INFO0027: Game Book Editor Project

> **Group size**: 2
> **Deadline**: 2025-05-02

1 Situation

Your team leader enters the room with a request: *"The game makers need a tool to edit their interactive adventures."* After a team discussion, you have agreed on a format that is both simple and feature-complete. While a modern Web application was considered, the team has decided on a more minimalist approach. Your task is to develop a text-based application that run in the terminal. Additionally, you must create **UML class diagrams** to illustrate your software architecture and apply **software design patterns** appropriately in your implementation.

1.1 The File Format

The project uses a custom text-based format, `.mini.twee`, for defining interactive stories. This format is compatible with existing tools like Twinery. It is a simplified version of the Chapbook format[1].

Valid `.mini.twee` files adhere to the following informal grammar[2]:

```
story          ::= storyTitle storyData node+
storyTitle     ::= ":: StoryTitle" NEWLINE TEXT NEWLINE+
storyData      ::= ":: StoryData" NEWLINE json NEWLINE+
node           ::= "::" INT NEWLINE textLine* action* NEWLINE
action         ::= "[[" linkText "->" INT "]]" NEWLINE
linkText       ::= TEXT
textLine       ::= TEXT NEWLINE
json           ::= /Valid JSON object with start key/
INT            ::= DIGIT+
TEXT           ::= /Any character except '[' or ']' or 'NEWLINE'/
NEWLINE        ::= <newline>
DIGIT          ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

[1]https://klembot.github.io/chapbook/
[2]The grammar roughly follows the W3C EBNF (Extended Backus-Naur Form) syntax.

An example would give:

File format example: example.mini.twee

```
:: StoryTitle
Sart-Tilman Quick Trip

:: StoryData
{ "start": 1 }

:: 1
Your course has just ended at the Amphithéâtre de l'Europe (B4).
You can either head to the university restaurant or step into the woods.

[[Head to the cafetaria->2]]
[[Go into the woods of Colonster->3]]

:: 2
You are at the restaurant.
Feeling full after your meal, you might head back towards the B4, or perhaps
take a digestive walk?

[[Go to B4->4]]
[[Take a walk in the woods->3]]

:: 3
You've entered the woods de Colonster.
You can hear the gentle sound of the stream of the Blanc Gravier.

[[Follow the stream->5]]

:: 4
Back at the B4, it's time for your next activity.


:: 5
You follow the peaceful Rau du Blanc Gravier through the woods and arrive at
the old mill of Colonster.
```

2 Requirements

Your app needs to implement the following set of functional requirements. The command syntax is normative, while the examples are only illustrative. Handle ambiguities by making reasonable assumptions or seeking clarification during Q&A sessions or on the eCampus discussion forums.

1. **Command-Line Interface**: The application presents a command-line prompt (`>>`) and waits for user input from standard input (`stdin`).

   ```
   Non-normative example of the user interface
   Story Title: The river
   Start Node: 1

   Node 1
   You are walking near a river and you see a wooden bridge to pass to the
   other side of the river.
   You can also continue near the river.

   1) Pass the bridge (go to node 3)
   2) Continue near the river (go to node 2)

   >>
   ```

2. **Current Node View**: The interface must display information about the currently being edited story and node, including:
   - Story Title
   - Start Node ID
   - Current Node ID
   - Node Text
   - Actions: Displaying each action with its label and target node ID.

3. **`history` Command**: Upon entering the history command, the view switches to display a list of past actions. The list should include an index for each action, with 0 representing the current state.

   ```
   Non-normative example of history view
   History:
   0) Current
   1) Edit text of node 1
   2) Add action to node 1: "Pass the bridge" go to node 3

