University of Zurich[UZH]

Design Document for

<YODA>

YOUR OUTRIGHT DOCUMENT ASSEMBLER

Daniela Flüeli
Joel Barmettler
Marius Högger
Spasen Trendafilov

Supervision:
Prof. Bertrand Meyer

Software Engineering – Department of Informatics
University of Zurich

December 17, 2017 | Zurich, Switzerland

Table of Contents

List of Figures

1 Introduction

This document describes the structure and design of YODA. For the structure, YODA uses multiple design patterns, for these patterns we describe why YODA uses them by pointing out the benefit of the pattern towards YODA. In some cases, we extend our reasoning by naming some other possible design patterns which could have been used instead and explain why we decided against them.

2 Class Diagram

Here we provide the full class diagram of YODA. A more detailed view of different parts of the class

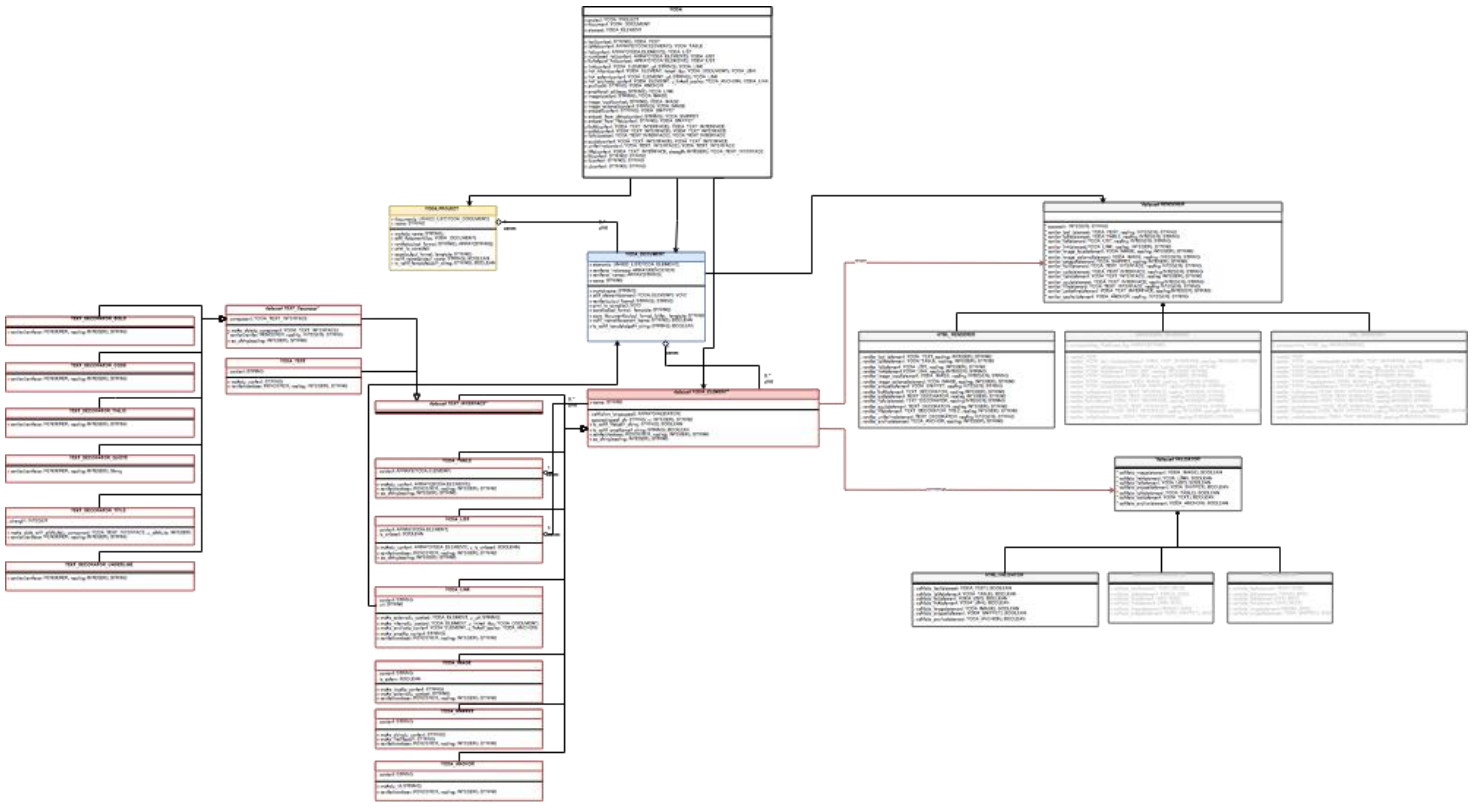

diagram can be found in the chapter 3 “Used Design Patterns”

FIGURE 1: CLASS DIAGRAM

3 Used Design Patterns

In this section we list all design patterns we used in YODA’s design. For each design pattern we specify the involved classes of YODA and give an explanation why we used it.

3.1 Composite

3.1.1 Description

Composites are used when we want to have the objects in a tree structure. Composites are especially useful if recursions are occurring and if the client wants to be able to uniformly treat single objects and compositions without having to know that there could be more sub objects within.

3.1.2 What YODA uses Composites for

YODA supports elements such as table and list. Those elements act like containers for other elements. In the simplest way, a table is a container for texts, but it can also contain images or other elements. YODA solves this so-called nesting with the composite. Since YODA does not want to check if an element contains another element or not, individual objects are handled the same way as compositions.

3.1.3 How YODA uses Composites

All elements which do allow nesting (YODA_TABLE, YODA_LIST) are implemented as composites. YODA_ELEMENTS represents the component class of the general, tree-like, composite-structure. This means that a composition of YODA-ELEMENTS gets handled like an individual YODA_ELEMENT. The elements that do not support nesting (YODA_TEXT, YODA_LINK, YODA_IMAGE, YODA_SNIPPET) are the "Leaves". These "Leaves" can be contained in composites but can't contain any other elements them self.

