

FUGA DALLA CRIPTA

NATRELLA DANIELE

Esame di

Metodi Avanzati di Programmazione

(track M-Z)

A.A. 2024/2025

Descrizione dell'avventura

“Fuga dalla Cripta” è un'avventura testuale/grafica ambientata in un luogo misterioso e sconosciuto. Il giocatore si risveglia senza memoria in una cripta fredda e oscura, e deve esplorare le varie stanze per trovare una via d'uscita e fuggire.

Durante l'avventura, il giocatore può utilizzare comandi testuali per muoversi tra le stanze, interagire con gli oggetti e risolvere enigmi necessari per proseguire nell'esplorazione.

L'obiettivo finale è riuscire a scappare dalla cripta prima che le forze si esauriscano.

Al termine della partita, il tempo di gioco viene salvato e mostrato in una classifica, chiedendo al giocatore di inserire il proprio username per registrare il punteggio.

Progettazione

Durante la fase di progettazione dell'avventura testuale/grafica *Fuga dalla Cripta*, il sistema è stato organizzato in più package per garantire una chiara separazione delle responsabilità e una maggiore manutenibilità del codice.

Ogni package contiene classi con competenze specifiche, progettate secondo i principi della programmazione orientata agli oggetti.

- 1) **Package Game:** rappresenta il cuore logico dell'applicazione, ovvero il motore dell'avventura. Contiene tutte le classi principali necessarie per gestire la struttura del gioco, il flusso degli eventi e la comunicazione tra le varie componenti.

Classe Engine: costituisce il motore principale del gioco. Gestisce l'esecuzione dell'avventura testuale, ricevendo i comandi del giocatore, interpretandoli tramite il parser e aggiornando lo stato del gioco. Si occupa inoltre di avviare e fermare il timer, di comunicare con il server della classifica per salvare e mostrare i punteggi finali, e di gestire l'interfaccia testuale tramite input e output da console.

Classe astratta GameDescription: definisce la struttura e i componenti principali dell'avventura. Contiene le liste delle stanze, dei comandi e dell'inventario, e gestisce la stanza corrente e lo stato del gioco.

Interfaccia GameObservable: definisce i metodi per implementare il design pattern *Observer*, permettendo a un oggetto di essere osservato da altri.

Fornisce i metodi `attach()` e `detach()` per gestire gli osservatori, e `notifyObservers()` per inviare notifiche in caso di modifiche allo stato del gioco.

Interfaccia GameObserve: definisce il comportamento degli osservatori nel pattern *Observer*. Ogni classe che la implementa può essere notificata dei cambiamenti di stato del gioco.

Classe GameUtils: contiene metodi di supporto per la gestione del gioco.

Fornisce metodi statici riutilizzabili, come la ricerca di un oggetto nell'inventario tramite il suo identificatore.

Classe Utils: contiene metodi di supporto per la gestione del testo e dei file.

Permette di leggere file di testo e di estrarre insiemi di parole (come le *stopwords*), oltre a fornire funzioni per l'analisi linguistica di stringhe.

- 2) **Package Game.database:** Gestisce la connessione con il database
Classe dbSetup: si occupa dell'inizializzazione del database, del caricamento dinamico delle stanze e delle loro connessioni, e dell'apertura/chiusura della connessione.

