Text Processor

By Martin Konov Inf, 45804

> **IMPORTANT: BEFORE RUNNING THE PROJECT FOR THE FIRST TIME, OPEN `textprocessor/dataFiles/dataFiles.hpp` AND CHANGE `#DEFINE BASE_PATH "/home/mkonov/cprojects/uni/projectOOP/textProcessor/"` TO THE ACTUAL PATH OF THE PROJECT ON YOUR LOCAL MACHINE. RELATIVE PATHS DO NOT WORK.**

Testing the project

On Windows:

To run the program, navigate to `textProcessor/build-windows/`

There you will find **textProcessor.exe** executable.

On Linux:

To run the program, navigate to `textProcessor/build` There you will find the textProcessor executable.

🔍 How to Run Predefined Test Sequences

1. **Start the Program**
   Launch the executable. You'll see a main menu with available commands.

2. **Use Macros for Testing**
   A feature called a **macro** allows you to execute a predefined sequence of commands that showcase specific functionalities of the project.

3. **View Available Macros**
   At the menu, type `macro` to see a list of all predefined test macros.

4. **Run a Macro**
   Enter the name of any macro from the list. The program will then automatically execute its sequence, demonstrating the core features step-by-step.

This project is divided into two main parts:

1. **Logic**
2. **Commands** (together with the Commands CLI)

Project Structure

1. Logic

At the core of the logic layer is the **FileManager**, which handles all file I/O operations. Around the FileManager, there are several classes operating at different levels of abstraction:

- **DocumentParser**: Parses the raw file contents into a structured format.
- **DocumentRegister**: Manages loaded `Document` instances.
- **ActiveDocument** / **ActiveBlock**: Track the currently selected document and text block for operations.
- **Line**, **Document**, **LineCreator**, **BlockRegister**, **MacroRegister**, **ActiveFormatter**: Provide specialized functionality (e.g., creating lines, managing blocks, handling macros, formatting text).

These classes consume information provided by the FileManager to implement the core text-processing functionality.

2. Commands & CLI

The command layer builds on the logic classes to expose user-driven actions:

- **Command** classes (e.g., `AddLineCommand`, `SaveCommand`, `ScrambleCommand`) wrap logic operations.
- **CommandCLI** classes handle user input and output for each command.
- **CommandRegister**: The central registry that holds and executes command instances.

The **TextProcessor.cpp** file provides the running interface:

- Displays available commands to the user.
- Routes input to the CommandRegister to execute the chosen command.
- Orchestrates creation and destruction of all commands and core data classes.

Documentation

- Detailed class and method descriptions: `Documentation/html/index.html`
- UML class diagram: `Documentation/classDiagram1.drawio.png`

Future Improvements

- **Container-based Command Management**: Refactor `TextProcessor.cpp` to use STL containers (e.g., `std::vector` or `std::map`) for managing CLI and command instances, instead of manual registration and deletion.
- **Frontend Abstraction Layer**: Introduce an intermediate layer between commands and the CLI classes to support alternative frontends (e.g., GUI, REST API) without changing command logic.

External Libraries Used

In addition to the standard C++ library, the following headers were included for specific functionality:

```
#include <random>
#include <algorithm>
```

- Used in `Document::scramble()` to randomly reorder lines within a document.