HTML Editor Project Manual

Table of Contents

Introduction

Welcome to the **HTML Editor Project Manual**. This document serves as a comprehensive guide to understanding, installing, using, and extending the command-line-based HTML Editor developed for educational purposes. The project aims to demonstrate object-oriented design principles, design patterns, and the implementation of a functional HTML editor with features such as element manipulation, spell checking, undo/redo operations, and session management.

The editor allows users to interactively modify HTML documents through a series of well-defined commands, providing both flexibility and control over the document structure. Additionally, the project includes automated tests to ensure the reliability and correctness of its functionalities.

This manual is structured to provide you with all the necessary information to effectively utilize and contribute to the HTML Editor project.

Repository Information

The **HTML Editor** project is hosted on GitHub and can be accessed via the following URL:

https://github.com/eggybyte/HtmlEditor.git

The repository contains all the source code for both the HTML Editor application and its associated test suites. Ensure you have access to this repository to clone and work with the project.

Installation

Prerequisites

Before installing the **HTML Editor**, ensure that your system meets the following requirements:

- **Operating System**: Windows, macOS, or Linux.
- **.NET SDK**: Version 9.0 or later. You can download it from the .NET official website.

- **Internet Connection**: Required for spell checking functionality using the LanguageTool API.
- **Text Editor/IDE**: Any code editor or Integrated Development Environment (IDE) of your choice (e.g., Visual Studio Code, Visual Studio, JetBrains Rider).

Cloning the Repository

1. **Open Terminal or Command Prompt**
   Navigate to the directory where you want to clone the repository.
2. **Clone the Repository**
   Execute the following command to clone the repository:

   ```
   git clone https://github.com/eggybyte/HtmlEditor.git
   ```

   This command will create a local copy of the repository in a folder named `HtmlEditor`.
3. **Navigate to the Project Directory**

   ```
   cd HtmlEditor
   ```

Restoring Dependencies

The project utilizes several third-party libraries and tools. Restore these dependencies using the .NET CLI:

```
dotnet restore
```

This command will download and install all necessary packages as specified in the project files (`.csproj`).

Building the Project

Compile the project to ensure all components are correctly set up:

```
dotnet build
```

If the build is successful, you will see a `Build succeeded.` message along with other build details.

Running Automated Tests

Automated tests are crucial for verifying the correctness and stability of the project. To run the tests, execute:

```
dotnet test
```

This command will execute all test suites within the `HtmlEditor.Tests` project, providing a summary of the results. Ensure that all tests pass before proceeding to use the editor.

Running the Editor

To launch the HTML Editor application, navigate to the `HtmlEditor` project directory and run:

```
dotnet run --project HtmlEditor
```

Alternatively, you can navigate into the `HtmlEditor` directory and execute:

```
cd HtmlEditor
dotnet run
```

Upon launching, you will see a welcome message:

```
Welcome to the HTML Editor. Type 'help' to see available commands.
>
```

Additional Setup for Spell Checking

The HTML Editor integrates with the **LanguageTool API** for spell checking functionalities. No additional setup is required beyond ensuring that your system can make HTTP requests to external APIs. The application handles API interactions internally.

Note: If you encounter issues with spell checking, verify your internet connection and ensure that access to `https://api.languagetool.org` is not blocked by firewalls or network policies.

Usage

Starting the Editor

After successful installation, start the editor by navigating to the project directory and running the application:

```
dotnet run --project HtmlEditor
```

Upon launching, you will see:

```
Welcome to the HTML Editor. Type 'help' to see available commands.
>
```

Available Commands

The **HTML Editor** operates through a set of commands that allow you to manipulate HTML documents. Commands are categorized into editing, display, input/output, and undo/redo operations.

Command Examples

Here are some examples to get you started:

1. **Initialize a New HTML Document**

   ```
   init
   ```

   Initializes the editor with a basic HTML template.
2. **Insert a New Paragraph Before an Existing Element**

   ```
   insert p para1 h1 "This is a new paragraph."
   ```

   Inserts a `<p>` element with ID `para1` before the element with ID `h1`.
3. **Append a New List Item**

   ```
   append li item4 list "Item 4"
   ```

   Appends a `<li>` element with ID `item4` to the parent element with ID `list`.
4. **Edit the Text of an Element**

```
edit-text para1 "Updated paragraph content."
```

Updates the text content of the element with ID `para1`.

5. **Delete an Element**

```
delete item3
```

Deletes the element with ID `item3`.

6. **Undo the Last Operation**

```
undo
```

Reverts the last editing action.

7. **Redo the Last Undone Operation**

```
redo
```

Reapplies the last undone action.

8. **Save the Current HTML Document**

```
save mypage.html
```

Saves the current state of the HTML document to `mypage.html`.

9. **Print the HTML Structure in Tree Format**

```
print-tree
```

Displays the HTML document's structure in a tree-like format.

10. **Perform Spell Checking**

```
spell-check
```

Checks the spelling of text content within the HTML document.

11. **Exit the Editor**

```
exit
```

Exits the HTML Editor, prompting to save any unsaved changes.

Command Reference

This section provides detailed descriptions of each available command, including syntax, parameters, and expected behavior.

Editing Commands

1. Insert Command

Syntax:

```
insert tagName idValue insertLocation [textContent]
```

- `tagName` : The HTML tag name of the new element (e.g., `div` , `p` , `h1`).
- `idValue` : The unique ID for the new element.
- `insertLocation` : The ID of the existing element before which the new element will be inserted.
- `textContent` : (Optional) The text content within the new element.

Description:

Inserts a new HTML element before a specified existing element within the document structure.

Example:

```
insert p para2 h1 "Introduction paragraph."
```

This command inserts a `<p>` element with ID `para2` and text content "Introduction paragraph." before the element with ID `h1` .

Behavior:

- Validates that the `insertLocation` element exists.
- Ensures that the `idValue` is unique within the document.
- Inserts the new element at the specified location.
- Marks the document as having unsaved changes.

Error Handling:

- If the `insertLocation` does not exist, the editor will notify the user with an appropriate error message.

- If the `idValue` already exists, the editor will prevent the insertion and prompt the user to provide a unique ID.

2. Append Command

Syntax:

```
append tagName idValue parentElement [textContent]
```

- `tagName` : The HTML tag name of the new element.
- `idValue` : The unique ID for the new element.
- `parentElement` : The ID of the parent element to which the new element will be appended.
- `textContent` : (Optional) The text content within the new element.

Description:

Appends a new HTML element as the last child of a specified parent element.

Example:

```
append li item4 list "Item 4"
```

This command appends a `<li>` element with ID `item4` and text content "Item 4" to the parent element with ID `list` .

Behavior:

- Validates that the `parentElement` exists.
- Ensures that the `idValue` is unique within the document.
- Appends the new element as the last child of the specified parent.
- Marks the document as having unsaved changes.

Error Handling:

- If the `parentElement` does not exist, the editor will notify the user.
- If the `idValue` is already in use, the editor will prevent the append operation.

3. Edit-ID Command

Syntax:

```
edit-id oldId newId
```

- `oldId` : The current ID of the element.
- `newId` : The new unique ID to assign to the element.

Description:

Changes the ID of an existing HTML element to a new unique value.

Example:

```
edit-id para1 para2
```

This command changes the ID of the element from `para1` to `para2` .

Behavior:

- Validates that an element with `oldId` exists.
- Ensures that `newId` is unique within the document.
- Updates the ID of the specified element.
- Marks the document as having unsaved changes.

Error Handling:

- If the `oldId` does not correspond to any element, an error message is displayed.
- If `newId` is already in use, the editor prompts the user to provide a unique ID.