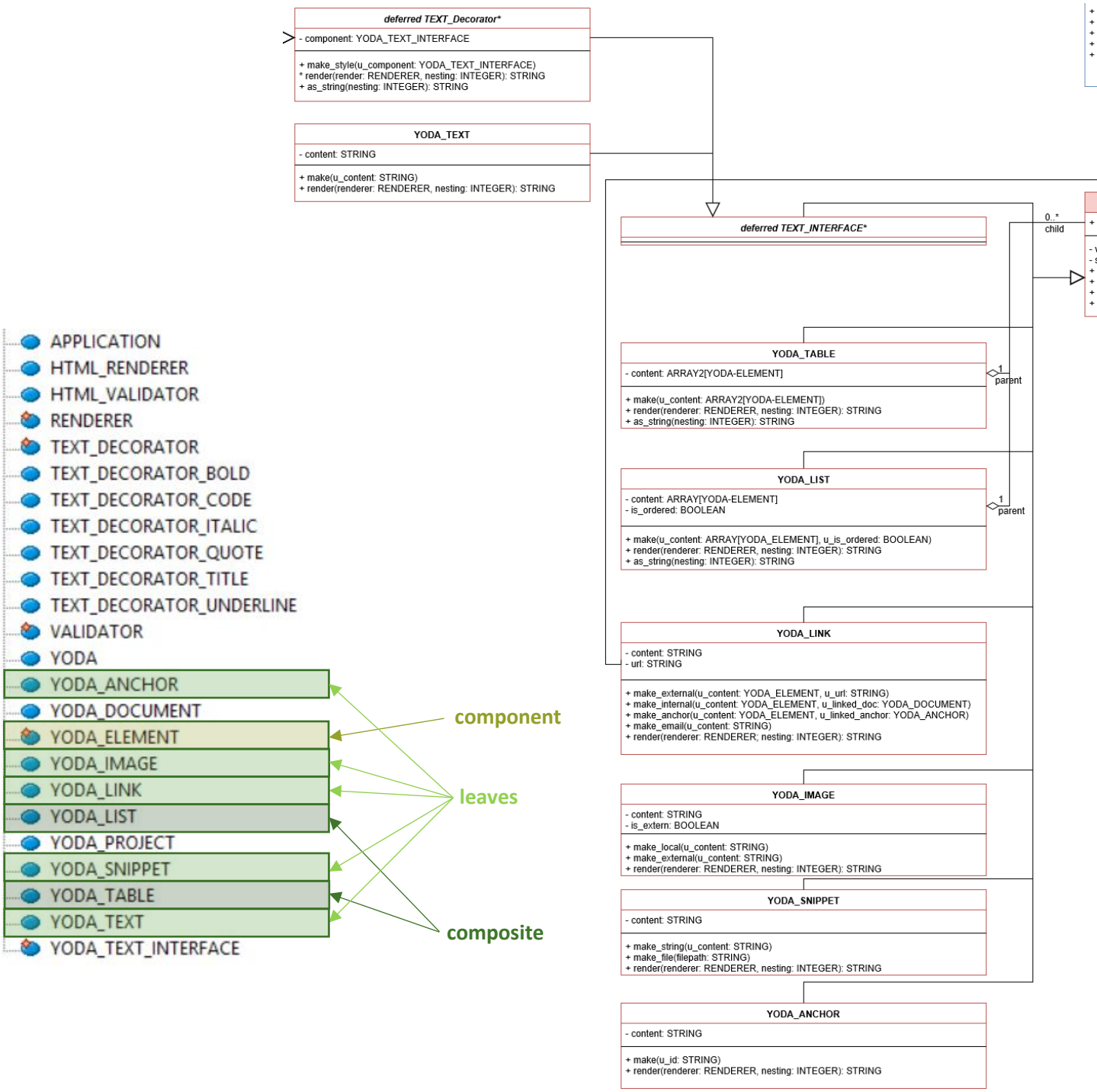

FIGURE 2: COMPOSITE

3.1.4 Alternative approaches and why YODA does not use them

Since YODA supports nesting there's almost no way around composites. One obviously very bad solution would be to create a new element class for every nesting layer supported. This would then hard code the deepness of the nesting and would create many classes which then are rarely used. Since YODA wants to be flexible composites are the right way to go.

3.2 Decorator

3.2.1 Description

The Decorator is a pattern that can be used if you want to add additional and optional functionalities, responsibilities, or attributes to an object. The Decorator's structures states that the core object inherits from an abstract interface. The Decorators also inherit from the same abstract interface.

3.2.2 What YODA uses Decorators for

YODA uses Decorators for the styling of the text-element. HTML and Markup languages support some basic text styling such as making the text bold, italic or underlined. For these stylings YODA uses Decorators such that any text can be theoretically wrapped with any number of decorators. In Addition to this basic styling YODA uses the decorators for tiles, quotes and displayed code, which are handled as own elements in most of the languages. Using decorators for such styling it is very easy to add and support more basic styling and from YODA preconfigured stylings which enhances the user-experience. In our opinion it seemed more fitting to make thigs like title a decorator, since a title in HTML format does not differ in great extent from a normal text beside having one more tag. Looking at the title representation of a title in maktdown gives an even better insight into that argumentation. Also, we think that having for titles own elements would be a bit strange and having it as a decorator leaves some space open for the user to create an instance of a text once and create multiple different variations of it using decorators. However, it can clearly be said that this is more like a matter of taste than anything else

3.2.3 How YODA uses Decorators

To use decorators in a proper way, YODA has a deferred TEXT_INTERFACE* class which acts as the abstract interface for both, the core YODA_TEXT class and the decorators (TEXT_DECORATOR*). To use the same functionalities (validation, rendering) as all the other YODA-Elements, the TEXT_INTERFACE* inherits from the YODA_ELEMENT* class.

On the other end YODA uses a deferred TEXT_DECORATOR* class where all the specific decorators inherit from. Therefore, it is easy to add new decorators. The key functionality of the decorator is to edit the rendering of the YODA-Text. Each decorators task is to extend the rendering in a way that it renders what's inside itself (YODA_TEXT) and then adds its own tag-strings around the returned string. YODA supports the rendering of multiple Markup languages and since we consider creating a separate decorator for each output-language as bad design, we use the visitor pattern for the rendering inside the decorators. Basically, the rendering the works the same as in the YODA-Elements. A precise description of that mechanism can be found in the chapter about the visitor. YODA does this in that manner so that all the implementation of the rendering is centralized in the same space, in one class, specific to the output language.

