**IL CASTELLO ENIGMATICO**

**INTRODUZIONE**

Questa documentazione rappresenta il resoconto finale del progetto dedicato alla creazione del gioco "*Il Castello Enigmatico*", un’avventura testuale basata sull’avventura di un giovane ragazzo all’interno di un mondo fantasy. Lo scopo principale di questo progetto è stato quello di creare un'avventura testuale interattiva destinata a utenti con una conoscenza di base sul suo funzionamento. L’interfaccia utente è stata implementata sia in modalità grafica (*GUI - Graphical User Interface*) che a riga di comando (*CLI - Command Line Interface*). Durante lo sviluppo del progetto, sono state applicate le competenze e le tecniche apprese durante il corso di *Metodi Avanzati di Programmazione*, concentrando l'attenzione sulla struttura modulare del codice e sull'efficienza nell'uso delle risorse. L’obiettivo principale è quello di dimostrare la capacità di applicare le conoscenze teoriche acquisite durante il corso con un’esperienza interattiva solida e funzionale.

****

INDICE

[1. MAPPA DEL GIOCO 3](#_Toc170506102)

[2. REQUISITI SPECIFICI 5](#_Toc170506103)

[2.1 Requisiti funzionali 5](#_Toc170506104)

[3. SPECIFICA ALGEBRICA DELLA STRUTTURA DATI “COMMAND LIST” 6](#_Toc170506105)

[4. DIAGRAMMA DELLE CLASSI 7](#_Toc170506106)

[Classe Story 8](#_Toc170506107)

[Classe SpeedRunTimer 8](#_Toc170506108)

[Classe DarkCrypt 8](#_Toc170506109)

[Classe Timer 9](#_Toc170506110)

[Classe UI 9](#_Toc170506111)

[Classe VisibilityManager 9](#_Toc170506112)

[Classe ParserOutput 10](#_Toc170506113)

[Classe Player 10](#_Toc170506114)

[Classe Map 10](#_Toc170506115)

[Classe Room 11](#_Toc170506116)

[Classe Stobj 11](#_Toc170506117)

[Classe Command 12](#_Toc170506118)

[Classe Door 12](#_Toc170506119)

[5. DESCRIZIONE ARGOMENTI UTILIZZATI 12](#_Toc170506120)

[5.1 Lamba expression 12](#_Toc170506121)

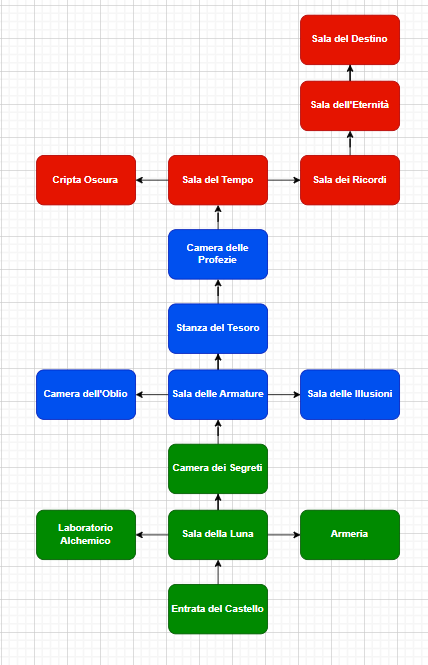
[5.2 Thread 13](#_Toc170506122)

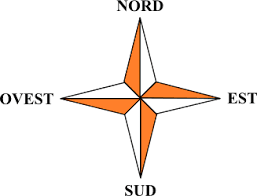
[5.3 Swing 14](#_Toc170506123)

[5.4 File 14](#_Toc170506124)

[5.5 REST 15](#_Toc170506125)

# 1. MAPPA DEL GIOCO





La Mappa del gioco è basata sull’interno del castello enigmatico di questo mondo fantastico. Un singolo piano è reso esplorabile in varie stanze che hanno certi temi specifici, con enigmi appropriati al tipo di stanza.