4. Edit-Text Command

Syntax:

```
edit-text element [newTextContent]
```

- `element` : The ID of the element to edit.
- `newTextContent` : (Optional) The new text content for the element. If omitted, the text content is cleared.

Description:

Edits the text content of an existing HTML element.

Example:

```
edit-text para2 "Updated content for paragraph."
```

This command updates the text content of the element with ID `para2` to "Updated content for paragraph."

Behavior:

- Validates that the specified `element` exists.
- Updates the text content of the element with the provided `newTextContent`.
- If `newTextContent` is omitted, the element's text content is cleared.
- Marks the document as having unsaved changes.

Error Handling:

- If the specified `element` does not exist, an error message is displayed.

5. Delete Command

Syntax:

```
delete element
```

- `element`: The ID of the element to delete.

Description:

Removes an existing HTML element from the document.

Example:

```
delete item3
```

This command deletes the element with ID `item3` from the document.

Behavior:

- Validates that the specified `element` exists and is not one of the protected tags (`html`, `head`, `title`, `body`).
- Removes the element from its parent.
- Marks the document as having unsaved changes.

Error Handling:

- Attempting to delete protected elements will result in an error message.
- If the specified `element` does not exist, an error is reported.

Display Commands

6. Print-Indent Command

Syntax:

```
print-indent [indent]
```

- `indent` : (Optional) The number of spaces per indentation level. Defaults to 2 if not specified.

Description:

Displays the HTML document in an indented format, making the structure more readable.

Example:

```
print-indent 4
```

This command prints the HTML structure with each level indented by 4 spaces.

Behavior:

- Converts the internal HTML model into an indented string representation.
- Outputs the formatted HTML to the console.
- Respects the `indent` parameter for customizing indentation.

Error Handling:

- If a non-integer value is provided for `indent` , the editor will notify the user and use the default indentation level.

7. Print-Tree Command

Syntax:

```
print-tree
```

Description:

Displays the HTML document's structure in a tree-like format, illustrating the hierarchical relationships between elements.

Example:

```
print-tree
```

Behavior:

- Converts the internal HTML model into a tree-like string representation.
- Outputs the formatted tree structure to the console.
- Supports marking elements with spelling errors (if spell-checking has been performed).

Error Handling:

- If the document is empty or not initialized, the editor will notify the user accordingly.

8. Spell-Check Command

Syntax:

```
spell-check
```

Description:

Performs spell checking on all text content within the HTML document, highlighting any spelling errors detected.

Example:

```
spell-check
```

Behavior:

- Initiates an asynchronous spell check of all text content within the HTML elements.
- Marks elements containing spelling errors.
- Outputs a report of detected spelling errors to the console.
- Enhances the display of the document structure by indicating errors in `print-tree`.

Error Handling:

- If the LanguageTool API is unreachable or an error occurs during the spell check, the editor will notify the user and proceed without marking errors.

Input/Output Commands

9. Read Command

Syntax:

```
read filepath
```

- `filepath` : The path to the HTML file to be read into the editor.

Description:

Loads an existing HTML file into the editor, parsing its content into the internal HTML model.

Example:

```
read mypage.html
```

Behavior:

- Validates that the specified `filepath` exists and is accessible.
- Parses the HTML content and constructs the internal object-oriented model.
- Marks the document as having unsaved changes if modifications are detected during parsing.
- Notifies the user upon successful loading.

Error Handling:

- If the specified file does not exist, the editor prompts the user with an error message.
- If the file content is malformed or cannot be parsed, the editor notifies the user.

10. Save Command

Syntax:

```
save filepath
```

- `filepath` : The path where the current HTML document will be saved.

Description:

Saves the current state of the HTML document to the specified file path.

Example:

```
save mypage.html
```

Behavior:

- Validates that the specified `filepath` is writable.
- Converts the internal HTML model into a well-formatted HTML string.
- Writes the HTML string to the specified file.
- Marks the document as saved (no unsaved changes).
- Notifies the user upon successful saving.

Error Handling:

- If the specified path is invalid or unwritable, the editor notifies the user with an error message.

11. Init Command

Syntax:

```
init
```

Description:

Initializes the editor with a basic HTML template, resetting any existing content.

Example:

```
init
```

Behavior:

- Resets the internal HTML model to a default template:

```
<html>
  <head>
    <title></title>
  </head>
  <body></body>
</html>
```

- Clears any existing editing history.
- Marks the document as having unsaved changes.
- Notifies the user upon successful initialization.

Error Handling:

- If the editor is already initialized, the command reinitializes the document, potentially discarding unsaved changes. Users are advised to save their work before reinitializing.

Undo/Redo Commands

12. Undo Command

Syntax:

```
undo
```

Description:

Reverts the last editing action performed in the editor. Supports multiple levels of undo.

Example:

```
undo
```

Behavior:

- Checks if there are actions available to undo.
- Reverts the document to the previous state.
- Updates the editing history accordingly.
- Notifies the user upon successful undo operation.

Error Handling:

- If no actions are available to undo, the editor notifies the user.

13. Redo Command

Syntax:

```
redo
```

Description:

Reapplies the last undone action. Supports multiple levels of redo.

Example:

```
redo
```

Behavior:

- Checks if there are actions available to redo.
- Reapplies the last undone action.
- Updates the editing history accordingly.
- Notifies the user upon successful redo operation.

Error Handling:

- If no actions are available to redo, the editor notifies the user.

Implementation Details

Architecture Overview

The **HTML Editor** is structured following a modular, object-oriented architecture, emphasizing separation of concerns and scalability. The primary components include:

- **CommandHandler**: Interprets and executes user commands.
- **Editor**: Represents individual HTML documents being edited.
- **HtmlEditor**: Manages the internal HTML model, including parsing, rendering, and manipulation.
- **HTMLElement and Subclasses**: Define the structure and behavior of various HTML elements.
- **HistoryManager**: Facilitates undo and redo operations.
- **SpellChecker**: Integrates with the LanguageTool API to provide spell checking capabilities.

- **DirPrinter**: Handles directory tree representations.
- **Session and SessionStateManager**: Manage multiple editors within a session and persist session states.
- **TagFactory**: Employs the Factory Design Pattern to create HTML elements based on tag names.
- **TreePrinter**: Assists in rendering tree and indented views of the HTML structure.

This architecture ensures that each module has a well-defined responsibility, promoting maintainability and ease of extension.

Core Modules

1. CommandHandler

- **Responsibility**: Parses and executes user commands.
- **Key Methods**:
  - `ExecuteCommandAsync(string input)` : Asynchronously processes a command string and returns the result.
- **Design Highlights**:
  - Utilizes a switch-case structure to handle various commands.
  - Ensures commands are executed in the context of the active editor.
  - Implements error handling to manage invalid commands gracefully.

Implementation Details:

The `CommandHandler` class is the central hub for interpreting user inputs and invoking the corresponding functionalities. It maintains the session state and ensures that each command affects the active editor appropriately. The class is designed to be easily extensible, allowing new commands to be added with minimal changes to existing code.

2. Editor

- **Responsibility**: Represents an individual HTML document within the session.
- **Key Properties**:
  - `HtmlEditor HtmlEditor` : Manages the HTML content and structure.
  - `bool IsDirty` : Indicates unsaved changes.
  - `bool ShowId` : Controls the visibility of element IDs.
  - `string FilePath` : Path to the associated HTML file.
- **Design Highlights**:
  - Integrates with `HtmlEditor` for content management.
  - Maintains state information for persistence and display purposes.