- 3) **Package Game.impl:** contiene l'implementazione dell'avventura, In esso viene definita la logica del gioco, la disposizione delle stanze, gli oggetti e i comandi che il giocatore può utilizzare
- Classe FugaCRIPTA:** è il cuore logico dell'avventura: definisce comandi, ambienti, oggetti e interazioni del gioco.
- Classe InventoryObserver:** è l'osservatore responsabile della gestione dell'inventario. Controlla se il giocatore possiede oggetti e mostra le relative informazioni quando viene eseguito il comando "inventario".
- Classe LookAtObserver:** gestisce il comando di osservazione del giocatore, mostrando la descrizione dettagliata della stanza o un messaggio predefinito se non c'è nulla di particolare da vedere.
- Classe MoveObserver:** controlla e gestisce i movimenti del giocatore tra le stanze dell'avventura.
- Classe OpenObserver:** controlla se un oggetto può essere aperto
- Classe PickupObserver:** gestisce l'azione di raccogliere gli oggetti. Controlla se l'oggetto può essere preso, aggiorna l'inventario del giocatore e modifica la descrizione della stanza di conseguenza.
- Classe UseObserver:** gestisce le interazioni tra gli oggetti (come usare la torcia, riempire la coppa, sbloccare il sarcofago).
- 4) **Package Game.parser:** contiene le classi per l'analisi e l'interpretazione dei comandi testuali inseriti dal giocatore
- Classe Parser:** trasforma il linguaggio naturale del giocatore in azioni eseguibili dal sistema.
- Classe ParserOutput:** serve per memorizzare il risultato dell'interpretazione del comando del giocatore.
- 5) **Package Game.socket:** contiene le classi per la comunicazione tramite rete tra client e server
- Classe LeaderboardClient:** permette al gioco di collegarsi al server della classifica per inviare il punteggio del giocatore a termine della partita e per ricevere la classifica aggiornata
- Classe LeaderboardServer:** coordina la classifica del gioco, riceve i punteggi dai giocatori e restituisce la classifica ordinata.
- 6) **Package Game.swing:** contiene le classi per la gestione dell'interfaccia grafica
- Jframe interfaccia:** rappresenta la schermata iniziale del gioco. È la finestra di avvio da cui il giocatore può cominciare la partita.
- Jframe GameFrame:** è la finestra principale del gioco, gestisce l'interazione dell'utente con l'avventura testuale
- Jframe LeaderBoardFrame:** è la finestra grafica della classifica del gioco. Viene mostrata quando il giocatore termina la partita per visualizzare i migliori tempi registrati.
- 7) **Package Game.thread:** contiene la classe per la gestione del timer
- Classe timer:** serve per misurare il tempo di gioco in modo semplice e indipendente tramite un thread dedicato.

8) **Package Game.type:** contiene le classi fondamentali che rappresentano gli elementi base del gioco

Classe AdvObject: serve per modellare gli oggetti del mondo di gioco, fornendo le informazioni e le proprietà necessarie per permettere al motore di gioco di gestirne le interazioni.

Classe AdvObjectContainer: rappresenta un oggetto che può contenere altri oggetti

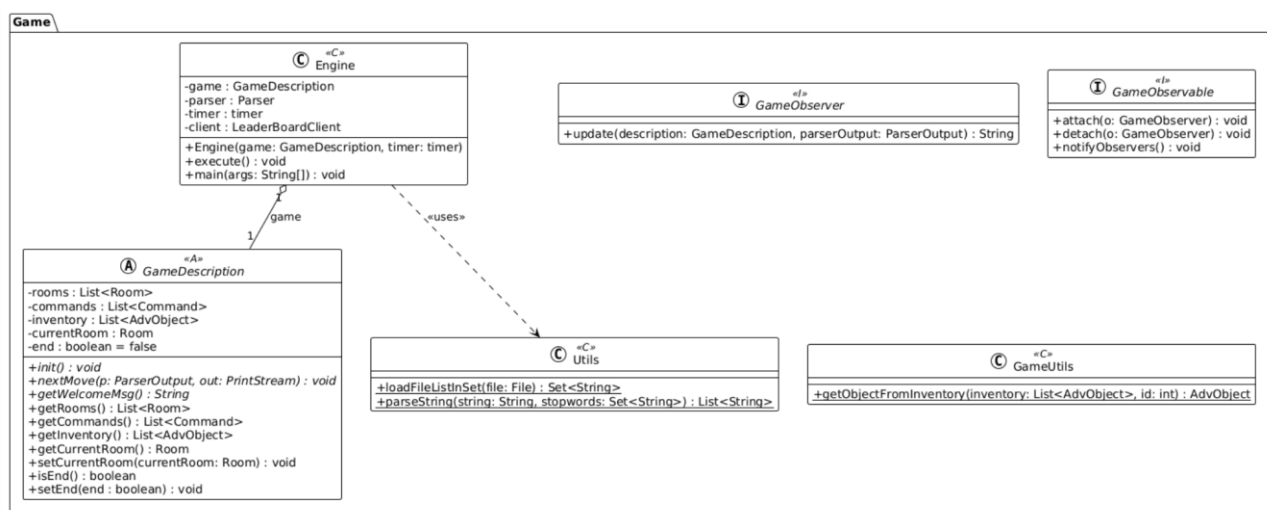
Classe Command: serve a definire e gestire i comandi del giocatore nel gioco di avventura testuale, permettendo di associare ciascun comando a un tipo d'azione e ai suoi sinonimi.

