SCANNER-DEL-DESTINO

GRUPPO DI LAVORO:

Gabriele Specchio: g.specchio5@studenti.uniba.it
Marsico Domenico: d.marsico4@studenti.uniba.it
Pascal Marius Velondaza: m.velondaza@studenti.uniba.it

Github Repository: https://github.com/MAPPROGETTO/Scanner-del-Destino.git

Sommario

INTRODUZIONE

Contesto

Il progetto sviluppato consiste in un gioco di avventura testuale con supporto grafico, ispirato ai classici giochi testuali ma con una struttura moderna che integra gestione della mappa, inventario e interazioni con eventi narrativi.
L'utente interagisce attraverso comandi testuali o scelte fornite dall'interfaccia, guidando il protagonista tra ambienti, oggetti e personaggi.

Obbiettivi

L'obiettivo del caso di studio è stato:

- progettare e realizzare un sistema software che unisca la narrativa interattiva con la gestione di strutture dati personalizzate (mappa, stanze, inventario);
- applicare i concetti teorici e pratici trattati nel corso (file, JDBC, lambda expression, thread, ...);
- sviluppare un codice modulare e documentato, corredato da un diagramma UML delle classi principali e da una specifica algebrica di una delle strutture dati utilizzate.

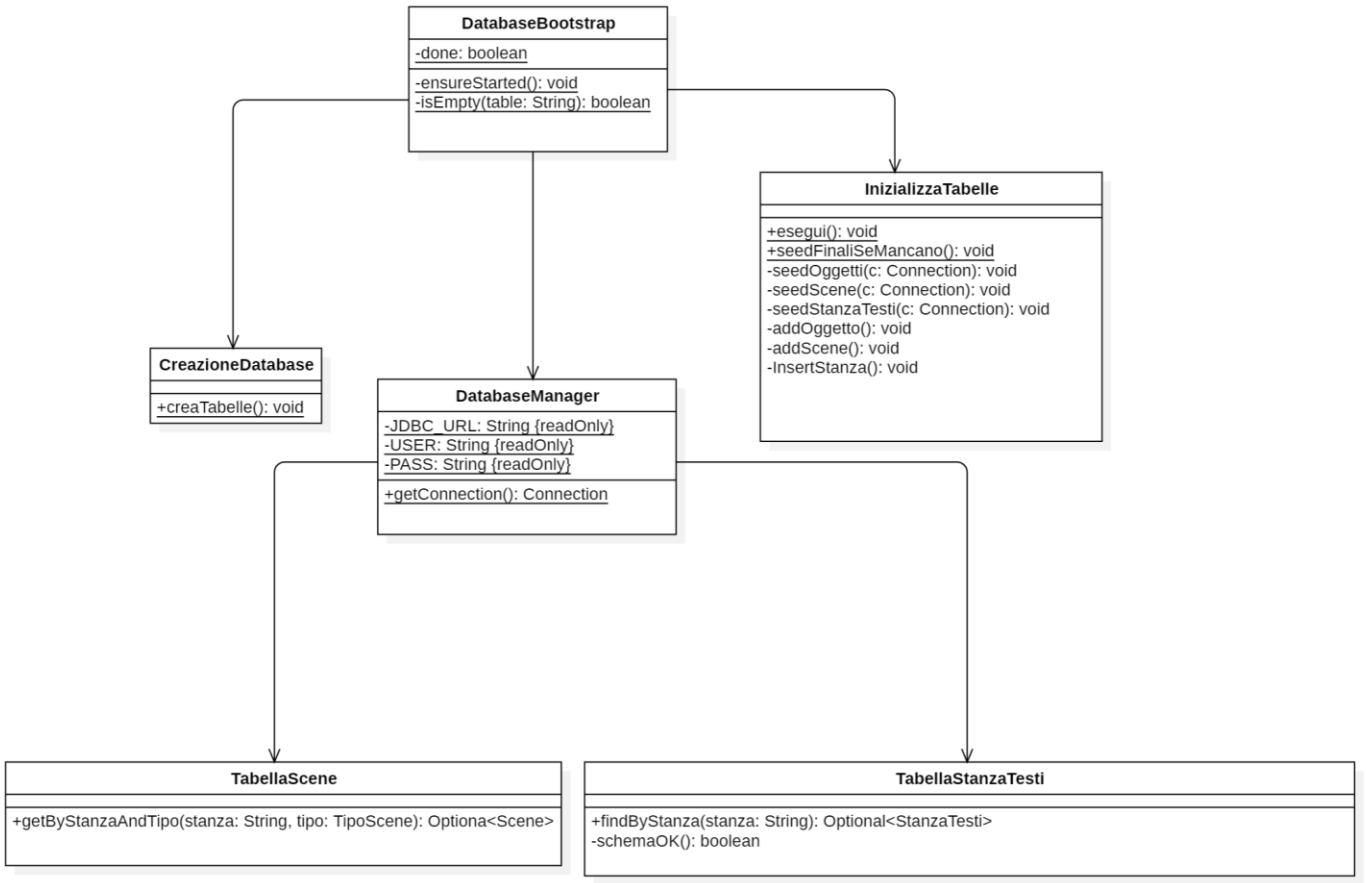
Trama

Lewis è un brillante e giovanissimo inventore, orfano di circa 12 anni e vive in un orfanotrofio in stanza con un vero talento sportivo di nome Michael, un po' più piccolo di lui, ma molto fragile e insicuro di sé.
Lewis sta creando uno scanner mnemonico per poter trovare, scavando nei ricordi, il giorno in cui sua madre l'abbandonò per poterla riconoscere e ritrovarla. Il giovane troverà alcuni bug nel suo congegno facendo fallire la sua presentazione alla mostra della scienza scolastica dove "incontrerà" l'Uomo con la bombetta", un uomo arrivato dal futuro con una macchina del tempo spaziale, per rubargli il congegno.