* Nell’*Entrata del Castello* possiamo osservare e raccogliere una *spada* che verrà utilizzata nella *Cripta Oscura* e un *guardiano* che fornisce un indizio cruciale nel corso dell’avventura.
* Nel *Laboratorio Alchemico* possiamo osservare e raccogliere una *pozione*, che verrà utilizzata per ripristinare la salute dell’avventuriero.
* Nell’*Armeria* è possibile osservare e raccogliere un’*ascia*.
* Nella *Camera dei Segreti* possiamo osservare e raccogliere una *moneta dorata*, che verrà utilizzata per aprire la porta presente nella *Stanza del Tesoro*.
* Nella *Sala delle Armature* è presente un *enigma*.
* Nella *Camera dell’Oblio* possiamo osservare e raccogliere una *pozione*, che verrà utilizzata per ripristinare la salute dell’avventuriero.
* Nella *Sala delle Illusioni* possiamo osservare tre personaggi: una *guardia*, una *signora* e l’*armaiolo*.
* Nella Stanza del Tesoro è possibile osservare e raccogliere una *moneta argentata* e un *enigma*. Per aprire la porta che conduce a *nord* bisogna raccogliere e quindi inserire la *moneta corretta*, se viene inserita la *moneta argentata*, vengono inflitti dieci punti di danno all’avventuriero, mentre se viene inserita la *moneta* *dorata* raccolta precedentemente nella *Camera dei Segreti*, la porta viene aperta.
* Nella *Camera delle Profezie* è possibile risolvere un *enigma* mediante l’ordine di attivazione delle tre leve.
* Nella *Sala del Tempo* si possono osservare tre *libri*: libro del *passato*, del *presente* e del *futuro*.
* Nella *Cripta Oscura* è possibile osservare una *statua piangente*, che rivelerà la verità dietro un’illusione attraverso l’utilizzo della *spada* precedentemente raccolta all’*Entrata del Castello*. La statua ormai trafitta lascia cadere un *cuore pulsante*, il quale sarà possibile raccogliere, e che verrà utilizzato in seguito per aprire una porta nella *Sala dell’Eternità*.
* Nella *Sala dei Ricordi* è possibile osservare e raccogliere una *chiave*, che verrà utilizzata per aprire una porta nella *Sala dell’Eternità*.
* Nella *Sala dell’Eternità* è possibile osservare e raccogliere una *chiave*, che verrà inserita all’interno della serratura della porta per poterla aprire. Successivamente, comparirà un’altra porta che verrà aperta attraverso l’inserimento del *cuore pulsante*, raccolto precedentemente nella *Cripta Oscura*.
* Nella *Sala del Destino* è possibile osservare una *pergamena*, ovvero l’*enigma*, e la risoluzione di quest’ultimo darà come premio l’*anello dell’eternità*. Raccolto e indossato l’*anello*, comparirà un messaggio di congratulazioni per aver terminato il gioco.

# 2. REQUISITI SPECIFICI

Questa sezione dettaglia tutti i requisiti funzionali necessari per il software "Il Castello Enigmatico".

## 2.1 Requisiti funzionali

I *FUR* (*Functional User Requirements*) delineano le capacità del software riguardanti:

* I servizi che il software deve rendere disponibili agli utenti.
* Le risposte attese dagli utenti in determinate situazioni.
* I risultati che il software deve generare in risposta a input specifici.

Tale applicazione fornisce le seguenti funzionalità:

* *Mostrare gli oggetti presenti nelle stanze* 🡪 al comando ‘osserva’ l’applicazione deve mostrare tutti gli oggetti presenti nella stanza corrente
* *Osservare uno specifico oggetto* 🡪 il comando ‘osserva’ seguito dal nome dell’oggetto interessato o un suo alias permette al giocatore di osservare un oggetto, imparando i suoi dettagli
* *Parlare* 🡪 al comando ‘parla’ seguito dal nome di un personaggio con cui si può interagire, il giocatore potrà sentire cosa ha da dire l’interlocutore interessato
* *Iniziare una nuova partita* 🡪 al Run del codice parte l’applicazione, nella GUI dopo aver cliccato sul tasto ‘Nuova partita’
* *Muovere a nord* 🡪 al comando ‘nord’, l’applicazione deve muovere il giocatore verso Nord, se il comando non è valido viene mostrato il messaggio ‘Non puoi proseguire in quella direzione’ e l’applicazione rimane in attesa di un comando valido.
* *Muovere a sud* 🡪 al comando ‘sud’, l’applicazione deve muovere il giocatore verso Sud, se il comando non è valido viene mostrato il messaggio ‘Non puoi proseguire in quella direzione’ e l’applicazione rimane in attesa di un comando valido.
* *Muovere a est* 🡪 al comando ‘est, l’applicazione deve muovere il giocatore verso Est, se il comando non è valido viene mostrato il messaggio ‘Non puoi proseguire in quella direzione’ e l’applicazione rimane in attesa di un comando valido.
* *Muovere a ovest* 🡪 al comando ‘ovest, l’applicazione deve muovere il giocatore verso Ovest, se il comando non è valido viene mostrato il messaggio ‘Non puoi proseguire in quella direzione’ e l’applicazione rimane in attesa di un comando valido.
* *Prendere un oggetto* 🡪 al comando ‘prendi’ l’applicazione deve prendere l’oggetto, inserirlo all’interno dell’inventario ed eliminarlo dalla stanza, se non ci sono oggetti viene mostrato il messaggio ‘Non ho capito cosa vorresti raccogliere. Specifica cosa raccogliere’ e l’applicazione rimane in attesa di un comando valido.
* *Usare un oggetto* 🡪 al comando ‘usa’ l’applicazione deve usare le proprietà dell’oggetto, se non ci sono oggetti viene mostrato il messaggio ‘Non succede niente’ e l’applicazione rimane in attesa di un comando valido.
* *Chiudere il gioco* 🡪 al comando ‘termina’ l’applicazione si deve chiudere.

# 3. SPECIFICA ALGEBRICA DELLA STRUTTURA DATI “COMMAND LIST”

new\_command\_list() -> command\_list

new\_command\_list() -> C

PRE: \

POST: C = {}

add\_command(command) -> command\_list

add\_command(c) -> C'

PRE: C = {c1, c2, ..., cn}

POST: C' = {c1, c2, ..., cn, c}

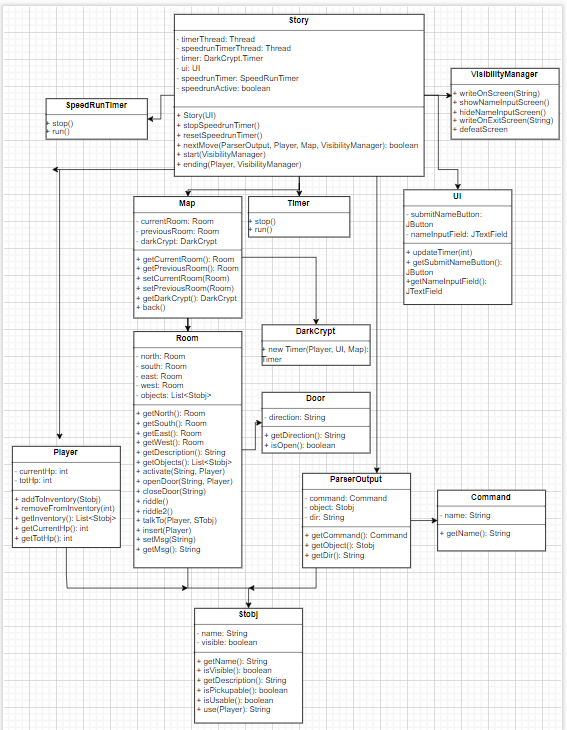
find\_command\_by\_alias(alias) -> command

find\_command\_by\_alias(a) -> c

PRE: C = {c1, c2, ..., cn} con n >= 0

POST: se esiste c in C tale che equal(a, alias(c)) = true allora c = alias(a)

# 4. DIAGRAMMA DELLE CLASSI

****

## Classe Story

La classe *Story* è il punto focale del gioco, gestendo il flusso principale e l'interazione con altre componenti. Essa contiene i seguenti campi e metodi:

* **Campi:**
  + *timerThread: Thread* - Thread per gestire il timer della stanza DarkCrypt.
  + *speedrunTimerThread: Thread* - Thread per gestire il timer della modalità speedrun.
  + *timer: DarkCrypt.Timer* - Istanza del timer specifico della stanza DarkCrypt.
  + *ui: UI* - Interfaccia utente del gioco.
  + *speedrunTimer: SpeedRunTimer* - Istanza del timer per la modalità speedrun.
  + *speedrunActive: boolean* - Indica se la modalità speedrun è attiva.
* **Metodi:**
  + *Story(UI ui)* - Costruttore della classe Story che inizializza i campi.
  + *stopSpeedrunTimer()* - Metodo per fermare il timer della speedrun.
  + *resetSpeedrunTimer()* - Metodo per resettare il timer della speedrun.
  + *nextMove(ParserOutput par, Player p, Map map, VisibilityManager vm): boolean* - Gestisce i comandi del giocatore e le interazioni nel gioco.
  + *start(VisibilityManager vm)* - Avvia la storia e il timer della speedrun.
  + *ending(Player p, VisibilityManager vm)* - Gestisce la sequenza di fine gioco.

## Classe SpeedRunTimer

La classe *SpeedRunTimer* gestisce il timer per la modalità speedrun del gioco. Contiene:

* **Metodi:**
  + *stop()* - Ferma il timer.
  + *run()* - Avvia il timer.

## Classe DarkCrypt

La classe *DarkCrypt* rappresenta una stanza specifica nel gioco. Contiene una classe interna Timer che rappresenta un timer specifico per la stanza DarkCrypt.

* **Metodi:**
  + *new Timer(Player, UI, Map): Timer* - Crea una nuova istanza di Timer.

## Classe Timer

La classe interna *Timer* di *DarkCrypt* gestisce un timer specifico per la stanza *DarkCrypt*.

* **Metodi:**
  + *stop()* - Ferma il timer.
  + *run()* - Avvia il timer.

## Classe UI

La classe *UI* gestisce l'interfaccia utente del gioco. Contiene:

* **Campi:**
  + *submitNameButton: JButton* - Bottone per inviare il nome del giocatore.
  + *nameInputField: JTextField* - Campo di testo per inserire il nome del giocatore.
* **Metodi:**
  + *updateTimer(int)* - Aggiorna il timer nell'interfaccia.
  + *getSubmitNameButton(): JButton* - Ritorna il bottone di invio nome.
  + *getNameInputField(): JTextField* - Ritorna il campo di testo per il nome.

## Classe VisibilityManager

La classe *VisibilityManager* gestisce la visualizzazione dei messaggi sullo schermo.

* **Metodi:**
  + *writeOnScreen(String)* - Scrive un messaggio sullo schermo.
  + *showNameInputScreen()* - Mostra la schermata di input del nome.
  + *hideNameInputScreen()* - Nasconde la schermata di input del nome.
  + *writeOnExitScreen(String)* - Scrive un messaggio sulla schermata di uscita.
  + *defeatScreen()* - Mostra la schermata di sconfitta.

## Classe ParserOutput

La classe *ParserOutput* contiene il risultato del parsing del comando inserito dal giocatore.

* **Campi:**
  + *command: Command* - Il comando riconosciuto.
  + *object: Stobj* - L'oggetto associato al comando.
  + *dir: String* - La direzione associata al comando (se applicabile).
* **Metodi:**
  + *getCommand(): Command* - Ritorna il comando.
  + *getObject(): Stobj* - Ritorna l'oggetto.
  + *getDir(): String* - Ritorna la direzione.

## Classe Player

La classe *Player* rappresenta il giocatore.

* **Campi:**
  + *currentHp: int* - Punti vita attuali del giocatore.
  + *totHp: int* - Punti vita totali del giocatore.
* **Metodi:**
  + *addToInventory(Stobj)* - Aggiunge un oggetto all'inventario.
  + *removeFromInventory(int)* - Rimuove un oggetto dall'inventario.
  + *getInventory(): List<Stobj>* - Ritorna l'inventario del giocatore.
  + *getCurrentHp(): int* - Ritorna i punti vita attuali.
  + *getTotHp(): int* - Ritorna i punti vita totali.

## Classe Map

La classe *Map* rappresenta la mappa del gioco.

* **Campi:**
  + *currentRoom: Room* - La stanza attuale.
  + *previousRoom: Room* - La stanza precedente.
  + *darkCrypt: DarkCrypt* - La stanza DarkCrypt.
* **Metodi:**
  + *getCurrentRoom(): Room* - Ritorna la stanza attuale.
  + *getPreviousRoom(): Room* - Ritorna la stanza precedente.
  + *setCurrentRoom(Room)* - Imposta la stanza attuale.
  + *setPreviousRoom(Room)* - Imposta la stanza precedente.
  + *getDarkCrypt(): DarkCrypt* - Ritorna la stanza DarkCrypt.
  + *back()* - Torna alla stanza precedente.