Classe CommandType: elenca tutte le possibili azioni riconosciute dal motore del gioco

Classe Inventory: è una semplice struttura dati che permette di gestire gli oggetti raccolti dal giocatore

Classe Room: definisce le stanze del mondo di gioco, gestendo le loro connessioni, descrizioni e oggetti contenuti, permettendo così la navigazione e l'interazione all'interno dell'avventura.

Diagramma delle classi



Il diagramma seguente illustra la struttura delle classi contenute nel package Game.

Questo package definisce il framework astratto e i componenti logici fondamentali dell'applicazione, agendo come **“cuore”** del sistema.

Il design di questo package è incentrato sul **disaccoppiamento dei componenti**, separando l'engine di gioco dalla logica specifica dell'avventura. Questo obiettivo è raggiunto attraverso l'applicazione di diversi principi chiave della programmazione a oggetti.

Il componente centrale del package è **GameDescription**, una **classe astratta** che funge da modello per qualsiasi implementazione concreta di gioco.

- **Astrazione:** la classe definisce la struttura dati comune (stanze, comandi, inventario) e i comportamenti fondamentali che ogni gioco deve possedere.

- **Ereditarietà:** stabilisce un contratto per le sottoclassi. Ogni classe concreta (ad esempio FugaCripa nel package Game.impl) deve estendere GameDescription e implementarne i metodi astratti.

Il package include anche due **interfacce**, GameObserver e GameObservable, che implementano il **design pattern Observer**, consentendo la comunicazione tra l'engine e i componenti dell'interfaccia utente. Queste interfacce sono indipendenti e favoriscono un'architettura estensibile e modulare. Le interfacce appaiono isolate in questo diagramma poiché le loro implementazioni risiedono in package esterni.

La relazione principale nel diagramma è l'**aggregazione** tra Engine e GameDescription: l'oggetto Engine possiede un riferimento a GameDescription, ma non dipende da una specifica implementazione concreta.

Questo approccio realizza un forte **principio di inversione delle dipendenze**, rendendo Engine riutilizzabile per qualsiasi avventura conforme al contratto definito da GameDescription.

Specifica algebrica

Specifica algebrica della Lista

Specifica sintattica

sorts: lista, item, boolean, nat

operations:

newlist() → **lista** (crea una lista vuota)

cons(item, lista) → **lista** (aggiunge un elemento alla lista)

head(lista) → **item** (restituisce il primo elemento)

tail(lista) → **lista** (restituisce la lista senza il primo elemento)

isnew(lista) → **boolean** (controlla se la lista è vuota)

length(lista) → **nat** (conta gli elementi)

append(lista, lista) → **lista** (concatena due liste)

Specifica semantica

declare l: lista, x: item

head(cons(x, l)) = x (Se creo una nuova lista aggiungendo un elemento x davanti a un'altra lista l, allora la testa della nuova lista è proprio x)

tail(cons(x, l)) = l (tutti gli elementi tranne il primo della lista [x] + l è semplicemente l)

isnew(newlist) = true (La lista creata con newlist è nuova)

isnew(cons(x, l)) = false (Se aggiungo almeno un elemento a una lista, allora non è più vuota)

length(newlist) = 0 (La lunghezza della lista vuota è 0)

length(cons(x, l)) = 1 + length(l) (Se aggiungo un elemento x davanti a una lista l, la nuova lunghezza è una in più rispetto a quella di l)

append(newlist, l) = l (Concatenare una lista vuota con una lista l restituisce semplicemente l)

append(cons(x, l1), l2) = cons(x, append(l1, l2)) (Se concateno una lista l1 che comincia con x e continua con altri elementi, a una lista l2, il risultato è una lista che comincia con x e poi ha la concatenazione del resto (l1) con l2)

Specifica di restrizione

restrictions

```
head(newlist) = error  
tail(newlist) = error
```

Dettagli implementativi

Programmazione generica

Nel progetto è stata ampiamente utilizzata la programmazione generica di Java per garantire sicurezza di tipo e riutilizzabilità del codice. Le collezioni come `List` e `Set` sono state tipizzate per contenere oggetti specifici, evitando la necessità di conversioni esplicite e riducendo il rischio di errori a runtime. Ad esempio, nelle classi `Inventory`, `Room` e `AdvObjectContainer` sono presenti liste generiche (`List<AdvObject>`) utilizzate per gestire gli oggetti presenti rispettivamente nell'inventario, nelle stanze e nei contenitori. Analogamente, nelle classi `AdvObject` e `Command` vengono impiegati insiemi generici (`Set<String>`) per rappresentare gli alias di oggetti e comandi. Anche nella classe grafica `LeaderBoardFrame` è stata utilizzata una lista generica (`List<String[]>`) per aggiornare dinamicamente la tabella della classifica. L'uso dei generics ha reso il codice più modulare, leggibile e coerente, semplificando la gestione di collezioni omogenee all'interno del gioco.