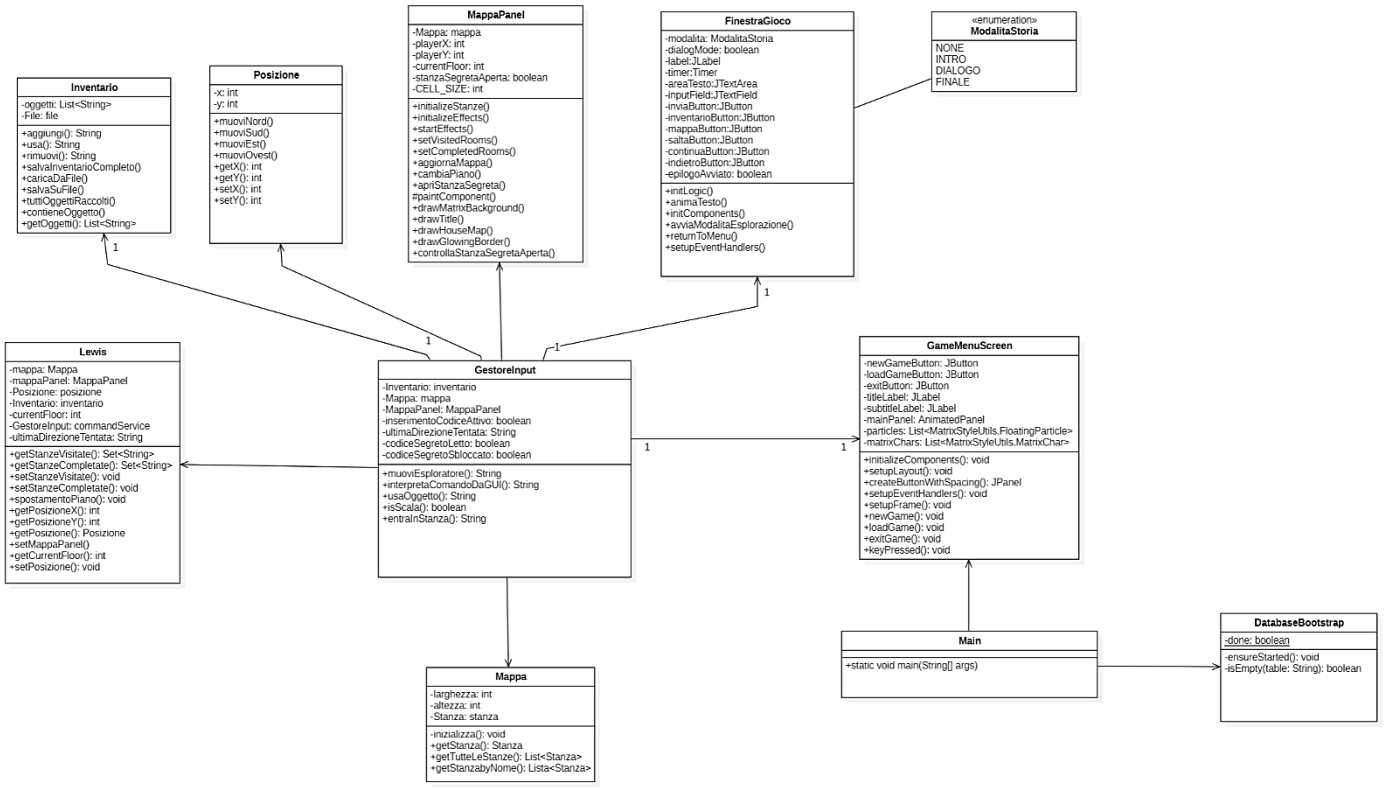
Diagramma delle classi

Il diagramma delle classi rappresenta la struttura principale del sistema e le relazioni tra le classi che compongono il gioco.
Sono state incluse soltanto le classi più significative, ossia quelle che incarnano la logica centrale dell'avventura testuale.

- **DatabaseManager:**



- **GestoreInput**: classe che interpreta i comandi inseriti dall'utente e coordina le azioni conseguenti.



Specifica algebrica

Per rendere più chiara la logica del progetto, una parte della documentazione è dedicata alla specifica algebrica di una struttura dati utilizzata.
Nel nostro caso la scelta è caduta sulla **Lista**, perché rappresenta una delle strutture dati più fondamentali: consente di memorizzare una sequenza ordinata di elementi e di gestirli tramite operazioni come inserimento, rimozione e ricerca.

Specifica sintattica:
Tipi: list, element, integer, boolean
Operazione:

- crea una lista vuota.
  ArrayList () → list
- aggiunge element nella lista in posizione index.
  add (list,integer, element) → list
- rimuove tutti gli elementi dalla lista.
  Clear(list) → list
- restituisce l'elemento in posizione index.
  get (list, integer) → element
- restituisce true se la lista è vuota, false altrimenti.
  isEmpty(list) → Boolean
- rimuove e restituisce l'elemento in posizione index.