- Ensures that each editor maintains its own history for undo/redo operations.

Implementation Details:

The `Editor` class encapsulates the state and behavior of a single HTML document. It interacts with the `HtmlEditor` to perform content manipulations and tracks whether the document has unsaved changes (`IsDirty`). The `ShowId` property allows users to toggle the visibility of element IDs in display commands.

3. HtmlEditor

- **Responsibility**: Manages the HTML document's internal model, including parsing, rendering, and manipulation.
- **Key Methods**:
  - `Init()` : Initializes a new HTML document.
  - `Read(string filepath)` : Loads an HTML file into the model.
  - `Save(string filepath)` : Saves the current model to a file.
  - `Insert(...)` , `Append(...)` , `EditId(...)` , `EditText(...)` , `Delete(...)` : Methods for modifying the HTML structure.
  - `Undo()` , `Redo()` : Manage undo and redo operations.
  - `PerformSpellCheckAsync()` : Initiates spell checking using the LanguageTool API.
- **Design Highlights**:
  - Employs a hierarchical model using `HTMLElement` classes.
  - Integrates `HistoryManager` for state tracking.
  - Uses `TagFactory` for element creation, promoting scalability.
  - Implements robust error handling to manage invalid operations.

Implementation Details:

The `HtmlEditor` class is the core of the editor's functionality, responsible for maintaining and manipulating the HTML document's structure. It uses an object-oriented approach, representing each HTML element as an instance of `HTMLElement` or its subclasses. The class supports various operations to modify the document, such as inserting, appending, editing, and deleting elements. Additionally, it integrates with the `HistoryManager` to provide undo and redo capabilities.

4. HTMLElement and Subclasses

- **Responsibility**: Define the structure and behavior of HTML elements.
- **Key Classes**:
  - `Html` : Represents the `<html>` root element.
  - `Head` , `Body` , `Title` : Represent respective HTML sections.

    - `Div`, `Paragraph`, `Header`: Represent common HTML elements.
    - `GenericElement`: Handles non-specific or custom tags.
- **Design Highlights**:
    - Implements the Composite Design Pattern, allowing uniform treatment of individual elements and compositions.
    - Each subclass tailors behavior specific to its tag type.
    - Ensures unique IDs across elements for reliable manipulation.
    - Supports cloning to facilitate history management.

Implementation Details:

The `HTMLElement` class serves as an abstract base for all HTML elements, encapsulating common properties such as `TagName`, `Id`, `Content`, and `Children`. Subclasses like `Html`, `Head`, `Body`, `Title`, `Div`, `Paragraph`, and `Header` extend `HTMLElement` to represent specific HTML tags with tailored behaviors. The `GenericElement` class provides flexibility for handling tags that do not have dedicated subclasses.

The composite structure allows for recursive operations, such as rendering and spell checking, by treating both individual elements and their collections uniformly.

5. HistoryManager

- **Responsibility**: Manages undo and redo operations by maintaining history stacks.
- **Key Methods**:
    - `SaveState(Html state)`: Saves the current state for potential undo operations.
    - `Undo()`: Reverts to the previous state.
    - `Redo()`: Reapplies an undone state.
- **Design Highlights**:
    - Utilizes two stacks (`undoStack`, `redoStack`) to track history.
    - Ensures a clear separation between undo and redo functionalities.
    - Prevents inconsistent state transitions by handling state snapshots accurately.

Implementation Details:

The `HistoryManager` class employs a stack-based approach to manage the history of HTML states, facilitating multiple levels of undo and redo operations. Each time an edit is made, the current state is cloned and pushed onto the `undoStack`. Performing an `Undo` operation pops the latest state from the `undoStack` and pushes it onto the `redoStack`, allowing for subsequent `Redo` operations. This design ensures that the user can navigate through their editing history seamlessly.

6. SpellChecker

- **Responsibility**: Provides spell checking capabilities by integrating with the LanguageTool API.
- **Key Methods**:
  - `CheckSpellingAsync(HTMLElement rootElement)` : Initiates asynchronous spell checking.
  - `CheckSpellingRecursiveAsync(...)` : Recursively checks text content within HTML elements.
  - `CheckTextWithLanguageTool(...)` : Communicates with the LanguageTool API to validate text.
- **Design Highlights**:
  - Handles asynchronous operations to maintain responsiveness.
  - Processes API responses to identify and mark spelling errors within the HTML model.
  - Implements error handling for API interactions.
  - Utilizes HTTP client efficiently with proper resource management.

Implementation Details:

The `SpellChecker` class is responsible for verifying the spelling of all textual content within the HTML document. It uses the LanguageTool API to perform spell checking, sending HTTP requests asynchronously to avoid blocking the main editing operations. The class processes the API responses to identify misspelled words and marks the corresponding `HTMLElement` instances with spelling errors. This integration enhances the quality of the HTML content by assisting users in identifying and correcting spelling mistakes.

7. DirPrinter

- **Responsibility**: Renders directory structures in both tree and indented formats.
- **Key Methods**:
  - `PrintDirTree(...)` : Generates a tree-like representation of directories and files.
  - `PrintDirIndent(...)` : Generates an indented list of directories and files.
- **Design Highlights**:
  - Leverages existing `TreePrinter` functionality for consistency.
  - Integrates with the session's list of editors to mark unsaved files.
  - Provides customizable indentation levels for better readability.

Implementation Details:

The `DirPrinter` class facilitates the visualization of the current working directory's structure. It provides two primary methods: `PrintDirTree` , which displays the directory and file structure in a tree-like format with symbols, and `PrintDirIndent` , which uses indentation to represent hierarchy. The class also integrates with the session's editors to indicate unsaved files, enhancing the user's ability to manage project files effectively.

8. Session and SessionStateManager

- **Session**:
  - **Responsibility**: Manages multiple editors within a single user session.
  - **Key Properties**:
    - `List<Editor> Editors` : Collection of open editors.
    - `Editor ActiveEditor` : Currently active editor.
  - **Key Methods**:
    - `AddEditor(...)` , `RemoveEditor(...)` : Manage editors within the session.
    - `GetEditorByFilePath(...)` : Retrieve editors based on file paths.
    - `GetEditorsState()` : Retrieves the current state of all editors for persistence.
- **SessionStateManager**:
  - **Responsibility**: Handles the persistence of session states to and from storage.
  - **Key Methods**:
    - `LoadSessionState(string tempFolder)` : Loads session state from a specified directory.
    - `SaveSessionState(Session session, string tempFolder)` : Saves the current session state.
  - **Design Highlights**:
    - Ensures session persistence across application restarts.
    - Manages the active editor and maintains the integrity of the editors' states.
    - Handles serialization and deserialization of session data using JSON.

Implementation Details:

The `Session` class encapsulates the state of a user's editing session, managing multiple `Editor` instances and tracking the currently active editor. This design allows users to work on multiple HTML documents simultaneously, switching between them as needed.

The `SessionStateManager` class is responsible for saving and loading the session state, ensuring that users can resume their work seamlessly after restarting the application. It serializes the session state to a JSON file located in a designated temporary folder (`.temp`), capturing information such as the list of open editors, their file paths, and the active editor. Upon loading, it reconstructs the session by deserializing the JSON data and initializing the corresponding `Editor` instances.

9. TagFactory

- **Responsibility**: Creates HTML elements based on tag names using the Factory Design Pattern.
- **Key Methods**:
  - `CreateElement(string tagName, string id, string content)` : Instantiates appropriate `HTMLElement` subclasses.

