Design Choice

General Class Structure

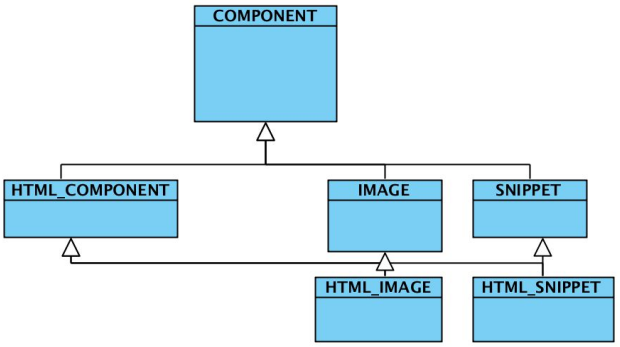

Figure 1 General Structure UML

For each class in the API will follow the structure illustrated above. It will inherit from either Markdown or HTML component and its own deferred class. This design allow easy extensions of MARKDOWN_COMPONENT in the future. It can be simply implemented by adding classes inheriting from MARKDOWN_COMPOMENT and the deferred class.

By restricting our Composites, that the leaves have to be HTML_COMPONENTS, we also can assure that no Markdown element makes its way into a HTML composite.

Abstract Factory Pattern

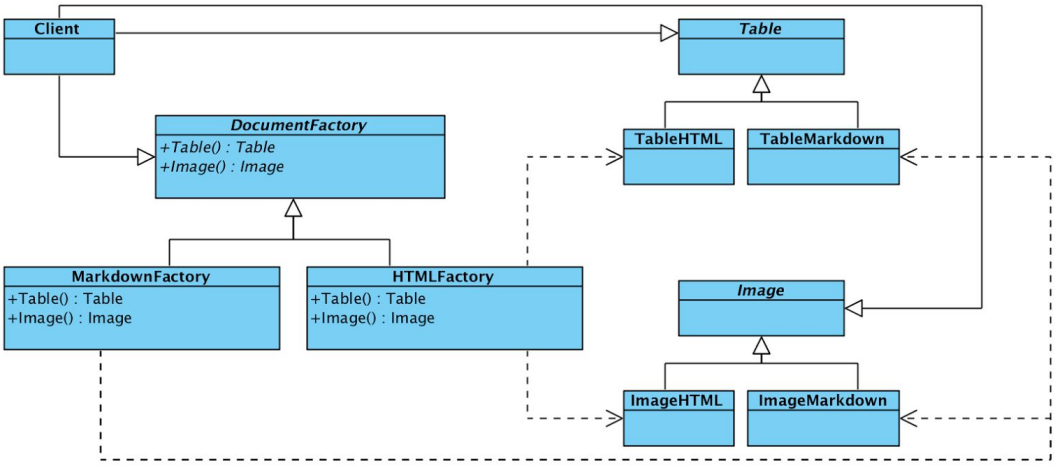

Figure 2 Abstract Factory UML

Support for HTML and Markdown

We used the abstract factory pattern to keep our program flexible so we can easily extend our library to markdown. To initialize the Markdown_Factory one just has to implement the same set of concrete classes as for the HTML_Factory. Once both Factories have been implemented, a user can switch very easy between the two output versions. By sticking to the abstract factory pattern, we can ensure that the factories do not mix in between. In this way we can guarantee that we either get a pure HTML or a pure Markdown document.

Composite Pattern

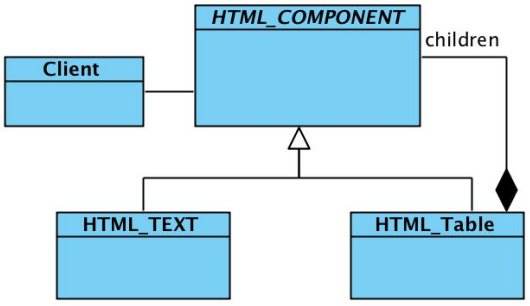

Figure 3 Composite Pattern UML

Our library supports encapsulation, which means for example that you can place a bold text and an image inside of a table. To implement this, we used the constructor pattern. All classes of the constructor pattern can be dedicated to one of two groups.

The classes table, list and paragraph are composite classes. As arguments they can take simple text or any class of the HTML_Factory. Theoretically, one can encapsulate an infinite number of classes, but we assume out of imaginable use cases that there will be no more than three encapsulations.

The classes image, text, title, link and snippet are leaf classes and cannot take any other class as input. They accept only a string as input and depending on the class some integer parameter as well.

Visitor Pattern

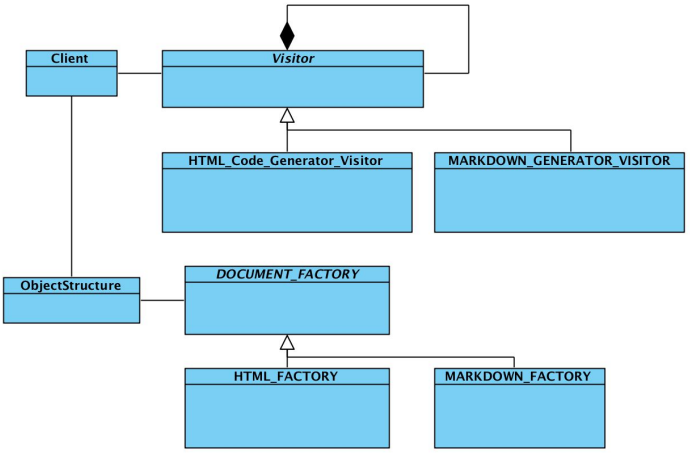

Figure 4 Visitor Pattern UML

The classes of the HTML_Factory are not doing anything except of generating objects and it inherits DOCUMENT_FACTORY, which is the deferred class for both HTML and Markdown modes. In our API, we only implement HTML_Code_Generator_Visitor, which will be responsible for generating the corresponding HTML code for each objects .

List for visitor

We decided to implement a list to allow the user to order the elements of the HTML page as he wishes to. This way we do not force him to create the objects in a very restricted manner. The user is free to decide if he wants to first create all the innermost objects of all elements or to first implement just the objects of the first element and then moving forward to the second. Therefore the user can create the objects without having to think about what will displayed in which order or if it is nice programming style. The only restriction is that all object that will be placed inside an element must be created above the element.
When the user is finished with creating his objects he can add them in the order he wishes to the list. If he wants to make changes in the order he can simply change the order of elements in the list without having to change the whole creating part. If he thinks for example that an image that was placed inside a table should be shown on its own and the table should not be shown, all he has to to

is changing the list, more precisely delete the name of the table from the list and replace it by the name of the image

When the program is finally running, the visitor visits all the generated objects in the right order of the list and extracts the HTML code that is stored inside the object. Finally it puts all that code together and stores it in a string.

Set up

The set up allows the user to define all objects before filling them with content and encapsulate them into another. For the setup is separated from the make feature, the user has an short overview of all the objects the user can use.