  remove(list,integer) → element
- restituisce il numero di elementi nella lista.
  size(list) → integer

Specifica semantica:
declare: l: list, e: element, i: interger

osservazioni	*Costruttori di l*	
	ArrayList ()	**add (l, I, e)**
get (l,i)	error	e
clear (l)	isEmpty(l) = true	size(l) = 0
isEmpty (l)	**True**	**False**
remove (l,i)	Error	**If** get(l,i) == e **then** l **else** add (l, I, remove(l,i))
size(l)	0	**If** add(l, i,e) **then** size(l) **else** size(l) +1

Specifica di restrizione:
get(arraylist(), i) = error
remove(arraylist(), i) = error

Descrizione dell'applicazione degli argomenti del corso

File

Nel progetto i file sono utilizzati per la gestione del salvataggio e caricamento delle partite.
L'intero stato del gioco (inclusi progressi, oggetti raccolti, stanze visitate e condizioni logiche) viene serializzato e memorizzato all'interno di file nella cartella dedicata salvataggi/.
Ogni salvataggio corrisponde a uno slot (slot1.dat, slot2.dat, slot3.dat), che può essere sovrascritto o eliminato.
L'operazione sfrutta le classi di java.io (ObjectOutputStream e ObjectInputStream) per scrivere e leggere oggetti Java direttamente su file.
Questo permette la persistenza dei dati anche dopo la chiusura del gioco e mantiene isolata la gestione dei file in una classe dedicata (ad es. GameSaver).