   >>
   ```

4. **Undo/Redo Functionality**: Implement `undo` and `redo` commands to revert and reapply operations. The application should support undoing and redoing operations performed since the file was opened.

5. **`revert <history index>`**: This command is available only when the application is in the history view. It allows reverting the current state to the state corresponding to the specified `<history index>`. `0` is the current state, `1` the previous state, etc. `revert 2` would be equivalent to calling twice `undo`.

6. **`open <file path>`**: Loads a file from the given `<file path>` in the `mini.twee` format. This switches to node view of the starting node.

7. **`save <file path>`**: Save the current story in the .mini.twee format to the specified `<file path>`.

8. **`save`**: Save the current story back to the file it was originally opened from.

9. **`set text <new text>`**: Edits the text content of the currently visible node. The `<new text>` is on one line, `\` represent a line break. This operation must be undoable.
10. **`set action <action index> <new target> <new label>`**: Modifies an action of the currently visible node. `<action index>` refers to the one-based index of the action to be edited, when bigger than the number of actions, it creates an action at the end. `<new target>` is the new target node ID (integer), and `<new label>` is the new text label for the action. This operation must be undoable.
11. **`set ids <id from> <id to> <+/-diff>`**: Changes the IDs of nodes within a specified range (inclusive). `<id from>` and `<id to>` define the range of node IDs to be affected, and `<+/-diff>` is an integer value to be added to (if positive) or subtracted from (if negative) the IDs within the range. For example, `set ids 3 5 +1` would increment the IDs of nodes 3, 4, and 5 to 4, 5, and 6, respectively. It should prevent having duplicates node ids. This operation must be undoable.
12. **`node <id>`**: Switch to node view of the specified node `id`. The node may not exists but setting its text will create it. Creating the node must be undoable.
13. **`node`**: Switch to node view with the starting node selected. Setting its text implicitly create the node, it must be undoable.
14. **`exit`**: Terminates the program.

3 Evaluation

This project counts towards 20% of your final mark. In this project, you will be evaluated on:

1. **Architecture and UML Class Diagrams**: Your report will discuss your *software architecture* with references to a *UML Class Diagram* of your solution.
2. **Software Patterns**: Your solution should use relevant software design patterns (at least 3), without overengineering it. Your report identifies these patterns, provides rationales for their application within your architecture, and discusses any potential trade-offs or limitations considered. Aim for effective use of patterns without unnecessary complexity.
3. **Functional Suitability**: Your solution must correctly implement all functional requirements. Automated tests will be used.
4. **Maintainability**[3]: your code maintainability will be estimated based on:
   - *Modularity*: The degree to which the Game Book Editor is composed of well-defined and independent components. Changes to one component should ideally have minimal impact on other parts of the application.
   - *Reusability*: The extent to which components within the Game Book Editor could be utilized in other parts of the editor or potentially in future, related tools.
   - *Analysability*: The ease with which the codebase of the Game Book Editor can be understood to assess the impact of potential changes (e.g., adding a new command), diagnose defects, or identify areas for improvement. Well-structured code and clear documentation contribute to analysability.
   - *Modifiability*: The effort required to implement new features (e.g., a new command), fix bugs, or adapt the Game Book Editor to future changes in the `.mini.twee` format without introducing new issues or negatively affecting existing functionality.

[3]Based on ISO/IEC 25010:2024 *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model*, accessible via https://www.nbn.be/fr/education

- *Testability*: The degree to which the Game Book Editor is designed to facilitate testing of its individual components and overall functionality.

4 Deliverables: Report and Implementation

Projects must be submitted before 2025-05-02, 23:59, on the submission platform, by groups of 2. After this time, a penalty will be applied to late submissions. This penalty is calculated as a deduction of 2^{N-1} marks (where N is the number of started days after the deadline).

Your submission archive should contain your **implementation** of the Game Book Editor and your **report**.

4.1 Report

Your report should be a PDF file at the root of your submission. It should be a strict maximum of 5 pages in English, in a reasonable layout.

- Briefly describe your application's main components and their interactions.
- Include a *UML Class Diagram* that visually represents the structure of your software design.
- Identify the *design patterns* you implemented. For each, briefly explain its purpose and trade-offs
- Include your design process, including any challenges you faced and how you overcame them.
- Discuss limitations of your design and implementation, including any known bugs or areas for improvement.

Think of this report as a helpful guide for the next person who will need to update or add features to this project based on the game makers' future requirements.

4.2 Implementation

You can choose to implement the project with **Java** or **Kotlin**. Choose whichever you dare to use.

To ensure your project can be built and executed correctly, include the necessary Gradle build configuration files (e.g. `build.gradle.kts` and `settings.gradle.kts`) and the Gradle wrapper (`./gradlew` and the `gradle` directory) at the root of your archive. Provide a task `montefiore` in your gradle build config that produces an executable JAR file in the root directory of the project named `GameBookEditor.jar`. To produce an executable JAR file, it needs to specify the `Main-Class` attribute in its `MANIFEST.MF` file. Here's an example of how to achieve this using Kotlin Gradle syntax:

build.gradle.kts
```kotlin
tasks {
    jar {
        manifest.attributes["Main-Class"] = "MainKt"
    }

    register<Copy>("montefiore"){
        group = "build"
        dependsOn(jar)
        val jarFile = jar.get().archiveFile
        from(jarFile)
        into(rootDir)
        rename(jarFile.get().asFile.name, "GameBookEditor.jar")
    }
}
```

Your application will run on Java 21.

You can use a library for JSON handling, but in general, try to minimize libraries. If you would like to use additional libraries, ask on the discussion system of eCampus. To embed your libraries inside the jar and avoid classpath issues, you may find the Shadow plugin useful to create self-contained “fat” JARs. One week before the deadline, the list of allowed libraries and any modifications to the online test script will be frozen.

Ask your questions early, and enjoy!