- **Design Highlights**:
  - Facilitates the addition of new HTML elements without modifying existing code.
  - Enhances scalability and maintainability by centralizing element creation logic.
  - Implements error handling to manage unsupported or invalid tag names gracefully.

Implementation Details:

The `TagFactory` class employs the Factory Design Pattern to abstract the creation of various `HTMLElement` instances. By centralizing the instantiation logic, it simplifies the process of adding support for new or custom HTML tags. The factory method examines the provided `tagName` and returns an instance of the corresponding subclass. If an unsupported tag is encountered, it defaults to creating a `GenericElement`, ensuring that the editor remains flexible and extensible.

10. TreePrinter

- **Responsibility**: Renders tree-like and indented representations of HTML elements.
- **Key Methods**:
  - `Print(...)`: Generates string representations of the HTML tree.
  - `PrintTreeRecursive(...)`: Helper method for recursive tree printing.
- **Design Highlights**:
  - Provides flexibility in display formats through configurable parameters.
  - Integrates with `HTMLElement` classes to access hierarchical data.
  - Supports marking elements with spelling errors for enhanced readability.

Implementation Details:

The `TreePrinter` class is a utility that converts the internal HTML model into human-readable string representations. It supports two primary formats:

1. **Tree Format**: Utilizes symbols like `└──` and `├──` to depict the hierarchical structure of HTML elements, making it easy to visualize parent-child relationships.
2. **Indented Format**: Uses spaces to represent the depth of each element in the hierarchy, providing an alternative view that may be preferable in certain contexts.

Additionally, `TreePrinter` integrates with the `SpellChecker` by marking elements containing spelling errors with a `[X]` prefix, aiding users in quickly identifying and addressing issues within the document.

Design Patterns Used

The **HTML Editor** leverages several design patterns to ensure a robust and maintainable codebase:

1. **Factory Pattern** (`TagFactory`): Simplifies the creation of `HTMLElement` instances based on tag names, promoting scalability and ease of adding new tags.
2. **Composite Pattern** (`HTMLElement` and its subclasses): Allows treating individual elements and compositions uniformly, facilitating recursive operations like rendering and spell checking.
3. **Command Pattern** (`CommandHandler`): Encapsulates user commands as objects, enabling flexible command execution and management.
4. **Singleton Pattern** (`HistoryManager`): Ensures a single instance of the history manager, maintaining consistent undo/redo operations across the application.
5. **Observer Pattern** (`SpellChecker` observing `HTMLElement`): Monitors changes in HTML elements to trigger spell checking automatically.
6. **Dependency Injection** (`CommandHandler` receiving `Session`): Enhances testability and decouples components by injecting dependencies.
7. **Strategy Pattern** (`DirPrinter` using different printing strategies): Allows selecting different algorithms (tree vs. indented) for directory representation at runtime.

Dependency Management

Managing dependencies is crucial for ensuring the stability and maintainability of the project. The **HTML Editor** uses the following strategies for dependency management:

- **NuGet Packages**: Utilizes NuGet for managing third-party libraries, ensuring that dependencies are versioned and easily updatable.
  - **xUnit**: For automated testing.
  - **coverlet.collector**: For code coverage analysis.
  - **Microsoft.NET.Test.Sdk**: For running tests.
- **Project References**: Maintains clear project references between the main application (`HtmlEditor`) and its test project (`HtmlEditor.Tests`), ensuring that tests have access to the necessary codebases.
- **Version Pinning**: Specifies exact versions of dependencies in the `.csproj` files to prevent unexpected behavior due to breaking changes in third-party libraries.
- **Isolation of Unstable Libraries**: Third-party libraries that are less stable or prone to changes are encapsulated within dedicated modules or interfaces, minimizing their impact on the core application logic.
- **Package Restore**: Ensures that all dependencies are restored before building or testing the project, facilitating seamless collaboration across different development environments.

Features

HTML Model

The **HTML Editor** maintains an internal object-oriented representation of HTML documents. This model mirrors the hierarchical structure of HTML, enabling efficient manipulation and rendering. Key aspects include:

- **Element Hierarchy**: Elements are organized in a tree structure, with `<html>` as the root, containing `<head>` and `<body>`, which in turn contain other elements.
- **Unique Identification**: Except for `<html>`, `<head>`, `<title>`, and `<body>`, all elements must have unique `id` attributes, ensuring precise targeting during operations.
- **Content Management**: Elements can contain text content and nested child elements, allowing for complex document structures.
- **Cloning and History Tracking**: The model supports cloning of elements to facilitate accurate state tracking for undo and redo operations.

Editing Functionality

The editor supports a range of commands to modify the HTML document:

- **Insert**: Adds new elements before specified existing elements.
- **Append**: Adds new elements as the last child of specified parent elements.
- **Edit-ID**: Changes the `id` attribute of existing elements.
- **Edit-Text**: Updates the text content within elements.
- **Delete**: Removes elements from the document.

These functionalities are designed to be intuitive and flexible, allowing users to construct and modify HTML documents efficiently.

Advanced Features:

- **Automatic ID Assignment**: For protected tags (`html`, `head`, `title`, `body`), IDs are assigned automatically if not provided, ensuring uniqueness and consistency.
- **Validation**: The editor validates the uniqueness of IDs and the existence of target elements before performing any operation, preventing structural inconsistencies.

Input/Output Operations

The editor facilitates reading from and writing to HTML files:

- **Read**: Imports an existing HTML file into the editor's internal model, allowing for continued editing.
- **Save**: Exports the current state of the HTML model to a file, ensuring that changes are preserved.
- **Init**: Initializes the editor with a basic HTML template, providing a clean slate for new documents.

Additional Features:

- **Error Handling**: Robust mechanisms are in place to handle scenarios where files are missing, paths are invalid, or content is malformed, providing meaningful feedback to users.
- **Session Persistence**: The editor can save and restore session states, allowing users to resume their work seamlessly after restarting the application.

Spell Checking

Integrated with the LanguageTool API, the spell checking feature enhances content quality:

- **Asynchronous Checking**: Ensures that spell checking does not block the main editing operations, maintaining responsiveness.
- **Error Marking**: Highlights elements containing spelling errors, aiding users in identifying and correcting mistakes.
- **Comprehensive Coverage**: Checks all text content within the document, including nested elements, ensuring thorough validation.
- **Error Reporting**: Provides detailed reports of spelling errors, including suggestions for corrections and contextual information.

Implementation Details:

- **Language Detection**: The spell checker automatically detects the language of the text content, providing relevant suggestions.
- **API Integration**: Utilizes the LanguageTool API to perform spell checks, handling HTTP requests and parsing responses effectively.
- **Error Resilience**: Implements error handling to manage API unavailability or network issues gracefully, ensuring that the editor remains functional even if spell checking fails.

Undo/Redo Operations

The editor implements a robust undo/redo system:

- **Multiple Levels**: Supports multiple consecutive undo and redo actions, allowing users to navigate through their editing history seamlessly.
- **State Management**: Each editing action updates the history stack, ensuring accurate state transitions and preventing loss of work.

- **Isolation**: Undo/redo operations are confined to the active editor, maintaining the integrity of multiple open documents within a session.
- **Command Exclusions**: Display and input/output commands are excluded from the undo/redo history, ensuring that only editing actions affect the history stack.

Implementation Details:

- **HistoryManager Integration**: The `HistoryManager` class tracks changes to the HTML model, enabling users to revert or reapply actions as needed.
- **State Cloning**: Before performing any edit operation, the current state is cloned and saved, ensuring that the history accurately reflects each change.
- **Conflict Resolution**: If a new edit is made after an undo operation, the redo stack is cleared to prevent inconsistent state transitions.