- **Gestione dei file di salvataggio:**

```
private static final String CARTELLA_SALVATAGGIO = "salvataggi/";
1 usage
private static final int NUM_SLOT = 3;

static {
    File dir = new File(CARTELLA_SALVATAGGIO);
    if (!dir.exists()) {
        if (!dir.mkdirs()) {
            System.err.println("> Errore: impossibile creare la cartella di salvataggio.");
        }
    }
}
```

- **salvataggio della partita:**

```
public static boolean salvaPartita(StatoGioco stato, int slotForzato) {
    if (stato == null) {
        System.err.println("> Errore: stato di gioco nullo, impossibile salvare.");
        return false;
    }
    int slot = slotForzato > 0 ? slotForzato : getPrimoSlotDisponibile( cercaSlotVuoti: true);
    if (slot == -1) {
        int scelta = JOptionPane.showConfirmDialog( parentComponent: null,
                message: "Tutti gli slot sono pieni. Vuoi sovrascrivere lo slot 1?",
                title: "Slot pieni",
                JOptionPane.YES_NO_OPTION);
        if (scelta == JOptionPane.YES_OPTION) {
            slot = 1;
        } else {
            System.err.println("> Salvataggio annullato dall'utente.");
            return false;
        }
    }
    String nomeFile = CARTELLA_SALVATAGGIO + "slot" + slot + ".dat";
    try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(nomeFile))) {
        oos.writeObject(stato);
        System.out.println("> Partita salvata su: " + nomeFile);
        return true;
    } catch (IOException e) {
        System.err.println("> Errore durante il salvataggio: " + e.getMessage());
        e.printStackTrace();
        return false;
    }
}
```

JDBC

Il mondo di gioco è descritto in un database **H2 embedded**, inizializzato automaticamente all'avvio. DatabaseManager centralizza la connessione (DriverManager), mentre DatabaseBootstrap garantisce il **bootstrap**: crea lo schema con CreazioneDatabase e popola i contenuti con InizializzaTabelle, inclusi i finali che vengono "top-uppati" se mancanti. I dati chiave sono in tre tabelle: **scene** (testi narrativi) e **stanza_testi** (descrizioni, osservazioni e stato di completamento per singola stanza). L'accesso applicativo è incapsulato in DAO mirati (TabellaScene, TabellaStanzaTesti) e in piccoli adapter narrativi (ScenesDb, NarrationDbAdapter) che espongono metodi ad alto livello ("dammi il testo d'ingresso per questa stanza", "dammi l'evento per questo oggetto"). Questa stratificazione consente di **evolvere i contenuti** senza toccare la logica, riducendo i null con Optional e mantenendo le query locali, piccole e chiare.

- **Connessione al Database**

```java
public final class DatabaseManager {
    1 usage
    private static final String JDBC_URL =
            "jdbc:h2:file:./scannerdb;" +
                    "MODE=MySQL;" +
                    "DATABASE_TO_LOWER=TRUE;" +
                    "CASE_INSENSITIVE_IDENTIFIERS=TRUE;" +
                    "DB_CLOSE_DELAY=-1;" +      // non chiudere l'engine a fine connessione
                    "PAGE_SIZE=4096;" +
                    "CACHE_SIZE=65536;" +
                    "TRACE_LEVEL_FILE=0";
    1 usage
    private static final String USER = "sa";
    1 usage
    private static final String PASS = "";

    no usages   GabrieleSpecchio <gabrielespecchio03gmail.com>
    private DatabaseManager() {}

    /**
     * Ottiene una connessione al database H2.
     * @return Connessione al database
     * @throws SQLException in caso di errori di connessione
     */
    7 usages   GabrieleSpecchio <gabrielespecchio03gmail.com>
    public static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(JDBC_URL, USER, PASS);
    }
}
```

- **Creazione del database:**

Per inizializzare il database H2 abbiamo definito un metodo dedicato alla **creazione delle tabelle principali** utilizzate dal gioco. Questo metodo, eseguito all'avvio, garantisce che le strutture necessarie siano disponibili senza doverle creare manualmente ogni volta a mano.
In particolare, vengono create tre tabelle:

- **oggetti** → contiene tutti gli oggetti presenti nell'avventura, con attributi come codice, nome, descrizione e proprietà (raccoglibile, usabile).
- **scene** → rappresenta le diverse scene del gioco, collegate a stanze o eventi narrativi, con titolo e testo da mostrare al giocatore.
- **stanza_testi** → descrive le stanze in dettaglio (descrizione, osservazioni, stato completato), permettendo di gestire più varianti della stessa scena.