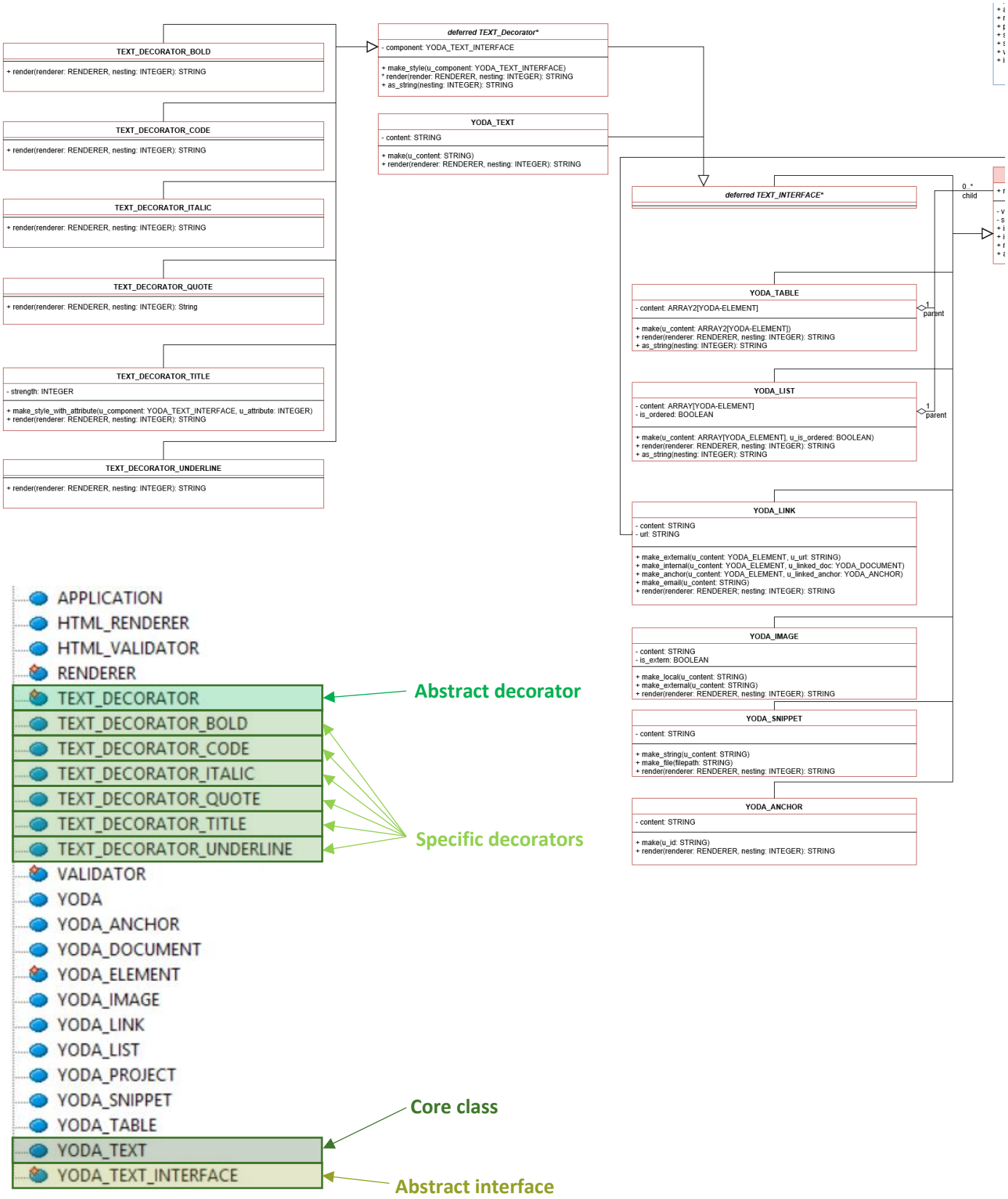

FIGURE 3: DECORATOR

3.2.4 Alternative approaches and why YODA does not use them

Instead of using decorators YODA could use Boolean attributes in the YODA_TEXT class. The client then could set the Booleans with different functions. When the Boolean is set to true the renderer then would render the styling tag. We considered this solution as very poor since having a lot of attributes in a class just for storing a Boolean can easily lead to a mess, especially when more styling is added later. Another approach would be to just create new YODA-Elements for each of the styling since they then would support having another YODA-Element inside them due to the composite pattern the YODA-Elements use. Since the YODA-Elements already use the visitor for the rendering the same render-functions as in the final design could be used. We went for the decorator however since the decorator pattern does better reflect the actual use of the styling which leads to a better understanding of the class structure.

3.3 Visitor

3.3.1 Description

The visitor design pattern describes an architectural scheme that separates an operation from an object structure on which it operates to solve the double dispatch problem. This is possible by the elegant collaboration of the three main participants, the target classes, the client classes and the visitor classes. Assume, a client needs to perform a certain operation on a concrete object. For this reason it calls a so-called accept() method on the concrete object of the visitable class. The desired operation is specified by the provided argument which is a concrete visitor. The concrete object itself calls the corresponding visit_class_x() method on this concrete visitor, providing itself as current as the argument. Finally, this method performs the desired operation on the concrete object. Note, the execution of the desired operation performed depends on the type of the concrete object. The concrete visitor has for each type a distinct implementation of the operation ready. This architectural scheme of the visitor pattern ensures high extendibility towards other operations, but bad extendibility towards new object types.

3.3.2 What YODA uses Visitors for

YODA uses the visitor pattern for the rendering of the different YODA-Elements in all the supported languages. After discussing the design alternatives below and due to the fact, that the concrete elements are different with respect to the different languages in terms of the rendering only, we finally became convinced that the visitor pattern offers the best design with respect to the rendering of the different YODA-Elements in all the supported languages. This design using the visitor pattern allows extendibility towards other output types / languages. For a new output type XML for instance, we only need to implement a XML_RENDERER. The implementation of the YODA_DOCUMENT class, the YODA_ELEMENT as well as the concrete element classes remain unchanged. The trade-off using the visitor pattern is a bad extendibility towards new visitables, which would involve updating every visitor that's already implemented. But since in YODA the visitables, the concrete elements, are not intended to change there is nothing that argues against this design. We took the liberty to rename the ***Accept()*** method to ***Render()***. We thought twice about it, but since no other operations than the rendering are possible, we think ***Render()*** is a good even more meaningful name.

3.3.3 How YODA uses Visitors

In this subsection we present our implementation of the rendering using the visitor pattern. These are the classes of YODA that are part of the visitor pattern:

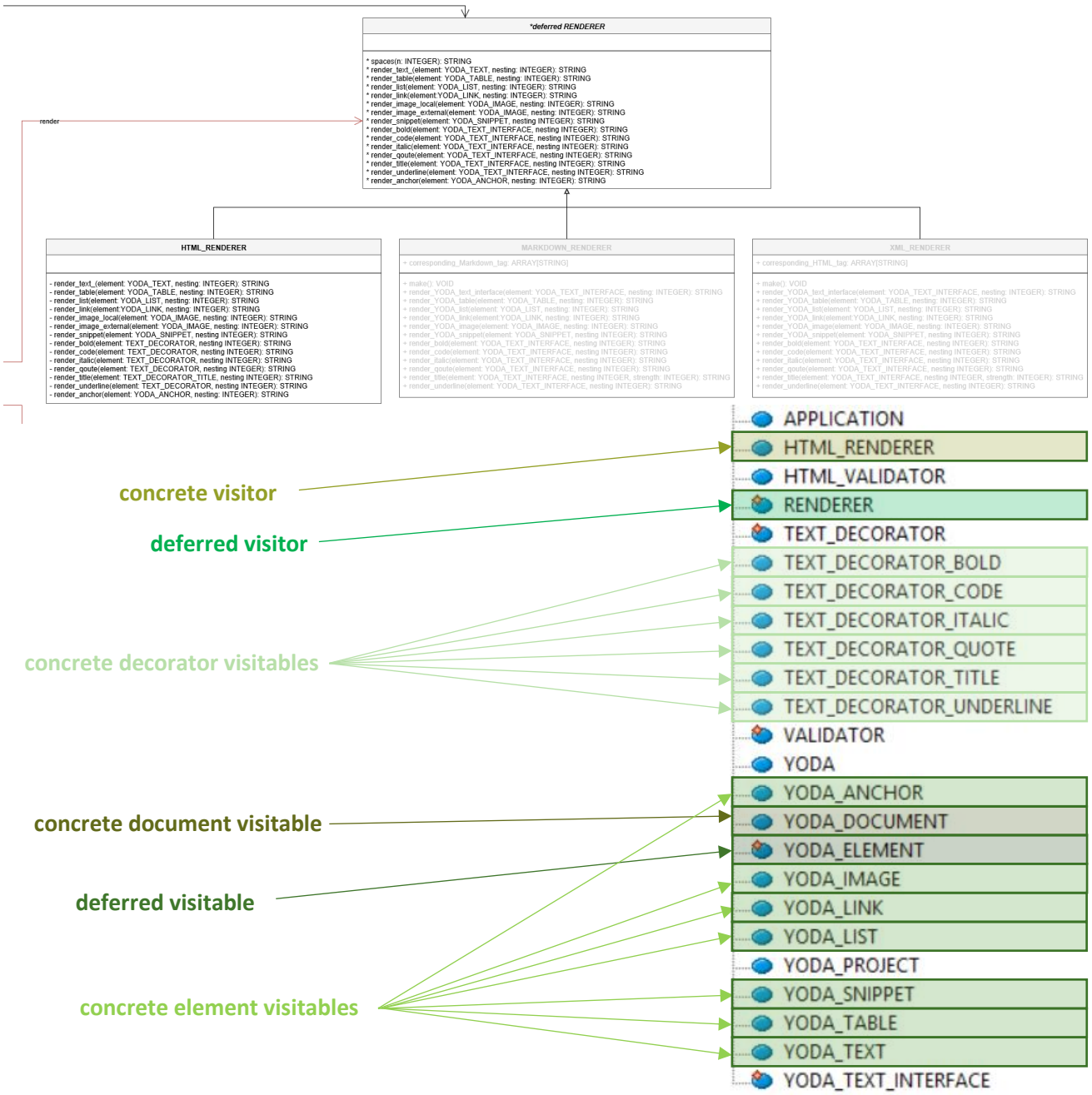

FIGURE 4: VISITOR

3.3.4 Alternative Approaches and why YODA does not use them

The most straightforward, but very naive, implementation to render the different YODA-Elements in the different supported languages would be a deferred procedure called render_to_X() for each language in the YODA-ELEMENT class and redefined procedures in all of its children, the concrete elements, as simulated in the code below.

```
deferred class
    YODA_ELEMENT

    feature {ANY}
        render_to_html ()
            do
            deferred
            end

        render_to_markdown ()
            do
            deferred
            end
        --...
        --...
        --(für jede Sprache)
```

```
class
    YODA_LIST inherit
        YODA_ELEMENT
        redefine render_to_html end
        redefine render_to_markdown end
        --...
        --...
        --(für jede Sprache)


    feature {ANY}
        render_to_html ()
            do
                --render list as HTML
            end

        render_to_markdown ()
            do
                --render list as markdown
            end
        --...
        --...
        --(für jede Sprache)
```

FIGURE 5: ALTERNATIVE DESIGN 1 FOR VISITOR

```
class
    YODA_DOCUMENT

    feature {ANY}
        elements: LINKED_LIST[YODA_ELEMENT]

    feature {ANY}
        render_to_html ()
            do
                -- for each element in elements element.render_to_html
            end

        render_to_markdown ()
            do
                -- for each element in elements element.render_to_html
            end
        --...
        --...
        --(für jede Sprache)
```

FIGURE 6: ALTERNATIVE DESIGN 2 FOR VISITOR

We are convinced that this is a very unlovely implementation. This code is on the one hand highly redundant, the procedures in the YODA_DOCUMENT class are almost the same. On the other hand, it leads to extendibility problems. Every concrete element must know every supported language. Support an additional language would demand adding a corresponding procedure to all the concrete elements, as well as to the YODA_DOCUMENT and the YODA-ELEMENT. Although this approach is feasible, we never took it into account for our design due to the mentioned disadvantages.

Another approach would be the use of the abstract factory pattern. The Image below shows how the factory structure of Yoda would look like. There is a concrete factory for every language. In addition, every element has a concrete subclass for every language.