Session Management

Managing multiple HTML documents within a single session is streamlined:

- **Multiple Editors**: Users can load and edit multiple HTML files simultaneously, each with its own independent undo/redo history.
- **Active Editor**: Only one editor is active at a time, but users can switch between editors seamlessly using the `edit` command.
- **Session Persistence**: Session states, including open editors and the active editor status, are saved and restored across application restarts, ensuring continuity of work.
- **Editor List Management**: Commands like `editor-list` allow users to view all open editors, indicating which one is active and which ones have unsaved changes.

Implementation Details:

- **Session Class**: Encapsulates the collection of open editors and tracks the active editor, providing methods to add, remove, and retrieve editors.
- **SessionStateManager**: Handles the serialization and deserialization of session states to JSON files, preserving information such as open editors, their file paths, and visibility settings.
- **User Feedback**: The editor provides clear feedback when loading, switching, or closing editors, ensuring that users are aware of the current session state.

Directory Display

The editor provides commands to visualize the file system structure:

- **Dir-Tree**: Displays the current working directory and its subdirectories in a tree-like format, with symbols representing the hierarchy.
- **Dir-Indent**: Shows the directory structure with indentation, offering an alternative view that emphasizes the nesting levels through spaces.

These features aid users in managing and navigating their projects effectively, providing a clear overview of the file organization.

Implementation Details:

- **Integration with TreePrinter**: Reuses the `TreePrinter` functionality to ensure consistency between HTML structure and directory representations.
- **Marking Unsaved Files**: When displaying directory structures, files that have unsaved changes are marked with an asterisk (`*`), alerting users to potential data loss.
- **Customization**: The `dir-indent` command supports customizable indentation levels, allowing users to adjust the display according to their preferences.

Testing

Test Strategy

The **HTML Editor** employs a combination of unit tests and integration tests to ensure the reliability and correctness of its functionalities. Utilizing the `xUnit` framework, the tests cover various components, including command handling, HTML manipulation, spell checking, and session management.

Key Objectives:

- **Functionality Verification**: Ensure that each command and operation behaves as expected under normal and edge-case scenarios.
- **Regression Prevention**: Detect and prevent regressions by continuously validating existing functionalities during development.
- **Code Coverage**: Achieve comprehensive test coverage to identify untested paths and improve code quality.
- **Automation**: Integrate automated tests into the development workflow to facilitate continuous integration and deployment.

Automated Tests

The project includes two primary test suites within the `HtmlEditor.Tests` project:

HtmlEditorCommandTests

This test suite verifies the correct execution of user commands and their impact on the session state.

- **TestCommandSequenceFromFileAsync**: Executes a sequence of commands from a file and logs the results, ensuring that the editor processes commands accurately.
  Key Features:
  - **Command Execution**: Reads commands from a `commands.txt` file and executes them sequentially.
  - **Logging**: Records each command and its corresponding output to a log file (`command_output_log.txt`).
  - **Error Handling**: Detects and reports issues encountered during command execution.
  - **Session State Persistence**: Saves the session state after executing all commands to verify persistence mechanisms.

  Implementation Details:
  This asynchronous test ensures that the editor can handle a batch of commands, simulating real-world usage scenarios. It validates that commands like `init`, `append`, `insert`, `print-tree`, `spell-check`, and `save` are executed correctly and that their effects are accurately reflected in the editor's state.

HtmlEditorTests

This suite focuses on the core functionalities of the `HtmlEditor` class.

- **TestInitialization**: Validates that the editor initializes with the correct HTML structure.
  Key Features:
  - **HTML Structure Validation**: Ensures that the `<html>`, `<head>`, `<title>`, and `<body>` elements are correctly instantiated.
  - **Default Content**: Checks that default content is set appropriately during initialization.
- **TestAppendElement**: Ensures that appending elements works as expected.
  Key Features:
  - **Element Creation**: Verifies that elements like `<div>` are correctly appended to the parent element.
  - **Content Assignment**: Confirms that the text content within appended elements is accurately set.
- **TestInsertElement**: Checks the insertion of elements before specified locations.

Key Features:
    - **Element Positioning**: Validates that new elements are correctly inserted before target elements.
    - **Hierarchy Maintenance**: Ensures that the parent-child relationships remain consistent after insertion.
- **TestEditText**: Verifies that editing text content updates the element correctly.
  Key Features:
    - **Content Modification**: Confirms that the text content of elements is updated as intended.
    - **State Tracking**: Checks that the editor correctly marks the document as having unsaved changes after text edits.
- **TestEditId**: Confirms that changing an element's ID maintains uniqueness and updates references.
  Key Features:
    - **ID Uniqueness**: Ensures that new IDs do not conflict with existing ones.
    - **Reference Updates**: Validates that elements can be retrieved using their new IDs after modification.
- **TestDeleteElement**: Ensures that deleting elements removes them from the document.
  Key Features:
    - **Element Removal**: Confirms that elements are successfully removed from the HTML model.
    - **Hierarchy Integrity**: Ensures that the parent elements correctly reflect the removal of child elements.
- **TestUndoRedo**: Tests the undo and redo functionalities, verifying state transitions.
  Key Features:
    - **History Tracking**: Validates that actions are correctly recorded in the history stacks.
    - **State Restoration**: Ensures that undo operations revert the document to previous states and that redo operations can reapply them.
    - **Consistency**: Checks that multiple levels of undo and redo are handled accurately.

Implementation Details:

Each test case in the `HtmlEditorTests` suite isolates specific functionalities, providing clear and focused validation of individual features. By simulating user actions and verifying their outcomes, these tests ensure that the editor behaves predictably and reliably across various scenarios.

Running Tests

To execute the automated tests, navigate to the project directory and run:

```
dotnet test
```

This command will execute all test suites within the `HtmlEditor.Tests` project, providing a detailed summary of the results, including any failures or issues that need addressing.

Test Reporting:

- **Console Output**: Displays real-time test execution progress and results.
- **Code Coverage**: Utilizing `coverlet.collector`, the project can generate code coverage reports to identify untested code paths.
  Generating Code Coverage Report:

  ```
  dotnet test /p:CollectCoverage=true
  ```

  This command generates a coverage report in various formats (e.g., Cobertura, opencover) that can be further analyzed using compatible tools.

Best Practices:

- **Continuous Testing**: Integrate automated tests into the development workflow, ensuring that changes do not break existing functionalities.
- **Test-Driven Development (TDD)**: Consider writing tests before implementing new features to guide development and ensure comprehensive coverage.
- **Mocking External Dependencies**: Use mocking frameworks to simulate interactions with external services (e.g., LanguageTool API) during testing, isolating the code under test.

Design Decisions

Several key design decisions were made to shape the architecture and functionality of the **HTML Editor**:

1. **Modular Architecture**: The project is divided into distinct modules, each responsible for specific aspects of the editor. This separation enhances maintainability and scalability, allowing developers to work on individual components without affecting others.
2. **Object-Oriented Design**: Utilizing object-oriented principles, the editor models HTML elements and operations in a natural and intuitive manner, facilitating easier manipulation and extension. Classes like `HTMLElement` and its subclasses mirror the structure of HTML, enabling hierarchical and recursive operations.