Questo approccio rende il progetto **scalabile e flessibile**, perché consente di modificare la storia o aggiungere contenuti agendo direttamente sul database, senza cambiare il codice Java.

```java
public final class CreazioneDatabase {
    no usages  GabrieleSpecchio <gabrielespecchio03gmail.com>
    private CreazioneDatabase() {}
    /**
     * Crea le tabelle del database se non esistono già.
     * Le tabelle create sono: oggetti, scene, stanza_testi.
     * @throws SQLException in caso di errori durante la creazione delle tabelle
     */
    1 usage  GabrieleSpecchio <gabrielespecchio03gmail.com> *
    public static void creaTabelle() throws SQLException {
        try (Connection c = DatabaseManager.getConnection(); Statement st = c.createStatement()) {
            // -------- OGGETTI --------
            st.execute( sql: """
                CREATE TABLE IF NOT EXISTS oggetti (
                  id BIGINT AUTO_INCREMENT PRIMARY KEY,
                  codice VARCHAR(64) UNIQUE NOT NULL,
                  nome VARCHAR(128) NOT NULL,
                  descrizione VARCHAR(1024),
                  raccoglibile BOOLEAN NOT NULL DEFAULT TRUE,
                  usabile BOOLEAN NOT NULL DEFAULT FALSE
                )
            """);
            // -------- SCENE --------
            st.execute( sql: """
                CREATE TABLE IF NOT EXISTS scene (
                  id BIGINT AUTO_INCREMENT PRIMARY KEY,
                  stanza VARCHAR(128) NOT NULL,
                  tipo VARCHAR(32) NOT NULL,
                  titolo VARCHAR(128),
                  testo CLOB NOT NULL
                )
            """);
            // Indici
            st.execute( sql: "CREATE INDEX IF NOT EXISTS idx_scene_stanza_tipo ON scene(stanza, tipo)");
            st.execute( sql: "CREATE INDEX IF NOT EXISTS idx_scene_titolo ON scene(titolo)");

            // Dedup prima di unique: elimina duplicati della tripla mantenendo l'id minore
            st.execute( sql: """
                DELETE FROM scene s
                WHERE EXISTS (
                  SELECT 1 FROM scene d
                  WHERE d.stanza = s.stanza
                    AND d.tipo = s.tipo
                    AND COALESCE(d.titolo,'') = COALESCE(s.titolo,'')
                    AND d.id < s.id
                )
            """);
            // Vincolo UNIQUE sulla tripla (stanza,tipo,titolo) per evitare doppi
            st.execute( sql: """
                ALTER TABLE scene
                ADD CONSTRAINT IF NOT EXISTS ux_scene_tripla
                UNIQUE(stanza, tipo, titolo)
            """);
            // -------- STANZA_TESTI --------
            st.execute( sql: """
                CREATE TABLE IF NOT EXISTS stanza_testi (
                  id BIGINT AUTO_INCREMENT PRIMARY KEY,
                  nome_stanza VARCHAR(128) NOT NULL,
                  descrizione CLOB,
                  osserva CLOB,
                  osserva_aggiornata CLOB,
                  completato BOOLEAN DEFAULT FALSE,
                  CONSTRAINT ux_stanza UNIQUE (nome_stanza)
                )
            """);
        }
    }
}
```

- **Bootstrap:**

La classe DatabaseBootstrap ha il compito di inizializzare il database all'avvio del gioco, questa classe è fondamentale perché garantisce che, al primo avvio, il gioco abbia sempre a disposizione un database pronto e popolato con i dati minimi necessari alla storia.