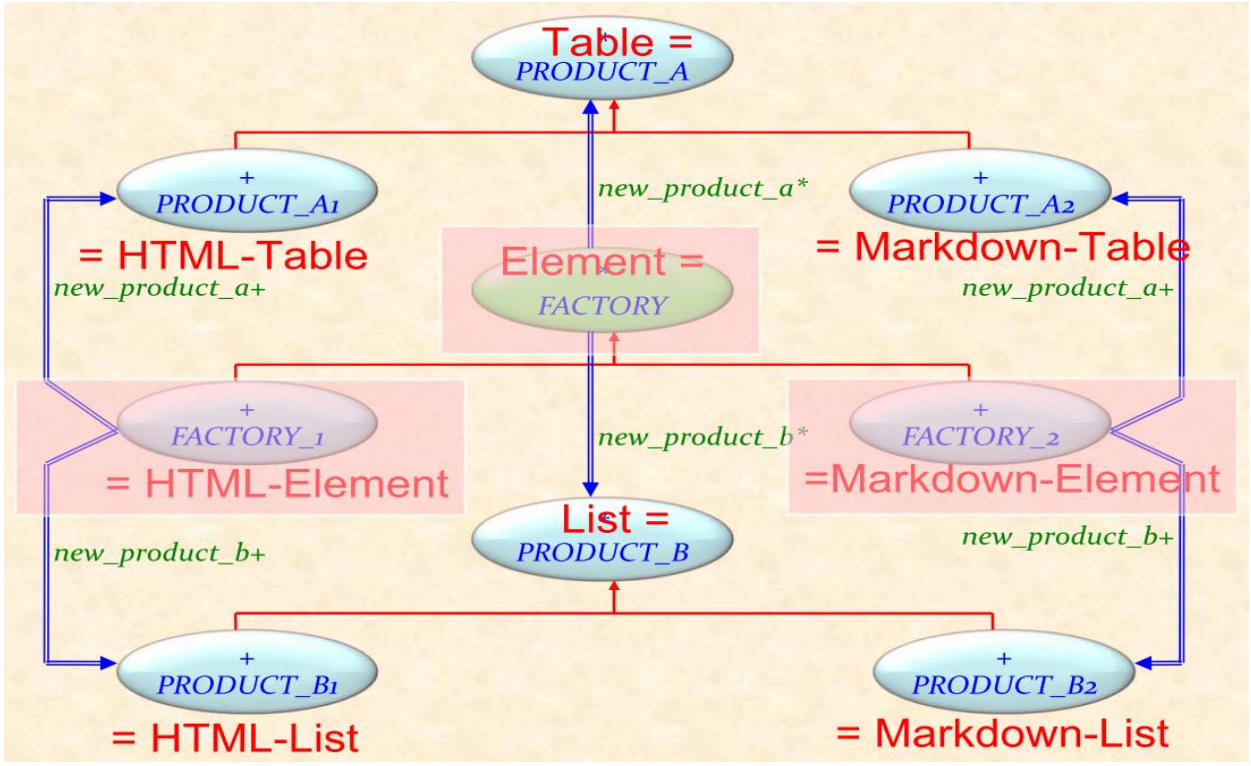

FIGURE 7: ALTERNATIVE DESIGN: ABSTRACT FACTORY

At first sight, this approach looked reasonable, since YODA consists of multiple families of related objects, which are only used together. We thrashed this implementation alternative out. The main argument against this implementation is a matter of memory. The user of YODA should be able to decide the output type after the creation of the elements. This is not that straightforward to achieve by such a design using the abstract factory pattern. It would imply that the elements the user creates must preliminary be of a language type “neutral” or so. After the user has chosen his desired output type, there must be created a new element of this output type for each element of type neutral. The neutral elements are not used anymore and remain in storage of no avail.

A third alternative would be to outsource the render functionality in a separate class, RENDERER. The renderer would have for every supported language X a render_to_X(e: ELEMENT) procedure. Within each of these procedures there is for each concrete element Y an if-attached-{Y}-e-then query. This approach would not clutter the concrete element classes with procedures, but instead the renderer class with conditionals. In addition, the benefits of dynamic binding would be lost.

3.4 Command

3.4.1 Description

The Command pattern is used whenever certain requests shall be performed on a certain client. The Command needs an encapsulated request in an object (or agent, in Eiffel) as well as a client on which the request is performed.

3.4.2 What YODA uses Commands for

Every specific YODA_ELEMENT needs to validate itself according to certain validation rules, that are different for each supported output language. For this validation purpose, YODA has its own VALIDATOR class with a subclass for every supported language, which contains validation functions for every specific YODA_ELEMENT. This means that every YODA_ELEMENT must validate itself with the corresponding validation-function in each of the language-validators. YODA uses a modified version of the command

pattern to construct a shared function between all the YODA Elements that validates all languages in one step.

3.4.3 How YODA uses Commands

The Command pattern used in YODA differs from the traditional implementation for different reasons. First, requests do not need to be stored in new, implemented classes, but Eiffel handles them as agents. Second, we only need the command for validation purposes, so we directly included the commander inside the abstract YODA_ELEMENT class for easier access along its children objects. When instantiating a new, specific YODA_ELEMENT, the element needs to confirm whether it corresponds to the YODA_RULES. This is done directly in the require-part of the make-process. An example for such a requirement for a YODA_TEXT is that the text shall have at least one character. But in addition to our constraints, the users input shall as well be in harmony with the supported features of the language. So, in addition to our own constraints, the make function also asks all the supported languages whether this element conforms the specific language constraints. Such a specific language constraint for the YODA_TABLE in combination with Markdown may be: The table does not contain a YODA_ELEMENT of Type YODA_LIST, because Markdown does not support lists. Important: This is just an example! In fact, lists in tables will be featured in YODA, they are just represented differently. But back to the validation. So, the make function shall call all the languages and ask whether the current object is valid there. So, what the make-function does is calling the validation_langauge.for_all feature do validate the object: It passes the current object and the corresponding validation-function to the commander (for_all). The commander then iterates through all the supported language validators (HTML_VALIDATOR, MARKDOWN_VALIDATOR, XML_VALIDATOR) that are stored inside the validation_langauge array, calls on each of them the provided validation function (HTML_VALIDATOR.validate) with the passed object as an argument, on which the validation will be performed (HTML_VALIDATOR.validate(element)). As an example, the table call will look like this (pseudocode, see src for Eiffel code):

table.make calls validation_langauge.for_all(current, validate_table)

The commander for_all iterates through the validators and calls first HTML_VALIDATOR.validate_table(element), with element being the table that called the validator. Each individual VALIDATOR will return True or False, depending whether the element is confirming with the languages rule for that specific object. The validator commander will return True iff all VALIDATORS returned True. In the end, the table.make's call of validation_language.for_all received True or False and, accordingly, lets the creation of a Table pass or gives an Error.

