Andrei Fangli
Class 248

Mini project 2 –Markdown Parser

1 Andrei15193.MarkdownParser

Markdown is a text-to-HTML language allowing users (primary dedicated to web writers) to write posts in a very readable format which can be transformed into a valid HTML. Basically it eliminates the need to know and understand how HTML works and instead one must use an intuitive format to get the same results.

Besides removing the HTML knowledge it also makes it safer to Content Management Systems as administrators that use Markdown or other text-to-HTML languages instead of HTML do not need to worry about potential attacks. Raw HTML can contain JavaScript that can be used to crash a web application, redirect to unwanted sites or other undesired actions. Because Markdown is HTML-free (the author does allow inline HTML to be used, however an administrator may forbid it and only allow “pure” Markdown) there is no way ill intended users to attack the application through JavaScript or other HTML exploits, however they can hack the application through links (e.g. the user clicks a link that says “Wikipedia – etc.” and it actually leads to an action that cancels that users account).

There are a number of Markdown parser implementations already, some of them are in C, some in PHP however there do not seem to be any available for .NET. Andrei15193.MarkdownParser comes to solve this problem and enable the use of Markdown in .NET applications as well. This is an open-source project hosted at https://bitbucket.org/Andrei15193/markdownparser/. At the moment the project the parser can translate Markdown formatted text into HTML using a Regex based parser (regular expressions are used to describe Markdown formats using capture groups to obtain different values for different HTML elements and attributes).

2 Parser Architecture

Text transforming tools go well on a data-flow architectural style, using pipes-and-filters architectural patterns because the source text goes through a series of transformations at different levels until it reaches its final form. Each step takes the input of the previous step (filter), performs some action and sends it to the next step (filter). The intermediary forms are required to be read one way or another (e.g. for reading text a TextReader can be used, for traversing a tree one can either use Visitor or provide an IEnumerator (and implementing the IEnumerable interface) that traverses the tree). These objects that facilitate reading are actually pipes as they are the bridge from the previous step (or input) to the next step (or output).

The Parser takes the Markdown formatted text as input and produces a tree structure representing the equivalent hierarchical form of the formatted text. Each node represents a Markdown element (e.g.: link, image, list, list item and so on). The tree can then be traversed to obtain equivalent HTML or obtain other equivalent formats (e.g.: Microsoft Open XML Word Documents (.docx)).

2.1 KEY COMPONENTS

A MarkdownNode being the base class for all types that represent Markdown elements (lists, paragraphs and so on). This is used mostly for validation. The Parser (obviously) that uses a number of MarkdownNode factories, basically they use a Regular Expression that describes what pattern the Markdown element must match and an operation taking the Regex match and descendant Markdown elements (if any) to construct the Markdown element instance. This freedom allows to solve dependencies between descendant nodes (e.g.: given a list with two elements in Markdown, if the two are separated by a blank line it will make the text of the 1st item to be wrapped in a paragraph element, for consistency this also happens to the text of the 2nd item because there is no 3rd item to tell whether the text of the 2nd to be wrapped in a paragraph or not).

The result of the parse is represented by the MarkdownRootNode subclass which represents the root of the entire Markdown tree. The root node can have only Markdown block elements (e.g. paragraphs, lists, blockquotes and so on).

The pipes-and-filters pattern in this case is simple, the input is provided as a string and is iterated by the .NET Regex engine by accessing characters by index (the engine source code was made public by Microsoft). The pipe between the input and the parser is the indexer. The parser is just a filter, it takes the Regex matches producing MarkdownNode instances through related factories that remove unnecessary characters (e.g. leading and trailing '####' for a level 3 heading characters are removed and only the text in between and on the same line is considered to be part of the header text).

To produce the equivalent HTML format one must only iterate the descendants of every node using the provided ReadOnlyList<> enumerator and for each node type provide code that transforms it to its equivalent HTML format. In this case the enumerator is a pipe and the translator is just another filter that removes the whole bunch of MarkdownNode instances to produce a string (or a HTML tree).

The main quality of this approach is that at any point another pipe and filter can be added to adjust the final output of the translation process because the parser does not simply takes a string and produces another string, it produces a tree of objects that can be iterated and interpreted in different ways.

3 PARSER DETAILED DESIGN

One of the Design Patterns used has already been mentioned but not really in the clear (as there is Factory Method and Abstract Factory). In this project Factory Method has been used, each Markdown element provider is in fact a Factory Method (actually a Markdown node specification that exposes an interface which in turn exposes an operation for creating the MarkdownNode. One may say that it is a strategy however it is a variant of the Factory Method as the intent is to create instances and not by an algorithm. Markdown node factories are not algorithms where one can be swapped with another and get the same result). Each implementation decides what concrete MarkdownNode to create.

As there are many flavors of Markdown out there (due to having a description that lacks details and most if not all rules are open to interpretation when it is found in different context) the Parser was thought to be easy to extend. Having Factory Methods that specify what a piece of text must match in order to be a given Markdown element and having the specific instantiation in the implementing Factory Method enables great flexibility. One must only implement the INodeSpecification<> interface and provide a Regex,

the actual Factory Method implementation, specify the types of nodes it can contain (e.g. MarkdownParagraphNode. MarkdownTextNode and so on) and it is done.

The Parser itself is a Template Method as the skeleton of the algorithm is written and only through some overrides a derived class can intervene and change the output (e.g.: the collection of MarkdownNode factories).

On the next page there is a class diagram (not complete as there is not much sense in describing every MarkdownNode, they become intuitive once the Markdown syntax is known) containing all key types and a few MarkdownNode subclasses.

Andrei Fangli
Class 248