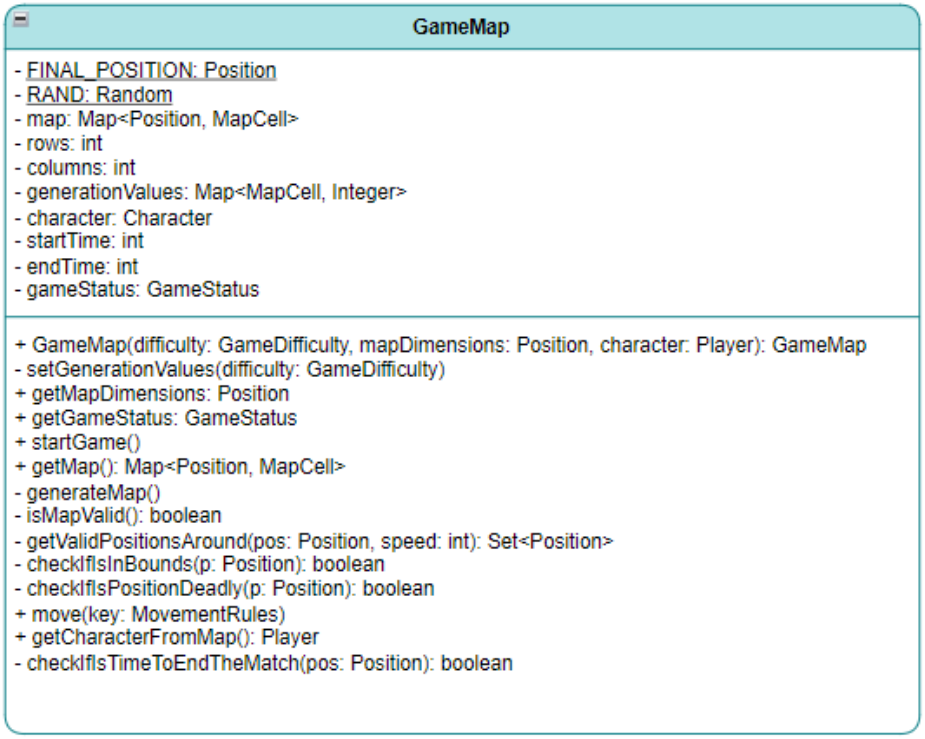
I premi e gli ostacoli sono le componenti principali del gioco poiché determinano il risultato finale della partita. I premi e gli ostacoli devono essere posizionati in modo randomico sulla mappa e in quantità differenti a seconda del livello di difficoltà selezionato.
Le casse e gli incantesimi immunità una volta colpiti devono modificare lo stato del personaggio, le prime aggiungendo delle monete a quelle raccolte e i secondi una vita che rende immuni agli NPC. Gli ostacoli - NPC, banana e bomba - invece, devono modificare rispettivamente la vita del personaggio,

la sua posizione e il numero di monete.

Soluzione

I premi e gli ostacoli hanno un proprio effetto, nome e suono; queste informazioni sono gestite da appositi metodi che, essendo comuni a entrambi, sono stati incapsulati dentro l'interfaccia **Element**. Questa deve essere implementata dalle classi **Box**, **ImmunitySpell**, **NPC**, **Banana** e **Bomb**, le quali dovranno implementarne i metodi inserendo il corretto effetto, nome e suono. Ogni chiamata al metodo `effect()` avrà una conseguenza sul personaggio, quindi il metodo avrà come parametro l'entità Player per poterne chiamare i metodi che ne modificano i campi.
Inoltre, l'entità **Sound** si compone di tutte e cinque le entità sopra descritte.

2.3.5 Generazione della Mappa



Problema

La generazione della mappa è una componente cruciale del progetto. Ne devono essere gestite la creazione dinamica e l'inserimento degli elementi all'interno, garantendo validità e coerenza della mappa stessa.

Soluzione

La classe responsabile di questi processi è `GameMap`, che effettua i seguenti passaggi:

- **Creazione dei Valori di Generazione (`createsGenerationValues`)**: All'interno del costruttore della classe `GameMap`, dopo aver inizializzato le dimensioni della mappa e il riferimento al personaggio, viene chia-