File

Nel progetto i file vengono utilizzati per gestire sia la persistenza dei dati, sia il caricamento di contenuti testuali del gioco.

In particolare, nella classe `LeaderboardServer` viene creato e mantenuto un file di testo (*scores.txt*) che memorizza i punteggi dei giocatori. Il salvataggio e il caricamento avvengono tramite le classi `FileWriter`, `PrintWriter`, `FileReader` e `BufferedReader`, permettendo di conservare una classifica permanente anche dopo la chiusura del programma.

Inoltre, i file vengono utilizzati anche per leggere e mostrare messaggi all'interno del gioco, come ad esempio il messaggio di benvenuto visualizzato all'avvio.

Database (JDBC)

Nel progetto il database viene utilizzato per gestire la creazione e il caricamento delle stanze di gioco. Tramite l'utilizzo delle API JDBC, l'applicazione si connette a un database in cui sono memorizzate le informazioni relative alle stanze, come ID, nome, descrizione e collegamenti con le altre stanze (nord, sud, est, ovest).

Lambda Expression (compresi stream e pipeline)

Nel progetto sono state utilizzate diverse espressioni lambda per rendere il codice più compatto e leggibile. Un primo esempio si trova nella classe *timer*, dove la lambda definisce il comportamento del thread che gestisce il conteggio del tempo. Un secondo esempio è presente nella classe *Room*, dove l'utilizzo della lambda all'interno di uno stream permette di filtrare e trovare rapidamente un oggetto con un determinato identificativo all'interno della lista della stanza. In entrambi i casi, l'uso delle espressioni lambda e degli stream ha semplificato la gestione di operazioni ripetitive e migliorato la chiarezza del codice, rendendolo più moderno ed espressivo.

SWING

Nel progetto è stata utilizzata la libreria Swing per la realizzazione dell'interfaccia grafica del gioco. In particolare, le classi del package `Game.swing`, come *interfaccia*, *GameFrame* e *LeaderBoardFrame*, gestiscono le varie finestre e componenti grafiche. Attraverso Swing sono stati implementati elementi come pulsanti, pannelli e label, che permettono all'utente di interagire in modo semplice e

immediato. La classe *interfaccia* rappresenta la schermata iniziale del gioco con il pulsante “GIOCA”, mentre *GameFrame* gestisce la finestra principale di gioco. Inoltre, la classe *LeaderBoardFrame* è stata creata per visualizzare la classifica dei migliori giocatori, comunicando con il server tramite socket. L’uso di Swing ha quindi consentito di costruire un’interfaccia completa, intuitiva e visivamente accattivante, migliorando l’esperienza di gioco complessiva.

Thread e programmazione concorrente

Nel progetto la programmazione concorrente è stata utilizzata in due punti principali.

La classe timer utilizza un thread dedicato per tenere traccia del tempo di gioco in background, aggiornando il contatore dei secondi senza interferire con l’esecuzione principale o con l’interfaccia grafica.

Inoltre, la classe *LeaderboardServer*, che implementa *Runnable*, sfrutta un thread separato per gestire la comunicazione di rete con i client. In questo modo il server della classifica può operare in parallelo rispetto al gioco, ricevendo e salvando punteggi senza bloccare l’esperienza del giocatore.

Socket e/o REST

Nel progetto è stato implementato un sistema di comunicazione client-server tramite socket TCP per gestire la classifica dei giocatori.

La classe *LeaderboardServer* si occupa di avviare un server in ascolto su una determinata porta, ricevendo le connessioni dei client. Quando un giocatore termina la partita, la classe *LeaderboardClient* si connette al server e invia il proprio nome utente insieme al tempo di completamento. Il server salva i punteggi in un file di testo e restituisce la classifica aggiornata.