3. **Design Patterns**: Implementing design patterns such as Factory, Composite, and Command patterns promotes code reuse, flexibility, and robustness. For instance, the `TagFactory` simplifies element creation, while the `Composite` pattern allows uniform treatment of individual and composite elements.
4. **Dependency Management**: Third-party libraries, particularly the LanguageTool API for spell checking, are encapsulated within dedicated modules. Dependencies are managed carefully using NuGet, specifying exact versions in the `.csproj` files to prevent unexpected behavior due to breaking changes.
5. **Extensibility**: The architecture allows for easy addition of new commands, HTML elements, and features without significant refactoring, supporting future enhancements. The use of interfaces and abstract classes facilitates the integration of new components.
6. **Error Handling**: Comprehensive error handling mechanisms are in place to manage unexpected scenarios gracefully, providing meaningful feedback to users and maintaining application stability. This includes handling invalid commands, duplicate IDs, file I/O errors, and API failures.
7. **User Experience**: Despite being a command-line tool, efforts have been made to ensure that the editor is user-friendly, with clear prompts, helpful messages, and intuitive command structures. The inclusion of display commands like `print-tree` and `print-indent` enhances readability and usability.
8. **Testing Strategy**: A robust testing framework using `xUnit` ensures that all functionalities are thoroughly validated. Tests cover both unit-level and integration-level scenarios, promoting code quality and reliability.
9. **Session Management**: Incorporating session management allows users to work on multiple documents simultaneously, similar to modern editors like VSCode. This design decision enhances productivity and aligns the editor's functionality with user expectations.
10. **Asynchronous Operations**: Implementing asynchronous spell checking ensures that the editor remains responsive during network operations, providing a smoother user experience.
11. **Configuration Management**: The use of `.temp` directories and JSON files for session persistence ensures that user settings and session states are maintained across application restarts, offering continuity of work.
12. **Scalability Considerations**: The architecture is designed to handle large and complex HTML documents efficiently, with optimizations in rendering and manipulation operations to maintain performance.

Future Improvements

While the **HTML Editor** meets the current requirements, several enhancements can be considered to further improve its functionality and user experience:

1. **Graphical User Interface (GUI)**: Developing a GUI would make the editor more accessible and intuitive, appealing to a broader range of users. A graphical interface could provide drag-and-drop functionalities, real-time previews, and visual editing tools.
2. **Advanced HTML Parsing**: Integrating a more robust HTML parser could handle complex and malformed HTML documents more effectively. Enhancements could include better error recovery mechanisms and support for a wider range of HTML features.
3. **Syntax Highlighting**: Implementing syntax highlighting in the display commands would improve readability and make structure analysis easier. Highlighting different tags, attributes, and text content can enhance the visual distinction of various elements.
4. **Plugin Support**: Allowing third-party plugins could extend the editor's capabilities, enabling customization and the addition of specialized features. A plugin architecture would facilitate community-driven enhancements and modular expansions.
5. **Enhanced Spell Checking**: Incorporating more advanced spell checking and grammar correction functionalities would further improve content quality. Features like grammar suggestions, style recommendations, and language support expansions can be integrated.
6. **Collaboration Features**: Adding support for collaborative editing would enable multiple users to work on the same document simultaneously. Features like real-time synchronization, conflict resolution, and user presence indicators can enhance teamwork.
7. **Version Control Integration**: Integrating with version control systems like Git would facilitate tracking changes and managing document versions. Features like commit history, branching, and merging can be incorporated to align with standard development workflows.
8. **Performance Optimization**: Optimizing performance for handling large HTML documents and complex operations would ensure responsiveness and efficiency. Techniques like lazy loading, efficient data structures, and parallel processing can be explored.
9. **Extensive Testing**: Expanding the test suite to cover more edge cases, integration scenarios, and performance benchmarks would further enhance code reliability and maintainability.
10. **Internationalization and Localization**: Supporting multiple languages for the user interface and spell checking can broaden the editor's usability across different regions and language preferences.
11. **Export Formats**: Allowing the export of HTML documents to other formats (e.g., Markdown, PDF) would add versatility to the editor, catering to various user needs.
12. **Security Enhancements**: Implementing security measures to prevent malicious inputs, ensure safe file operations, and protect user data can make the editor more secure for widespread use.

13. **Documentation and Tutorials**: Developing comprehensive documentation, tutorials, and example projects can assist users in getting started and leveraging the editor's full potential.
14. **User Customization**: Allowing users to customize settings, key bindings, and display preferences can enhance the personalization and adaptability of the editor.
15. **Integration with Development Tools**: Connecting the editor with other development tools like linters, formatters, and build systems can streamline the development workflow.

Conclusion

The **HTML Editor** project serves as a practical application of object-oriented design principles and design patterns, demonstrating how a command-line tool can effectively manage and manipulate HTML documents. Through its modular architecture, comprehensive feature set, and robust testing framework, the editor provides a solid foundation for further development and enhancement.

By adhering to best practices in software design and ensuring thorough testing, the project not only meets the immediate educational objectives but also sets the stage for future extensions and real-world applicability. Whether used as a learning tool or as a foundation for more sophisticated applications, the HTML Editor exemplifies the integration of theoretical concepts into functional software solutions.

The ongoing development and potential enhancements outlined in this manual underscore the project's commitment to scalability, usability, and performance, ensuring that it remains a valuable resource for both users and developers.

Appendices

Appendix A: Installation Scripts

For ease of installation, consider adding scripts or Makefiles that automate the setup process.

Example: `setup.sh` (for Unix-based systems)

```
#!/bin/bash

# Clone the repository
git clone https://github.com/eggybyte/HtmlEditor.git
cd HtmlEditor

# Restore dependencies
dotnet restore

# Build the project
dotnet build

# Run tests
dotnet test

echo "Installation complete. You can now run the editor using 'dotnet run --project HtmlEditor'.
```

Example: `setup.bat` (for Windows)

```
@echo off

:: Clone the repository
git clone https://github.com/eggybyte/HtmlEditor.git
cd HtmlEditor

:: Restore dependencies
dotnet restore

:: Build the project
dotnet build

:: Run tests
dotnet test

echo Installation complete. You can now run the editor using 'dotnet run --project HtmlEditor'.
pause
```

Usage Instructions:

1. **Unix-based Systems (macOS/Linux):**
   - Save the above script as `setup.sh` in your desired directory.
   - Make the script executable:

     ```
     chmod +x setup.sh
     ```

   - Execute the script:

     ```
     ./setup.sh
     ```

2. **Windows:**
   - Save the above script as `setup.bat` in your desired directory.
   - Double-click the script to execute or run it via Command Prompt:

     ```
     setup.bat
     ```

Notes:

- Ensure that `git` and the `.NET SDK` are installed and accessible in your system's PATH.
- The scripts clone the repository, restore dependencies, build the project, run tests, and notify the user upon completion.

Appendix B: Sample Commands File (`commands.txt`)

For testing purposes, a sample `commands.txt` file can be used to automate a sequence of commands. This file allows developers and testers to verify that the editor processes commands correctly and that the resulting HTML structure matches expectations.