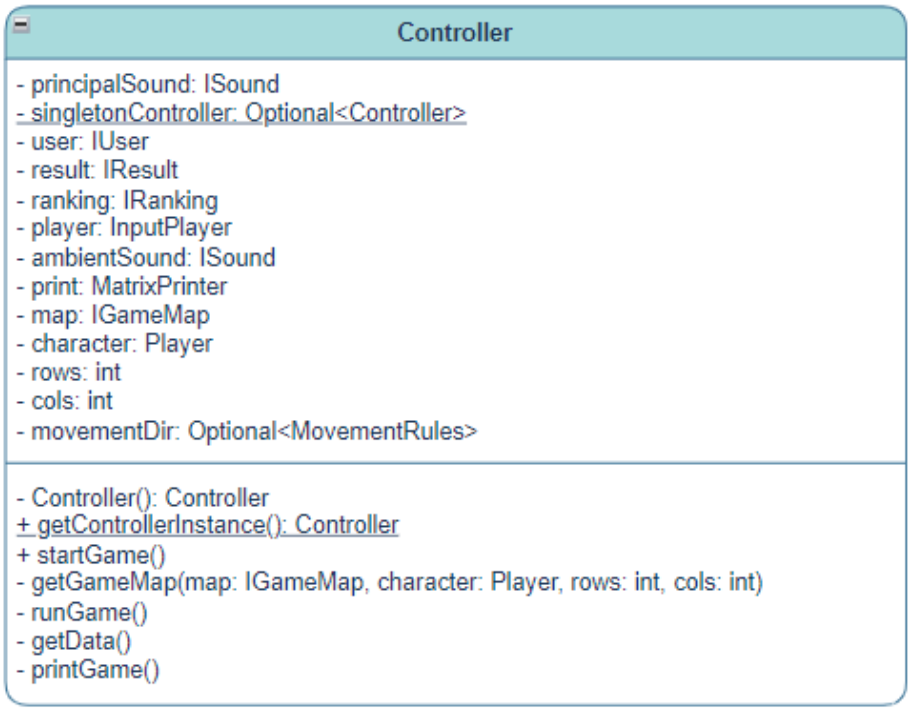
mato il metodo `createsGenerationValues` a cui viene passato come parametro il livello di difficoltà scelto dall'utente. Questo metodo calcola il numero di celle libere sulla mappa e ne rimuove una certa quantità per garantire che ci sia almeno una via percorribile e non siano presenti solo muri. Successivamente inserisce questi valori in una mappa per la generazione degli elementi.

- **Metodo** `generateMap`: La mappa viene generata in un ciclo `while` fino a che non viene considerata valida (massimo 30.000 generazioni). Durante le iterazioni iniziali vengono create le mura di confine. Successivamente, tutte le posizioni vuote vengono raccolte e gli elementi vi vengono inseriti basandosi sui valori precedentemente calcolati. Se durante il posizionamento si incontrano muri, questi vengono generati in coppia per formare una struttura coerente. Dopo aver popolato la mappa, viene inserito il punto di uscita ed eseguito un controllo di validità tramite `isMapValid`.

- **Validazione della Mappa (**`isMapValid`**)**: Questo metodo utilizza un approccio di ricerca basato su un insieme ordinato di posizioni da visitare, iniziando da quella iniziale del personaggio. Per ogni posizione esaminata vengono calcolate tutte le possibili posizioni raggiungibili e, se uno di questi percorsi permette di arrivare al punto di uscita, la mappa viene considerata valida.

Questo processo garantisce una mappa dinamica e coerente che supporta il gioco, mantenendo un equilibrio tra difficoltà e giocabilità.

2.4 Design del Controller



Problema

Il Controller è il componente che funge da intermediario tra il modello e la vista in un'architettura MVC. Nel nostro caso il controller ha il compito di aggiornare le informazioni sulla partita a fronte di ogni movimento, modificare i dati del modello come il nome utente, il giocatore selezionato, il livello scelto e infine il risultato parziale che farà parte del ranking.

Soluzione

Il Controller è formato da una sola classe che si occupa di mettere in comunicazione il modello con la view. Il compito è quindi quello di memorizzare le due istanze e chiamare i rispettivi metodi a fronte di ogni evento generato

dall'utente (gli eventi provengono dall'interfaccia grafica).
La classe **Controller** sta principalmente in ascolto degli input generati dal giocatore. Questi sono dati dalla pressione dei tasti sulla tastiera e dalla scelta dei livelli e personaggi per avviare la partita.
I principali metodi che implementano questa logica sono `getGameMap()`, `runGame()` e `getData()`. Il primo si occupa di ascoltare ciò che viene dai tasti per il movimento del personaggio e inoltrare gli input al modello, il secondo controlla lo stato della partita (che può assumere i valori `GAME_LOST` `GAME_INPROGRESS` e `GAME_WON`) e agisce di conseguenza, mentre l'ultimo serve a ottenere informazioni sul nome utente scelto.
Il Controller si compone delle classi **User**, **Result**, **Ranking**, **InputPlayer** e **MatrixPrinter**, mentre imposta una relazione di aggregazione con la classe **GameMap**