## Classe Room

La classe *Room* rappresenta una stanza nella mappa del gioco.

* **Campi:**
  + *north: Room* - Stanza a nord.
  + *south: Room* - Stanza a sud.
  + *east: Room* - Stanza a est.
  + *west: Room* - Stanza a ovest.
  + *objects: List<Stobj>* - Oggetti nella stanza.
  + *msg: String* - Messaggio della stanza.
* **Metodi:**
  + *getNorth(): Room* - Ritorna la stanza a nord.
  + *getSouth(): Room* - Ritorna la stanza a sud.
  + *getEast(): Room* - Ritorna la stanza a est.
  + *getWest(): Room* - Ritorna la stanza a ovest.
  + *getDescription(): String* - Ritorna la descrizione della stanza.
  + *getObjects(): List<Stobj>* - Ritorna gli oggetti nella stanza.
  + *activate(String, Player)* - Attiva un oggetto nella stanza.
  + *openDoor(String, Player)* - Apre una porta nella stanza.
  + *closeDoor(String)* - Chiude una porta nella stanza.
  + *riddle()* - Risolve un enigma nella stanza.
  + *riddle2()* - Risolve un secondo enigma nella stanza.
  + *talkTo(Player, Stobj)* - Parla con un oggetto nella stanza.
  + *insert(Player)* - Inserisce un oggetto nella stanza.
  + *setMsg(String)* - Imposta il messaggio della stanza.
  + *getMsg(): String* - Ritorna il messaggio della stanza.

## Classe Stobj

La classe *Stobj* rappresenta un oggetto nel gioco.

* **Campi:**
  + *name: String* - Nome dell'oggetto.
  + *visible: boolean* - Indica se l'oggetto è visibile.
* **Metodi:**
  + *getName(): String* - Ritorna il nome dell'oggetto.
  + *isVisible(): boolean* - Indica se l'oggetto è visibile.
  + *getDescription(): String* - Ritorna la descrizione dell'oggetto.
  + *isPickupable(): boolean* - Indica se l'oggetto può essere raccolto.
  + *isUsable(): boolean* - Indica se l'oggetto può essere usato.
  + *use(Player): String* - Usa l'oggetto e ritorna un risultato.

## Classe Command

La classe *Command* rappresenta un comando del gioco.

* **Campi:**
  + *name: String* - Nome del comando.
* **Metodi:**
  + *getName(): String* - Ritorna il nome del comando.

## Classe Door

La classe *Door* è una sottoclasse di *Stobj* e rappresenta una porta nel gioco.

* **Campi:**
  + *direction: String* - Direzione della porta.
* **Metodi:**
  + *getDirection(): String* - Ritorna la direzione della porta.
  + *isOpen(): boolean* - Indica se la porta è aperta.

# 5. DESCRIZIONE ARGOMENTI UTILIZZATI

## 5.1 Lamba expression

Una lambda expression è una funzione anonima (una funzione senza nome) che può essere utilizzata come un'espressione di sintesi per rappresentare istanze di interfacce funzionali (interfacce con un solo metodo astratto). Le lambda expressions sono state introdotte in Java 8 per fornire una sintassi più concisa e funzionale per definire e utilizzare funzioni.

*Perché conviene usarle?*

* **Sinteticità**: le lambda expressions permettono di scrivere codice più conciso rispetto alle classi anonime. Questo riduce il boilerplate code, rendendo il codice più leggibile e mantenibile.
* **Migliore leggibilità**: la sintassi compatta delle lambda expressions rende il codice più leggibile, soprattutto quando si lavora con API che utilizzano ampiamente interfacce funzionali, come le Stream API.
* **Programmazione funzionale**: le lambda expressions facilitano lo stile di programmazione funzionale, dove le funzioni possono essere passate come argomenti, restituite come valori e assegnate a variabili.
* **Parallelismo**: le lambda expressions, combinate con le Stream API, rendono il parallelismo più semplice da implementare e gestire.