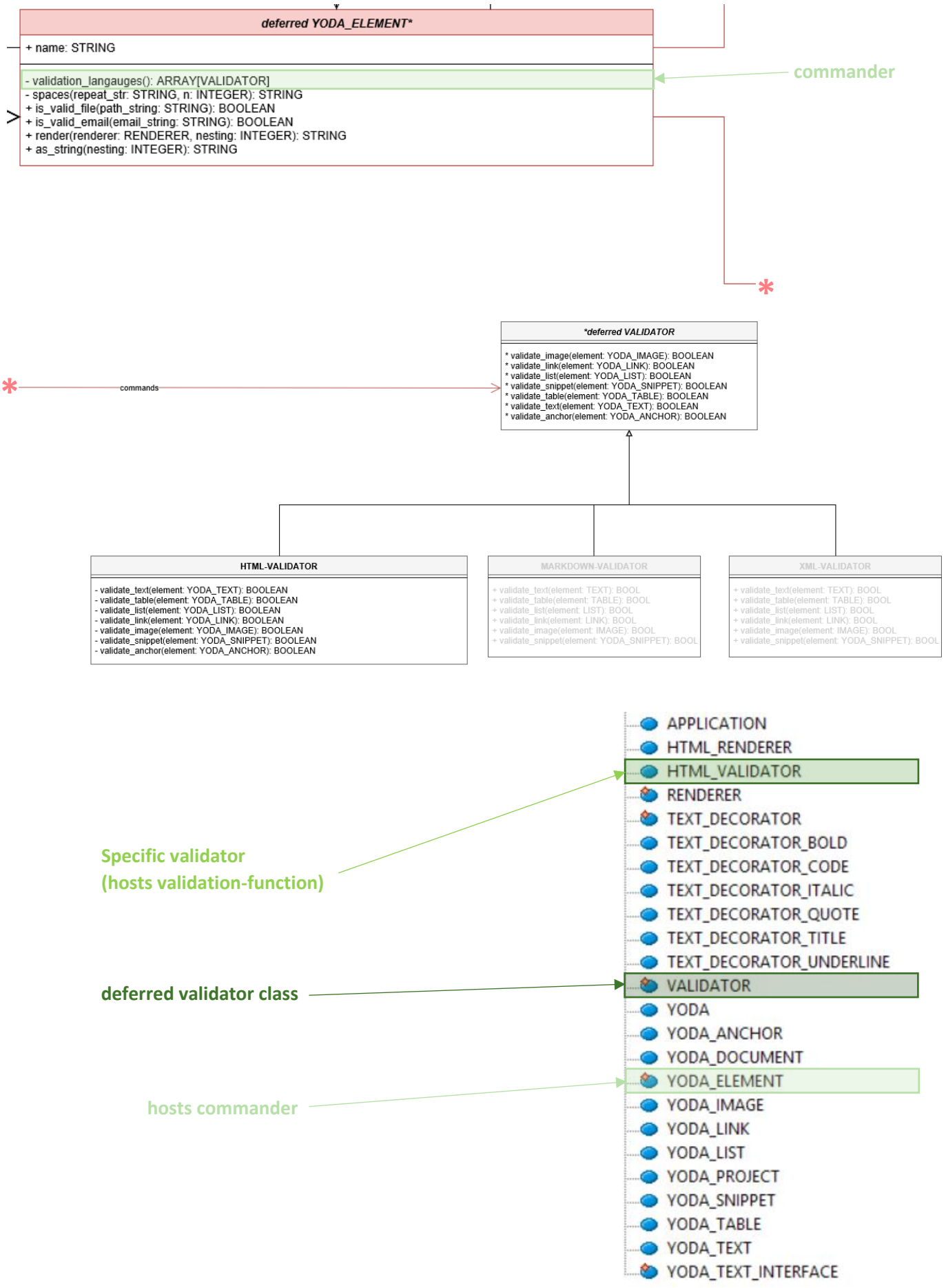

FIGURE 8: COMMAND

3.4.4 Alternative approaches and why YODA does not use them

Instead of using the command-like pattern to validate all languages per element, we could also have made a validation function for each specific element. This would imply that Table, List, Text etc. would all have individually implemented validation functions, that would create instances of all languages, loop over them and validate them with the specific function. So, the functions would do the same thing as the for_all commander, but a lot more functions would be needed and changing the validation process would be a pain, cause every validation function of each element would have to be changed. So, what we did is just “outsourcing” this loop to the commander and handed the only thing that differs the validation process as an argument: the for_all feature.

3.5 Factory

3.5.1 Description

The Factory or Factory Method is intended as an interface for the creation of objects. One of the aspects of the Factory Methods is that the decision which class to instantiate is still made in the subclasses, not in the interface.

3.5.2 What YODA uses Factories for

YODA uses the Factory only as an interface for the user for the creation of the object so that the class structure is hidden. Like that YODA provides an easy to use function which handle all construction and the handling on the classes which shall be accessed. In the end the user only must know which function he can use, all information about class names and intern creation-function are hidden. The factory-function YODA provides use intuitive naming for a better usability for the user. Since the aspect of letting the subclasses decide which object should be instantiated is not used in YODA we call this pattern just “Factory” instead of “Factory Method”.

3.5.3 How YODA uses Factories

YODA uses the YODA-class to set up the factory-functions. Like this all factory-function are accessible the same way. The user can use code like:

```
yoda.quote(yoda.text("May the Force be with you!"))
```

This in far easier to than accessing the quote decorator and the text-element class, which would look like this:

```
create {TEXT_DECORATOR_QUOTE}.make_style(create {YODA_TEXT}.make("May the Force be with you!"))
```

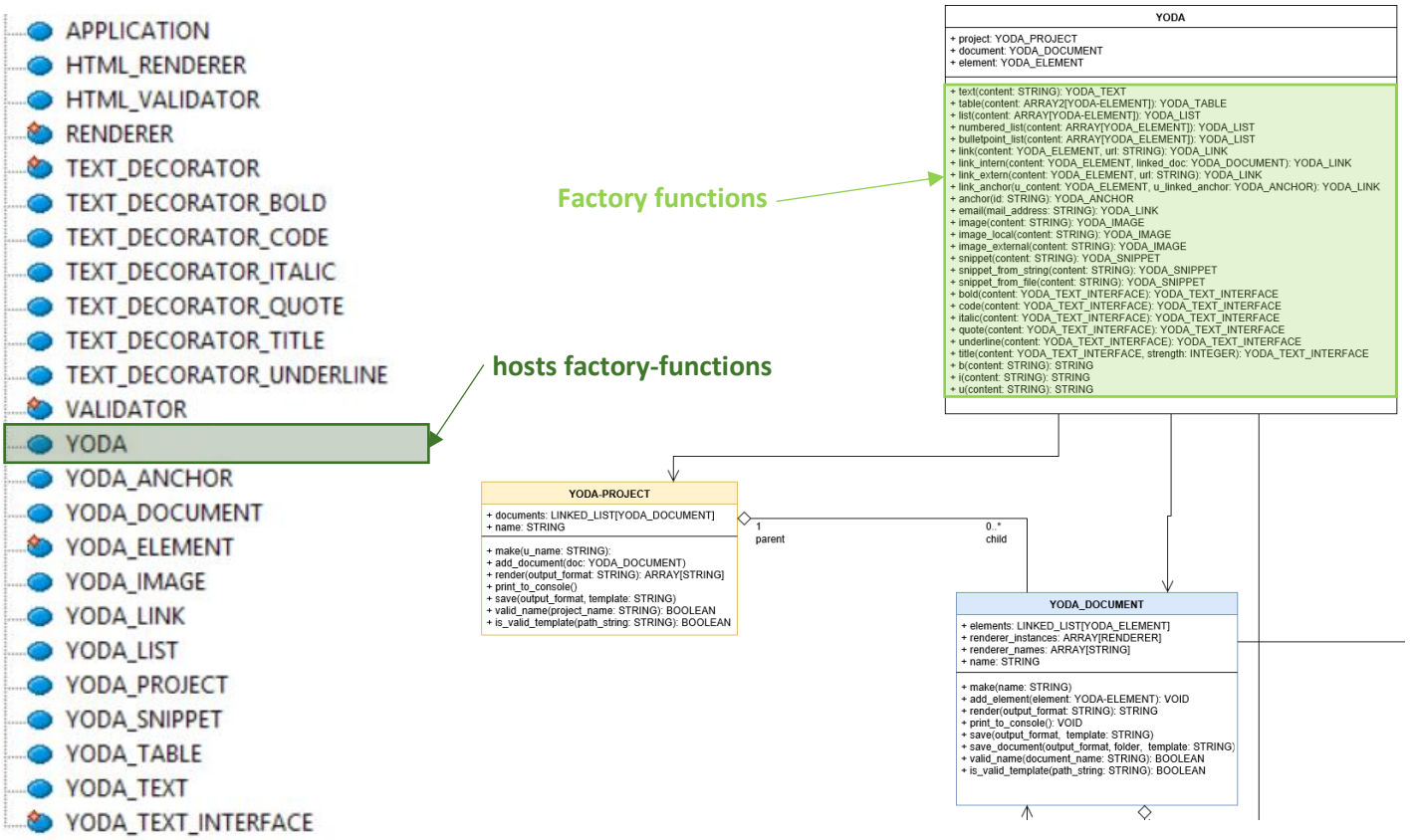

FIGURE 9: FACTORY

3.5.4 Alternative approaches and why YODA does not use them

The simplest approach would be to just not use Factories; however, the example above should give a good impression why YODA uses them to enhance the user experience.