2.5 Design della View

La View ha il compito di mostrare la partita all'utente e di gestire la raccolta degli input che poi saranno inoltrati al controller; i dati di input si riferiscono principalmente al nome utente, il personaggio scelto, il livello di difficoltà da affrontare e i movimenti rilevati attraverso i tasti W, A, S, D. (rispettivamente, avanti, sinistra, indietro, destra). Si pone, inoltre, il problema di stampare il ranking in caso di vittoria e di mostrare una schermata di game over in caso di sconfitta.
Per la creazione della view abbiamo sviluppato quattro entità principali:

- FileViewer, che mostra all'utente la sua posizione sul ranking quando l'utente non viene ucciso dagli NPC e quindi riesce ad uscire dalla mappa, assieme al risultato della partita appena terminata.
- MatrixPrinter, che si occupa di stampare la mappa e di aggiornarla a fronte di ogni spostamento dell'utente.
- GameOverFrame, che comunica all'utente che ha perso in seguito a un'uccisione per mano di un NPC.
- InputPlayer, che mostra la finestra di avvio del gioco tramite cui l'utente riesce a inserire i dati per far cominciare la partita.

Le entità della view non sono in relazione tra loro, infatti è il controller che si compone della classe MatrixPrinter e InputPlayer e utilizza i metodi delle classi GameOverFrame e FileViewer.

3 Sviluppo

3.1 Testing Automatizzato

Per il sistema di entità sono state testate le caratteristiche principali di ciascuna delle macro-implementazioni ovvero:

- Per il package **Characters** è stata testata la classe Fonzie nella sua interezza.
- Per il package **Elements** sono state testate tutte le classi nella loro interezza, ad eccezione della classe banana che è stata testata solo nel caso di una retrocessione di cinque passi.
- Per il package **GameMap** sono stati testati i metodi principali della parte core del modello (presenti nella classe GameMap).
- Per il package **Movement** sono state testate tutte le classi nella loro interezza.
- Per il package **Ranking** è stata testata la classe FileMask nella sua interezza.
- Per il package **Sounds** è stata testata la classe Sound nella sua interezza

3.1.1 Characters

Per il package Characters ci siamo maggiormente concentrati sul testing di una sola classe per via delle poche differenze con le altre.
La classe è stata testata per tutti i suoi metodi ovvero:

- Costruttore, abbiamo testato la corretta costruzione della classe con tutti i suoi attributi iniziali, cioè la velocità, le monete e il nome.
- Metodi per gestire le monete del personaggio.
- Metodi per gestire la vita del personaggio.
- Metodi per il corretto funzionamento del salvataggio e la cancellazione delle posizioni, essenziali per l'elemento banana.
- Design Pattern Singleton, abbiamo testato il corretto funzionamento del pattern in questione.

3.1.2 Elements

Poiché tutti gli elementi del package sono fondamentali, abbiamo deciso di testarli tutti, però a causa dell'implementazione del pattern Singleton, il test della banana, se eseguito con tutti gli altri test contemporaneamente, non sempre va a buon fine. Seguono i test effettuati per ogni elemento:

- Box, testato il suo effetto di aggiunta delle monete al giocatore.
- ImmunitySpell, testato il suo effetto di immunità verso gli NPC.
- Banana, testato solo il suo effetto di far tornare indietro di 5 posizioni il personaggio.
- Bomb, testato il suo effetto di sottrarre monete al giocatore.
- Npc, testato il suo effetto di togliere una vita al personaggio.

3.1.3 Gamemap

Per il package Gamemap, core del modello, ci siamo concentrati sul testing dei metodi più importanti, ovvero:

- TestMapGeneration, si occupa di verificare il corretto funzionamento del corrispondente metodo in GameMap per generare dinamicamente la mappa.
- TestRandomElementsGeneration , si occupa del verificare se gli elementi nella mappa sono stati generati correttamente non nella posizione ma nel numero.
- TestMapValidity, si occupa di verificare se, a fronte dei precedenti test, si ha una mappa valida per giocare, ovvero una mappa priva di collisioni e dove vi è sempre un percorso per la risoluzione del gioco.

3.1.4 Movement

Per il package Movement ci siamo concentrati sul testing di ogni classe poiché sono essenziali nel modello:

- Movement, testati tutti i suoi metodi e il costruttore, nello specifico è stato testato il corretto funzionamento del movimento in tutte le sue direzioni: su, giù, destra, sinistra, se incontra un muro e i reset del numero di passi del giocatore.
- Position, testati tutti i suoi metodi.