Le sue responsabilità principali sono:

- Creare le tabelle tramite CreazioneDatabase.creaTabelle().
- Verificare se la tabella delle scene è vuota (isEmpty("scene")).
- Se vuota, popolare il database con i dati iniziali tramite InizializzaTabelle.esegui().
- Se non vuota, aggiungere eventuali scene finali mancanti con seedFinaliSeMancano().

```java
public final class DatabaseBootstrap {
    2 usages
    private static volatile boolean done = false;
    no usages  GabrieleSpecchio <gabrielespecchio03gmail.com>
    private DatabaseBootstrap() {}

    Assicura che il database sia inizializzato. Crea le tabelle se non esistono e popola i dati iniziali se la
    tabella 'scene' è vuota. Questo metodo è thread-safe e viene eseguito solo una volta.

    3 usages  GabrieleSpecchio <gabrielespecchio03gmail.com>
    public static synchronized void ensureStarted() {
        if (done) return;
        try {
            CreazioneDatabase.creaTabelle();

            // Se la tabella 'scene' è vuota → fai seed completo (il tuo InizializzaTabelle ora TRUNCATE+INSERT)
            if (isEmpty( table: "scene")) {
                InizializzaTabelle.esegui();
            } else {
                // DB già popolato → fai solo top-up dei finali in modalità "insert-if-missing"
                InizializzaTabelle.seedFinaliSeMancano();
            }

            done = true;
        } catch (Exception e) {
            throw new RuntimeException("Impossibile inizializzare il DB", e);
        }
    }
    /** Controlla se una tabella è vuota. ...*/
    1 usage  GabrieleSpecchio <gabrielespecchio03gmail.com>
    private static boolean isEmpty(String table) {
        String sql = "SELECT COUNT(*) FROM " + table;
        try (var c = DatabaseManager.getConnection();
             var st = c.createStatement();
             var rs = st.executeQuery(sql)) {
            rs.next();
            return rs.getInt( columnIndex: 1) == 0;
        } catch (Exception e) {
            // Se fallisce (es. Tabella non esiste ancora) trattiamo come vuota
            return true;
        }
    }
}
```

Lambda expression

Le lambda vengono utilizzate per semplificare il codice e renderlo più leggibile, soprattutto quando si devono passare azioni come parametri (callback, eventi o azioni temporizzate).
Nel progetto compaiono principalmente in:

Gestione dell'interfaccia Swing → ad esempio negli ActionListener di pulsanti (btnNuovaPartita, btnCarica, ecc.), sostituendo le classi anonime.
Sequenze narrative temporizzate → nei Timer Swing per gestire i ritardi nella scrittura del testo o nelle animazioni.
In questo modo il codice risulta più compatto, migliorando la leggibilità nelle sequenze di gioco.

- **Class Main:**

Nel metodo main, ad esempio, usiamo una lambda come argomento di SwingUtilities.invokeLater(), che

```
public static void main(String[] args) {
    DatabaseBootstrap.ensureStarted(); // <- crea tabelle + popola se vuoto
    SwingUtilities.invokeLater(() -> {
        // Nessun salvataggio → mostra il menu principale
        new GameMenuScreen().setVisible(true);
    });
}
```

richiede un'implementazione di Runnable.
Invece di scrivere una classe anonima verbosa, con la lambda otteniamo un codice più compatto e leggibile, che mostra direttamente l'azione da compiere (mostrare il menu principale della GUI).

Thread

Il progetto adotta una disciplina chiara: niente lavoro pesante sull'EDT e animazioni gestite in modo sicuro. Per le operazioni di background si usa SwingWorker: negli splash (SplashScreen, TimePortalSplashScreen) un worker inizializza H2 e risorse pubblicando messaggi e avanzamento su JProgressBar; in GameMenuScreen lo stesso approccio carica gli slot senza bloccare la finestra. Per il ritmo visivo e l'effetto di battitura del testo si impiega javax.swing.Timer, che scatta sull'EDT e aggiorna in modo dolce componenti e repaint (testi, particelle, "gocce" Matrix). L'unico Thread.sleep rimane confinato dentro il worker per scandire lo stato di avanzamento (percezione del caricamento), mai nella UI. L'avvio della GUI passa da SwingUtilities.invokeLater (in engine.Main), rispettando il modello di threading di Swing. Risultato: interfaccia sempre reattiva, assenza di race condition, e codice di concorrenza limitato agli scenari giusti.