*Dove vengono utilizzate?*

* Nelle *actionlistener*.
* Nei cicli *while* all’interno del *ParserOutput*, attraverso la chiamata dei metodi *checkforCommand* e *checkforObject*.
* In altri cicli come i *for* e *foreach* utilizzati per controllare il contenuto dell’inventario del giocatore

## 5.2 Thread

Un thread è la più piccola unità di elaborazione che può essere eseguita indipendentemente. Permettono l'esecuzione parallela di codice, consentendo di eseguire più compiti simultaneamente all'interno di un programma.

*Perché conviene usarli?*

* **Concorrency**: i thread consentono a un programma di eseguire più attività simultaneamente, migliorando l'efficienza e l'utilizzo delle risorse di sistema. Questo è particolarmente utile per operazioni I/O, come lettura/scrittura su file o comunicazione di rete, che possono essere eseguite in parallelo con altre operazioni.
* **Parallelismo**: in ambienti con CPU multi-core, i thread permettono di suddividere un lavoro in parti più piccole e di eseguire queste parti in parallelo, migliorando le prestazioni e riducendo i tempi di esecuzione.
* **Miglior utilizzo delle risorse**: i thread permettono di sfruttare meglio le risorse del sistema, mantenendo la CPU occupata mentre si attendono operazioni I/O e altre risorse lente.
* **Reattività dell’interfaccia utente**: nei programmi con interfacce grafiche (GUI), i thread possono essere utilizzati per eseguire operazioni lunghe in background, mantenendo l'interfaccia utente reattiva. Questo evita che l'interfaccia si blocchi mentre vengono eseguite operazioni intensive.
* **Semplificazione di problemi complessi**: alcuni problemi sono naturalmente adatti ad essere risolti tramite threading. Ad esempio, nei server web, ogni richiesta client può essere gestita da un thread separato, semplificando la logica del server.

*Dove vengono utilizzati?*

In due tipi di timer diversi all’interno del gioco. In una sezione, i thread usati sono eseguiti anche allo stesso momento.

* **Timer speed run:** un timer che tiene conto della durata della ‘run’ del giocatore. Questo thread fa partire un timer solo quando un giocatore inizia una ‘Nuova Partita’. Esso viene interrotto all’uscita del giocatore dal gioco (per qualsiasi motivo), ed i dati relativi ad esso vengono salvati in un DB solo in caso il giocatore abbia finito il gioco nella stessa sessione in cui ha iniziato una nuova partita.
* **Timer Countdown:** un countdown che parte tramite l’uso di un thread e conta da 30 a 0. Questo timer è presente in una sola stanza del gioco e viene eseguito in contemporanea al thread che tiene conto del tempo necessario al completamento del gioco. Quando il timer raggiunge lo 0 il giocatore viene espulso dalla stanza alla sua prossima azione.

## 5.3 Swing

Swing è una parte del pacchetto Java Foundation Classes (JFC) che fornisce un'API per la creazione di interfacce grafiche utente (GUI) in Java. Fa parte della piattaforma Java Standard Edition (Java SE).

*Perché conviene usarle?*

* **Portabilità**: poiché Swing è scritto interamente in Java, le applicazioni Swing sono indipendenti dalla piattaforma. Questo significa che un'applicazione Swing funzionerà allo stesso modo su Windows, macOS, Linux e altre piattaforme che supportano Java.
* **Componenti avanzati**: swing offre componenti avanzati e ricchi di funzionalità che non sono disponibili in AWT. Ad esempio, Swing fornisce tabelle (JTable), alberi (JTree) e aree di testo avanzate (JTextPane).
* **Flessibilità e personalizzazione**: swing consente un alto grado di personalizzazione dei componenti GUI. Gli sviluppatori possono creare interfacce utente sofisticate e personalizzate utilizzando le capacità di rendering e disegno di Swing.
* **Supporto per threading**: swing è progettato per funzionare bene con il modello di threading di Java. Le operazioni che modificano l'interfaccia utente devono essere eseguite sul thread dell'evento di dispatching (Event Dispatch Thread), mentre le operazioni di lunga durata possono essere eseguite su thread separati per mantenere l'interfaccia utente reattiva.
* **Strumenti di sviluppo**: ci sono molti strumenti di sviluppo e ambienti integrati (IDE) che supportano Swing, come IntelliJ IDEA, Eclipse e NetBeans. Questi strumenti forniscono supporto per il design visivo delle interfacce utente, facilitando lo sviluppo di applicazioni Swing.