Sample `commands.txt`

```
init
append h1 page-title body "Welcome to my webpage"
append p description body "This is a paragraph."
append ul list body
append li item1 list "Item 1"
append li item2 list "Item 2"
append li item3 list "Item 3"
append div footer body "this is a text contect in div"
append p last-updated footer "Last updated: 2024-01-01"
append p copyright footer "Copyright © 2021 MyWebpage.com"
print-tree
spell-check
save sample.html
```

Explanation of Commands:

1. **init**: Initializes the editor with a basic HTML template.
2. **append h1 page-title body "Welcome to my webpage"**: Appends an `<h1>` element with ID `page-title` and the specified text to the `<body>`.
3. **append p description body "This is a paragraph."**: Appends a `<p>` element with ID `description` to the `<body>`.
4. **append ul list body**: Appends a `<ul>` element with ID `list` to the `<body>`.
5. **append li item1 list "Item 1"**: Appends an `<li>` element with ID `item1` to the `<ul>` with ID `list`.
6. **append li item2 list "Item 2"**: Appends an `<li>` element with ID `item2` to the `<ul>` with ID `list`.
7. **append li item3 list "Item 3"**: Appends an `<li>` element with ID `item3` to the `<ul>` with ID `list`.
8. **append div footer body "this is a text contect in div"**: Appends a `<div>` element with ID `footer` and the specified text to the `<body>`.
9. **append p last-updated footer "Last updated: 2024-01-01"**: Appends a `<p>` element with ID `last-updated` to the `<div>` with ID `footer>`.
10. **append p copyright footer "Copyright © 2021 MyWebpage.com"**: Appends a `<p>` element with ID `copyright`.
11. **print-tree**: Displays the HTML structure in a tree-like format.
12. **spell-check**: Performs spell checking on the HTML content.
13. **save sample.html**: Saves the current HTML document to `sample.html`.

Usage Instructions:

1. **Create the Commands File:**
   - Save the above commands in a file named `commands.txt` in the root directory of the project.
2. **Run the Test:**
   - Execute the `HtmlEditorCommandTests.TestCommandSequenceFromFileAsync` test, which reads commands from `commands.txt`, processes them, and logs the results to `command_output_log.txt`.
3. **Verify the Output:**
   - Check the `sample.html` file to ensure that the HTML structure matches the expected outcome.
   - Review the `command_output_log.txt` file for detailed information about each command execution and any errors encountered.

Customization:

Feel free to modify the `commands.txt` file to include additional commands or different scenarios, allowing for extensive testing of the editor's capabilities.

Appendix C: Dependencies

The **HTML Editor** project relies on the following third-party libraries and tools:

- **xUnit**: For automated testing.
  - **Version**: 2.9.0
  - **Purpose**: Provides a robust framework for writing and executing unit and integration tests.
- **xunit.runner.visualstudio**: For integrating xUnit tests with Visual Studio.
  - **Version**: 2.8.2
  - **Purpose**: Enables running xUnit tests within Visual Studio's Test Explorer.
- **coverlet.collector**: For code coverage analysis.
  - **Version**: 6.0.2
  - **Purpose**: Collects code coverage data during test execution, aiding in identifying untested code paths.
- **Microsoft.NET.Test.Sdk**: For running tests.
  - **Version**: 17.11.1
  - **Purpose**: Provides the necessary infrastructure to run tests using various testing frameworks.

- **LanguageTool API**: For spell checking functionality.
  - **URL**: https://dev.languagetool.org/public-http-api
  - **Purpose**: Offers a free and open-source service for checking grammar and spelling in multiple languages.
- **System.Text.Json**: For JSON serialization and deserialization.
  - **Version**: Included in .NET 9.0
  - **Purpose**: Facilitates the conversion between JSON strings and .NET objects, essential for session state management.

Dependency Management Strategies:

1. **NuGet Package Management:**
   - Dependencies are managed using NuGet, ensuring that all third-party libraries are versioned and easily updatable.
   - The `.csproj` files specify exact versions for each package to prevent incompatibilities and unexpected behavior due to version changes.
2. **Project References:**
   - The `HtmlEditor.Tests` project references the `HtmlEditor` project, ensuring that tests have access to the necessary codebases.
   - This setup promotes encapsulation and separation between the application logic and its tests.
3. **Isolation of External APIs:**
   - The `SpellChecker` module handles all interactions with the LanguageTool API, isolating external dependencies from the core application logic.
   - This design allows for easier mocking and testing, as well as potential replacement of the API if needed.
4. **Implicit Usings and Nullable Contexts:**
   - The projects enable `ImplicitUsings` and `Nullable` contexts in the `.csproj` files, simplifying code by reducing boilerplate and enforcing null safety.

Example: `.csproj` Dependency References

HtmlEditor.csproj

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net9.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <OutputType>Exe</OutputType>
    <PublishDir>..\release\HtmlEditor\</PublishDir>
  </PropertyGroup>

</Project>
```

HtmlEditor.Tests.csproj

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net9.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <IsPackable>false</IsPackable>
    <PublishDir>..\release\HtmlEditor.Tests\</PublishDir>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="coverlet.collector" Version="6.0.2" />
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.11.1" />
    <PackageReference Include="xunit" Version="2.9.0" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.8.2" />
  </ItemGroup>

  <ItemGroup>
    <Using Include="Xunit" />
  </ItemGroup>

  <ItemGroup>
    <None Update="files\**\*">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </None>
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\HtmlEditor\HtmlEditor.csproj" />
  </ItemGroup>

</Project>
```

Notes:

- The `PublishDir` property specifies the output directory for the build artifacts, organizing them into `release\HtmlEditor\` and `release\HtmlEditor.Tests\` respectively.
- `IsPackable` is set to `false` for the test project to prevent it from being packaged as a reusable library.
- The `coverlet.collector` package is used for collecting code coverage data during test runs.
- The `xunit.runner.visualstudio` package enables integration with Visual Studio's Test Explorer, allowing for convenient test execution and monitoring.

Appendix D: Project Structure

An overview of the project's directory structure provides insight into how the components are organized, facilitating easier navigation and maintenance.

```
HtmlEditor/
│
├── HtmlEditor/
│   ├── Program.cs
│   ├── CommandHandler.cs
│   ├── Editor.cs
│   ├── HtmlEditor.cs
│   ├── HTMLElement.cs
│   ├── Html.cs
│   ├── Head.cs
│   ├── Body.cs
│   ├── Title.cs
│   ├── Paragraph.cs
│   ├── Div.cs
│   ├── Header.cs
│   ├── GenericElement.cs
│   ├── HistoryManager.cs
│   ├── SpellChecker.cs
│   ├── DirPrinter.cs
│   ├── TagFactory.cs
│   ├── TreePrinter.cs
│   ├── Session.cs
│   ├── SessionStateManager.cs
│   └── HtmlEditor.csproj
│
├── HtmlEditor.Tests/
│   ├── HtmlEditorCommandTests.cs
│   ├── HtmlEditorTests.cs
│   └── HtmlEditor.Tests.csproj
│
├── files/
│   ├──.temp/
│   ├── session_state.json
│   └── commands.txt
│
├── README.md
├── setup.sh
├── setup.bat
└── (Other project files)
```

Detailed Breakdown:

1. **HtmlEditor/**
   Contains the main application code for the HTML Editor.
   - **Program.cs**: The entry point of the application, handling the initialization of sessions and command processing loops.
   - **CommandHandler.cs**: Processes user commands, executing corresponding actions on the HTML model.
   - **Editor.cs**: Represents an individual HTML document, managing its state and interactions.
   - **HtmlEditor.cs**: Manages the internal HTML model, including parsing, rendering, and manipulation operations.
   - **HTMLElement.cs**: Abstract base class for all HTML elements, defining common properties and methods.
   - **Html.cs, Head.cs, Body.cs, Title.cs, Paragraph.cs, Div.cs, Header.cs, GenericElement.cs**: Subclasses of `HTMLElement` representing specific HTML tags with tailored behaviors.
   - **HistoryManager.cs**: Handles the undo and redo functionality by maintaining state history.
   - **SpellChecker.cs**: Integrates with the LanguageTool API to perform spell checking on text content.
   - **DirPrinter.cs**: Provides functionality to display directory structures in tree or indented formats.
   - **TagFactory.cs**: Factory class for creating HTML elements based on tag names.
   - **TreePrinter.cs**: Utility class for rendering tree-like and indented representations of the HTML model.
   - **Session.cs**: Manages multiple editors within a single user session.
   - **SessionStateManager.cs**: Handles the persistence of session states to and from storage.
   - **HtmlEditor.csproj**: Project file defining build configurations, dependencies, and other settings for the main application.
2. **HtmlEditor.Tests/**
   Contains the test suites for the HTML Editor application.
   - **HtmlEditorCommandTests.cs**: Integration tests verifying the correct execution of user commands and their effects on the session state.
   - **HtmlEditorTests.cs**: Unit tests focusing on the core functionalities of the `HtmlEditor` class, ensuring reliable operations.
   - **HtmlEditor.Tests.csproj**: Project file defining build configurations, dependencies, and other settings for the test project.
3. **.temp/**
   A hidden directory used for storing session state files. This directory is crucial for session persistence, allowing users to resume their work across application restarts.