- **Class SplashScreen:**

- nel metodo closeAndLaunch() utilizziamo SwingUtilities.invokeLater() per lanciare la nuova finestra del gioco.
  Questa chiamata sfrutta il concetto di thread in Swing: la creazione della GUI non viene eseguita immediatamente, ma delegata al thread dell'interfaccia grafica (EDT).

```
private void closeAndLaunch() {
    if (!isVisible()) {
        // se non era stata resa visibile qui, chiudi comunque risorse e lancia gioco
        stopAndDispose();
        SwingUtilities.invokeLater(FinestraGioco::new);
        return;
    }
    stopAndDispose();
    SwingUtilities.invokeLater(FinestraGioco::new);
}
```

- Il metodo step(...) utilizza Thread.sleep(350) per introdurre un piccolo ritardo nell'aggiornamento della barra di progresso o dei messaggi di caricamento.

```
private void step(int progress, int messageIdx) throws InterruptedException {
    // piccolo ritardo per percepire l'avanzamento (facoltativo)
    Thread.sleep( millis: 350); // regola a piacere
    setProgress(Math.min(100, Math.max(0, progress)));
    if (messageIdx >= 0 && messageIdx < loadingMessages.length) {
        publish(loadingMessages[messageIdx]);
    }
}
```

Java Swing

L'interfaccia grafica del progetto è realizzata in Java Swing e costituisce la cornice narrativa dell'avventura. Si compone principalmente di due schermate:

- **GameMenuScreen**: La prima è l'interfaccia iniziale (GameMenuScreen), una finestra essenziale che presenta il titolo e i tre ingressi fondamentali all'esperienza: Nuova Partita per avviare una run pulita, Carica Partita per riprendere dai salvataggi, ed Esci per chiudere l'applicazione.

```
private void initializeComponents() {
    particles = MatrixStyleUtils.createFloatingParticles( count: 20,  width: 1000,  height: 800);
    matrixChars = MatrixStyleUtils.createMatrixChars( count: 30,  width: 1000,  height: 800);

    titleLabel = new JLabel( text: "GAME MENU", SwingConstants.CENTER);
    titleLabel.setFont(MatrixStyleUtils.COURIER_TITLE);
    titleLabel.setForeground(MatrixStyleUtils.MATRIX_GREEN);

    subtitleLabel = new JLabel( text: "Seleziona un'opzione per continuare", SwingConstants.CENTER);
    subtitleLabel.setFont(MatrixStyleUtils.COURIER_FONT);
    subtitleLabel.setForeground(MatrixStyleUtils.MATRIX_GREEN_TRANSPARENT);

    newGameButton = MatrixStyleUtils.createStyledButton( text: "NUOVA PARTITA", MatrixStyleUtils.MATRIX_GREEN);
    loadGameButton = MatrixStyleUtils.createStyledButton( text: "CARICA PARTITA", MatrixStyleUtils.MATRIX_GREEN_TRANSPARENT);
    exitButton = MatrixStyleUtils.createStyledButton( text: "ESCI", MatrixStyleUtils.DANGER_COLOR);

    mainPanel = (AnimatedPanel) paintComponent(g) -> {
            super.paintComponent(g);
            Graphics2D g2d = (Graphics2D) g.create();

            MatrixStyleUtils.applyAlternativeGradient(g2d, getWidth(), getHeight());
            MatrixStyleUtils.drawFloatingParticles(g2d, particles);
            MatrixStyleUtils.drawMatrixChars(g2d, matrixChars);
            MatrixStyleUtils.drawMatrixEffect(g2d, getWidth(), getHeight());

            g2d.dispose();
    };
```

- **FinestraGioco**: La seconda è l'interfaccia di gioco (FinestraGioco), cuore dell'interazione: qui la narrazione scorre all'interno di una JTextArea dedicata, mentre i comandi del giocatore vengono inseriti in un JTextField collegato al parser. Due pulsanti **"Salta"** e **"Continua"** consentono di governare il ritmo del testo (avanzando la scena) senza perdere il filo della storia.

```
public FinestraGioco() {
    label = new JLabel( text: "");
    add(label);
    setTitle("Scanner del Destino");
    setSize( width: 1000,  height: 800);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLocationRelativeTo(null);

    inputGest = new GestoreInputGUI();

    initComponents();
    initLogic();
    setupEventHandlers();

    setVisible(true);
}
```

Accanto alla parte testuale trova spazio un pannello Mappa, che visualizza in modo sintetico la posizione corrente e le stanze esplorate, aggiornandosi man mano che il personaggio si sposta.