*Dove viene utilizzata?*

Ovunque nel gioco ‘Il Castello Enigmatico’. L’interfaccia di gioco è creata totalmente tramite l’uso di SWING, ciò comprende tutte le schermate, le stampe a schermo e le varie interazioni che si possono avere durante il corso di una sessione del gioco.

## 5.4 File

Un file è una raccolta di dati memorizzati in un dispositivo di archiviazione, come un disco rigido o una memoria flash. I file possono contenere qualsiasi tipo di informazione, da testo semplice a dati binari complessi. In Java, la gestione dei file è parte integrante dell'API standard, che fornisce strumenti per leggere, scrivere e manipolare vari tipi di file.

*Perché Usare i File?*

* ***Persistenza dei Dati***: I file permettono di memorizzare dati in modo permanente, anche dopo che il programma termina. Questo è essenziale per applicazioni che devono salvare configurazioni, log, dati degli utenti, ecc.
* ***Scambio di Dati:*** I file sono un metodo comune per scambiare dati tra diversi programmi e sistemi. Ad esempio, CSV, JSON e XML sono formati di file utilizzati per l'interoperabilità tra diverse applicazioni.
* ***Backup e Ripristino:*** I file sono spesso utilizzati per creare backup dei dati importanti e per ripristinare il sistema in caso di perdita di dati o malfunzionamenti.
* ***Registrazione e Monitoraggio:*** Molte applicazioni utilizzano file di log per registrare eventi, errori e altre informazioni di runtime, che possono essere utili per il debugging e il monitoraggio delle prestazioni.

*Dove sono utilizzati?*

Il sistema di salvataggio di ‘Il Castello Enigmatico’ si basa totalmente sull’utilizzo di file per il salvataggio di dati della partita in corso. I dati salvati sono relativi alle informazioni del giocatore e della mappa. Essi vengono poi caricati una volta che il giocatore decide di ‘Continuare’ una vecchia partita. Solo un salvataggio può essere presente in un determinato momento e viene sovrascritto ogni talvolta che il giocatore decide di salvare i dati.

5.5 REST  
  
REST sta per ‘Reprentational State Transfer”. In Java, le funzioni REST vengono utilizzate per servizi di rete per comunicazioni tra client e server basate sul protocollo HTTP. Ci sono vari framework e librerie per l’uso di API RESTful su Java, ed in questo caso si fa utilizzo di Spring Boot con Spring web, che offre totale supporto per le operazioni REST tramite annotazioni come ‘@RestController’, ‘@GetMapping’ etc…

*Perché Usare funzioni REST?*

* ***Semplicità e Standardizzazione:*** REST utilizza il protocollo HTTP, che è ampiamente conosciuto e supportato. Le operazioni CRUD (Create, Read, Update, Delete) possono essere facilmente mappate ai metodi HTTP standard: POST, GET, PUT, DELETE. Questo rende le API RESTful semplici da capire e da implementare.
* ***Cacheabilità:*** Le risposte delle API RESTful possono essere cacheate, migliorando le prestazioni e riducendo il carico sul server. Utilizzando intestazioni HTTP appropriate, è possibile controllare quali risposte possono essere cacheate e per quanto tempo.
* ***Ampio Supporto di Strumenti e Framework:*** Esiste un vasto ecosistema di strumenti, librerie e framework per lo sviluppo, il test, la documentazione e la gestione delle API RESTful. Questo supporto rende più facile e veloce il processo di sviluppo.

*Dove sono utilizzate?*L’intenzione è quella di utilizzare una funzione REST per un’operazione di tipo CRUD (Create, Read, Update, Delete), in questo caso, un’operazione Delete. Ogni qualvolta venga chiamato il metodo per mostrare a schermo i contenuti della classifica conservati all’interno del database, un’operazione ‘RESTful’ verrebbe eseguita sul DB per eliminare tutti i dati superflui, ovvero qualsiasi dato appartenente alla classifica che andrebbe a posizionare il record di un giocatore oltre il 10imo posto, con una classifica ordinata in base al tempo. Questo permetterebbe di evitare il ‘bloating’ dei dati contenuti nel database del gioco.