4 General Design decisions

In the following subchapters, we'll try to explain some of our most important design decisions, what our general thoughts were, with what solutions we were coming up during the solving-process and what trade-off we decided to take.

3.5 Validation

For every supported output language, there's a corresponding VALIDATION class, which contains a validation function for every supported element. As explained in the Command pattern description, on creation, each element validates whether the user given constellation of data matches the requirements for each output language. This process shall ensure that the user is never able to create a certain element that will later cause troubles when rendered – so for every renderer, there is a validator beforehand that checks whether the renderer will be able to handle the input constellation, and only if he does, lets it pass without an error. This process of validation on creation allows the user to freely assemble his document with elements, as long as no errors occur on creation of the element, he can be sure that all elements will render nicely. He is free to choose in the very end what output language he likes to have, like Markdown or HTML, without worrying about the look and feel of the produced output.

This design decision has the drawback that YODA can offer the subset functionalities of all the supported output language, and as soon as one feature is not supported by just one of the selectable output language, the feature cannot be used at all. This further implies that YODA would have decreasing

functionality with increasing support of new output languages. This would be a deadly strike for a library, every update would be feared by the user because it may cause the previously written code to crash.
Therefore, we extended the validation process by not just looking whether the element fits the language-standards, but if not, making it fit later in the renderer. This means that, as long as it is somehow possible to re-arrange the users input in some way that the language can display its wished, the elements gets accepted by the validator, but may be changed later in the renderer to fit the language. Let me give you an example: The user creates a new text element, that contains the substring "</html>". The create procedure of YODA_TEXT is totally fine with this, nothing is violating the YODA rules cause it is a valid string. But it is clear that a substring like </html> would make the whole HTML output crash. So why does the validator still let such a string pass? Simply cause the validator knows that the renderer later on can handle such a problem by replacing "<" by "<" which looks the same but prevents the browser from interpreting the following text as a tag, so the HTML_VALDIATOR does not raise an error. In fact, the validators should let pass as many constellations as possible to provide the user with the most freedom possible. The HTML_VALIDATOR Class is therefore quiet empty cause HTML is not very restrictive, but we build our design in a way to later easily support languages that may be more restrictive, like Markdown. With our design, we can easily add support for another language by creating a new VALIDATOR class and transform the user-input in a form that makes it fit the language.

4.2 Render & Nesting

At the very end, the user always wants to render the document he assembled out of elements to a string, or even a file on his disk. The problem we are facing here is that not all output languages support the same features, some are very open like HTML, some are more restrictive like Markdown. YODA wants to offer the biggest possible set of features to the user, but with ensuring that the output will look fine for all supported output languages. So, the renderer is not just language-specific with the different tags it sets around the content, but the different language renderer also have different set of possibilities, according to the language they render to. For example, rendering a list inside of a table is no problem for the HTML renderer, he just follows the composite pattern and renders the list inside the table, everything is fine. But the same constellation will case heavy problems for markdown, which has no direct representation for a list inside of a table. The markdown renderer cannot simply render the list, it's just not possible for the markdown language. So, we have two options here: We either tell the Markdown validator to not allow lists in tables at all, but this would lead to the loss of this constellation in general, also for HTML output, which is not acceptable. Another solution is needed, and we decided to allow lists in tables for markdown, but make the renderer transform the list in a way that is acceptable for markdown as well. The renderer would therefore identify the problem of having a list inside a table, and react to it by not rendering the list as a markdown list element, but alternatively as a separated text. A list with three entries 1, 2, 3 would then not look like a vertical bullet point list, but like more like the following text: | 1 | 2 | 3 |. This means that the MARKDOWN_Validator can let a list inside of a table pass because he knows that the Markdown renderer would take care of it. We know that not every constellation has such an alternative form in which it can be rendered, but we are optimistic to find a solution for most of such problems. Like, when dealing with a table inside a table, we could just render one table after the other and add an anchor to it. Every constellation that cannot be solved with alternative forms get restricted in the validator. This implies that future output language updates of YODA will only have an impact on very strange creations like multiple nesting of different elements, all other creations will still be supported. Note that the MARKDOWN language is not implemented yet, but the library is already designed in a way to, theoretically, easy support it later on, so the validator and renderer already addresses problems that did not yet occur, but maybe raise in the future with additional language support.

4.3 Process of adding new Language support

YODA focussed its extendibility design on new output languages. The goal was that new output languages can be supported or unsupported with just adding new classes, only a few single lines of code shall be

adapted. This is also a reason why we've designed separate validators for all output languages. So, to be precise, when a new output language is supported, the YODA developer need to analyse the constraints of the new language and find out how certain elements are being displayed. First, a new LANGUAGEX_RENDERER is created which renders each element, either directly or constructs it out of existing ones, like constructing a numbered List out of just a normal list but with manually added numbers to the front. The developers shall pay attention to carefully decide what feature or constellations are impossible to implement. These features and constellations get restricted in the LANGUAGEX_VALIDATOR class, which also has to be added. Finally, when both classes are created, the user needs to change:

1. In the YODA_ELLEMENT class, the new VALIDATOR needs to be added to the languages_array of the feature "validation_languages"
2. The new renderer needs to be added to the "renderer_instances" array in the make feature of the YODA_DOCUMENT class, as well as its string-name to the "renderer_names" at the same location

After these two steps, the language is fully supported!

4.4 Process of adding new Element support

YODA is also built to be extendible on the element side, but adding a new element will not happen frequently and is therefore not that easy as adding a new language support. But it's definitely possible without changes in the existing code, only with adding new lines. First, a new Class "YODA_XELEMENTX" needs to be created, inheriting from YODA_ELEMENT, that defines all features. Next, a Factory in the YODA class is created calling the make procedure of that new element class. Finally, in all RENDERER and VALIDATOR classes, a new function that validates/renders this element has to be created, meaning two functions for each supported language + two deferred function in the deferred classes VALIDATOR and RENDERER.

4.5 Save & Template

As the task description defines, YODA is able to render documents or projects as STRING in the certain language. YODA decided to go one step further and output documents/projects as files. The goal of the save function is to create a folder containing the assembled output files in the correct file format, and having all local resources needed for the project fetched and stored in a local resources folder, so that the user could just take the project/document output folder, copy-paste it to another directory or even FTP-Server and all document features would still work as expected. This is done by, when the user calls save on a project/document, creating a temporary output folder that lies in the current working directory, and every element that is rendered saves its local resources into a resource folder inside of this temp folder. When the rendering process finished, the temporary output folder is renamed to the projectname_output folder and the rendered content is saved inside of this folder as a file, with all its local paths pointing to the previously created resource folder. We used the approach with the temporary folder since we wanted to apply a project/document specific name. However, the image-renderer needs to know the exact name of the folder, and from the renderer we don't have an easy access to the project/document name.