3.1.5 Ranking

Per il package Ranking ci siamo concentrati sul testing della sola omonima classe, poiché core del package, per controllarne il corretto funzionamento. Testati i suoi metodi:

- TestWriteInFile, si occupa di verificare se il file viene popolato correttamente o meno in base alle righe scritte.
- TestSortFile, si occupa di verificare se il sorting del file viene eseguito correttamente o meno, controllando il file prima e dopo il sorting e verificando se tutti gli elementi nel file sono ordinati correttamenente

3.1.6 Sound

Per il package Sound ci siamo concentrati nel testing di ogni suono presente nel gioco e del corretto funzionamento dell'omonima classe che ha i seguenti metodi:

- PlaySound, ulteriormente testata in tutti i suoni la variante mono, invece la loop (ovvero riproduzione continua del suono) una singola volta per semplicità.
- StopSound, testata su un singolo suono per semplicità.
- IsRunning, testata su un singolo suono per semplicità.

3.2 Metodologia di Lavoro

Strumenti Utilizzati:

- Git: utilizzato per il versionamento e lo sviluppo cooperativo.
- Visual Studio Code: utilizzato come IDE.
- GitHub: utilizzato come repository principale.
- draw.io: utilizzato per la creazione e la modellazione dell'UML.
- Overleaf: utilizzato per la redazione della relazione.

Per quanto riguarda la divisione dei compiti, il progetto è stato seguito da tutti i membri in tutte le sue fasi, tuttavia alcuni componenti si sono focalizzati verso aree specifiche del progetto:

- Controller, Viewer e Suoni: Giuseppe Benedetti e Eloi Ricci.
- Mappa, Movimento e Posizione: Elia Renzoni e Eloi Ricci.
- Personaggi ed Elementi: Annarosa Clemente e Giuseppe Benedetti.
- User e Ranking: Elia Renzoni e Giuseppe Benedetti.
- Testing: Giuseppe Benedetti.
- UML: Annarosa Clemente, Elia Renzoni.
- Relazione: Annarosa Clemente.

3.3 Note di Sviluppo

Attraverso questo progetto abbiamo appreso e consolidato concetti come **record** e **stream**. I primi, introdotti in Java 14, sono utili per la creazione di entità contenenti dati immutabili, mentre gli ultimi ci sono serviti per rendere più moderno e arricchito il codice del nostro progetto.
Oltre a concetti puramente tecnici abbiamo capito l'importanza di standardizzare alcuni aspetti come l'indentazione, l'uso dello stesso editor e lo stile di programmazione.
In particolare ogni membro del gruppo ha imparato i seguenti aspetti:

- Elia Renzoni ha consolidato l'uso di Git e l'utilizzo di aspetti avanzati di Java come le stream. Inoltre ha imparato a creare il design di un software partendo da idee e concetti, infine ha appreso l'importanza di una buona comunicazione tra i membri del team, per evitare che prevalgano idee senza prima essere concordate e analizzate correttamente. Le principali difficoltà incontrate sono state la gestione del movimento e la creazione della mappa con le limitazioni sul numero di ostacoli e sulle mura.

- Annarosa Clemente ha consolidato l'uso di Git, dell'IDE Visual Studio Code e di aspetti avanzati come le lambda e gli stream; inoltre, ha imparato a modellare software prima di implementarlo e l'importanza di concordare le proprie idee con il team di sviluppo; le principali difficoltà incontrate riguardano la comprensione iniziale del nuovo ambiente di sviluppo e, nei primi tempi, l'installazione e il setting di Git.

- Giuseppe Benedetti ha consolidato le competenze tecniche come: l'uso di Git, dell'IDE VSCodium, uso del LaTeX per le relazioni, aspetti avanzati di Java come: le stream, i record, le operazioni su file, il suono, la grafica tramite Java Swing e i test automatizzati junit.
  Inoltre ha imparato l'importanza di una buona comunicazione con i membri del team, che comprende l'organizzazione del lavoro con il metodo top-down (dalla modellazione al design di un software), l'importanza di eseguire al meglio, in tempo, i propri compiti e non fare di testa propria, ma avere uno spirito pronto al confronto ascoltando gli altri membri del team.

- Eloi Ricci ha consolidato l'uso di aspetti avanzati come stream, lambda ed hash map. Le difficoltà incontrate erano relative al trovare modi per

gestire la risolvibilità (cioè che esista un percorso dall'inizio alla fine) della mappa, risolti con un BFS modificato. Anche se ha fallito in questo aspetto, ha imparato a comunicare le proprie idee sul progetto