- **session_state.json**: JSON file containing serialized session data, including open editors, active editor status, and visibility settings.

4. **commands.txt**
   A sample commands file used for testing and demonstrating the editor's capabilities. It contains a sequence of commands that can be executed automatically to verify functionality.
5. **README.md**
   A markdown file providing an overview of the project, including installation instructions, usage guidelines, and other essential information. This manual can serve as a more detailed counterpart to the `README.md`, offering in-depth documentation.
6. **setup.sh & setup.bat**
   Shell scripts for automating the installation process on Unix-based systems (`setup.sh`) and Windows (`setup.bat`). These scripts streamline the setup by cloning the repository, restoring dependencies, building the project, and running tests.
7. **Other Project Files**
   - **.gitignore**: Specifies files and directories to be excluded from version control, such as build artifacts, temporary files, and sensitive information.
   - **LICENSE**: Contains the licensing information for the project, outlining usage rights and restrictions.
   - **Documentation/**: (Optional) Additional documentation files or resources to support users and developers.

Notes:

- The project adheres to standard .NET project structures, promoting familiarity and ease of navigation for developers accustomed to .NET conventions.
- Proper organization of files and directories facilitates collaborative development, enabling multiple contributors to work on different components without conflicts.

Appendix E: Troubleshooting

Common Issues and Solutions

1. **Failed to Restore Dependencies**
   Symptoms:
   - Errors during the `dotnet restore` process.
   - Missing or incompatible packages.

   Solutions:

- **Internet Connection**: Ensure that your system has a stable internet connection to download NuGet packages.
- **.NET SDK Installation**: Verify that the correct version of the .NET SDK (net9.0) is installed. You can check your installed SDKs with:

  ```
  dotnet --list-sdks
  ```

- **Clear NuGet Cache**: Sometimes, corrupted cache can cause restore failures. Clear the cache using:

  ```
  dotnet nuget locals all --clear
  ```

- **Check `.csproj` Files**: Ensure that the `.csproj` files do not contain typos or invalid package references.

2. **Spell Checker Not Working**

   Symptoms:

   - The `spell-check` command fails or does not mark any errors.
   - No output is produced after running `spell-check`.

   Solutions:

   - **Internet Connection**: Confirm that your system can access `https://api.languagetool.org`. A lack of connectivity will prevent spell checking.
   - **API Availability**: Check the LanguageTool API status to ensure that the service is operational.
   - **Firewall Settings**: Verify that firewall or network policies are not blocking outbound HTTP requests to the LanguageTool API.
   - **Error Logs**: Review the console output for any error messages related to spell checking, such as API request failures or parsing issues.

3. **Unique ID Constraints Violated**

   Symptoms:

   - Commands like `insert` or `append` fail due to duplicate IDs.
   - Elements cannot be retrieved or manipulated using their IDs.

   Solutions:

   - **Ensure Uniqueness**: When assigning IDs to elements, ensure that each ID is unique within the document. Avoid reusing IDs.
   - **Validate Input**: Before executing commands that add or modify IDs, verify that the new ID does not already exist.
   - **Use Descriptive IDs**: Adopt a naming convention that reduces the likelihood of duplicate IDs, such as prefixing with element types or numbers.

4. **Undo/Redo Not Functioning Properly**

Symptoms:

- The `undo` or `redo` commands do not revert or reapply changes as expected.
- The editor throws errors when attempting to undo or redo.

Solutions:

- **HistoryManager State**: Ensure that the `HistoryManager` is correctly saving states after each edit operation. Any failure in state saving will disrupt the undo/redo functionality.
- **Command Exclusions**: Verify that only editing commands affect the history stack. Display and input/output commands should not be recorded in the history.
- **Session Integrity**: Check that the session is maintained correctly, with the active editor and its history accurately tracked.
- **Error Logs**: Look for error messages related to undo/redo operations in the console output to identify specific issues.

5. **File Save/Load Errors**

   Symptoms:

   - The `save` command fails to write to the specified file.
   - The `read` command cannot load the specified file.
   - File contents are corrupted or not as expected after saving/loading.

   Solutions:

   - **File Permissions**: Ensure that you have the necessary read/write permissions for the target directories and files.
   - **Valid Paths**: Verify that the file paths provided to `read` and `save` commands are valid and correctly formatted.
   - **File Locking**: Ensure that no other application is locking the file being accessed, preventing read/write operations.
   - **Disk Space**: Confirm that there is sufficient disk space available for saving files.
   - **Error Logs**: Review the console output for specific error messages related to file operations.

6. **Command Not Recognized**

   Symptoms:

   - The editor responds with "Unknown command" or similar messages when entering valid commands.
   - Typographical errors in commands result in unexpected behavior.

   Solutions:

   - **Command Syntax**: Double-check the syntax of the commands to ensure correctness. Refer to the Command Reference section for accurate usage.
   - **Case Sensitivity**: Commands are case-insensitive, but parameters (like `id` values) are case-sensitive. Ensure that you are using the correct casing.
   - **Help Command**: Use the `help` command to list available commands and their correct syntax.

- **Update Editor**: Ensure that you are using the latest version of the editor, as newer commands may not be recognized by older versions.

7. **Session Persistence Issues**

   Symptoms:

   - The editor does not restore the previous session upon restarting.
   - Open editors are not listed, or the active editor is not correctly set after a restart.

   Solutions:

   - **Session State File**: Ensure that the `.temp` directory exists and contains the `session_state.json` file. This file is crucial for restoring session states.
   - **File Integrity**: Verify that the `session_state.json` file is not corrupted. If it is, consider deleting it and starting a new session.
   - **Proper Exit**: Always use the `exit` command to close the editor, ensuring that the session state is saved correctly.
   - **File Permissions**: Confirm that the editor has write permissions to create and modify the session state file.

8. **Editor Initialization Failures**

   Symptoms:

   - The `init` command fails to reset the HTML structure.
   - Subsequent commands behave unexpectedly after initialization.

   Solutions:

   - **Command Sequencing**: Ensure that the `init` command is used appropriately, and be aware that it resets the document, potentially discarding unsaved changes.
   - **Error Logs**: Look for error messages in the console output that indicate why the initialization failed.
   - **Manual Verification**: Use display commands like `print-tree` to verify that the HTML structure has been correctly initialized.

Support

For further assistance, consider the following resources:

- **GitHub Issues**: Submit issues or feature requests through the project's GitHub Issues page.
- **Documentation**: Refer to this manual for comprehensive guidance on installation, usage, and troubleshooting.
- **Community Forums**: Engage with other users and developers on relevant forums or discussion boards to seek advice and share experiences.
- **Contact Information**: If the project maintainer has provided contact details, reach out directly for personalized support.