To allow even more flexibility, the user has the ability to inject the document content into a template file. This template file is a text file with a YODA-specific {{content}} tag in it that marks the place where YODA is supposed to inject the rendered document content. Such a template can be used for different purposes. First, it allows the user to define statements outside of YODA to create a fully workable output-file when YODA-saving. In HTML, such a template file may contain header information linking to an online cascade style sheet, or some javascript that is called from the snippet he inserted. There is a variety of possibilities. Second, the template may be an already fully functional file, like an already existing and working website, in which YODA shall inject some content at the right place (which is marked by the {{content}} tag), in order to directly publish the saved documents online without having to copy-pasting strings out of the

console to the right places in the file every time the content in YODA changes. The user may even write a script that fetches YODAs output documents and uploads it to some FTP server, which would imply that the user could change data inside of YODA and when saved, all data is automatically changed on the online webpage directly.

4.7 Anchor

Anchor links allow the user to not just be brought to another document, but to a specific position within a certain document. This way, Anchors allow navigation inside a file by jumping up and down. There are different ways such an anchor can be implemented. One of the easiest ways would be to just give every element on the page a random ID, to which an anchor link can point, but this would mess up the aesthetics of the produced code cause every element had an ID, even though most of them did not even need one. As an alternative, we could just have set the ID's when needed, so only the elements that are really linked would get one. This would be quite easy to implement, but the set-up process of such a design goes somewhat against the implementation of YODA. It would be fairly simple to implement: Every element has a standard ID of -1 on creation. When the user creates an anchor link to some element X, he specifies the ID of X, and the creation procedure of the anchor link takes care to correctly set this new ID and link to it – if the element does not already have a valid ID that is not -1, in this case the user's wished ID would be ignored cause a valid one is already set. But this process would imply that the user would need to have local entities of the YODA element he'd like to link to. Let me illustrate the process of creating an anchor link to a text with this implementation:

```
local
        t: YODA_TEXT
        doc: YODA_DOCUMENT
do
        t := yoda.text("This is the Text that is linked")
        doc.add_element( yoda.link_anchor( yoda.text("Click here!"), t, "some_id") )
        doc.add_element( t )
```

While this is a valid way of creating an anchor, it is neither intuitive nor nice to create. YODA has built factories so that normal elements never have to have local instances, they get directly added to documents. The YODA_TEXT "t" from above beaks this concept here, and the adding process of a link is confusing. So what YODA did is splitting the anchor from elements. Instead of giving elements ID's that are linked, YODA has decided to have a new ANCHOR element that serves as an invisible element which marks a place that is to be linked. So, instead of linking to this YODA_TEXT "t" directly, an anchor element is inserted right above the text, to which an anchor link will link. All visible element can still be added to a document without dealing with local instances, just the special anchor element needs an instance. Let me illustrate how the interaction with anchors may look like

```
local
        anchor: YODA_ANCHOR
        doc: YODA_DOCUMENT
do
        anchor := yoda.anchor("some_id")
        doc.add_element (yoda.link_anchor( yoda.text("Click here"), anchor)
        doc.add_element( anchor )
        doc.add_element( yoda.text("This is the Text that is linked") )
```

First, a local instance of an anchor element is created with the ID this element shall have. Then, at some point in the document, the anchor link to this anchor is inserted, as well as the anchor at the location the

link shall bring the user, so above or under some important content. Later, this anchor can be freely moved independent from the page content, while everything remains working as expected.

4.8 as_string

YODA includes a simple method to get an overview over the all the elements in a document or project, the print_to_console features. This feature simply loops over the elements and prints its abstract names, like “table” for a table, to the console, without showing its content. This is neat to ensure that an assembled document follows the right structure, without having to scroll through the outputted HTML content. We made this little feature quite restrictive to not display heavy nesting but only the first two layers, mainly cause it would have been to difficult to represent a table inside of a table etc, and if we’d have addresses such problems, we could also have implemented markdown as well.

4.9 Image and folder system

For including Images into the documents there are two ways of specifying the resource Image file. On the one hand the client can just add a web URL where the image then gets fetched from. This is not a great problem in HTML. But the rely on a web image can bear great drawback, imagine the image get changed at one time and the client does not notice it. Therefor we wanted to provide the client with the option to give a local image as source. Since, YODA’s first version only supports HTML we focused on how HTML includes local images. One big burden that HTML has, considering local images, is that the path to the image must be relative and not absolute. Since we expect our client to not leave the saved files in the same place where we write it to, we had to find a way to lower the chances that the relative path gets broken. So, we decided to create a project folder where the documents get saved to and where we then also create a resource folder a copy the image file to that resource folder. From that we expect that the relative path to be rather stable since it does not reach far. Also, we expect the client to when moving the project, moving the whole folder, so the document and the resources always stay in the same relative relation to each other.

Creating these project folders seems to be a great solution, however this lead to more problems. Since we do also give the option to only save documents individually, we had to make sure that then we create a document specific folder. Therefore, we had to create another save function for the document, so that we have one for being call by the project and one for the “direct” use of the client.

The whole argument above only considers the case of the save process, but what should happen during the rendering where we have to set the correct tag? We agreed that in this case we create a temp folder where the image gets coped to and then write a relative path from the working directory into the resource folder in that temporary folder. Since for just rendering the images won’t be displayed we do not really need to copy the images. This could be a point to overthink again. However, we considered it so be kind of useful for the user to see what kinds of resources he accesses. Also, since when he copy-pastes the string from the rendering he has to manually change the paths. This for sure is not very neat but this is a consequence of working with a relative path.

Another solution would have been to just expect the user to enter the correct relative path. Problem here would be that we could not check if the image really exist since the root of the relative path might not be the working directory, and also the client would ned to know in advance how the file structure will look in the end.

4.10 “Four in One” link

It might seem confusing that we four different make functions for the link. However, after further inspection of the functionality of the link it should become clear, that the link element is a very versatile object. in the case of YODA links can point to an external website, to another YODA-Document, to an email (“mailto”) or to an anchor. Since all of these links follow the same structure, namely an argument

for what should be displayed and where the link should point to (exception here is the email where both is the same, as we display the string of where we point to), we decided to not split the link-element into four different elements, but rather use four different make functions. Not only the similar structure speaks for keeping all four variations in one element but also that the rendering works the same for all four. Therefore, we saved up on the work of implementing the “same” renderer four times. However, since we use four different make-functions we also need four different validators, but with our architecture that’s no problem since we can call different validators dependent on the make function.