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

October 29, 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 chases 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"

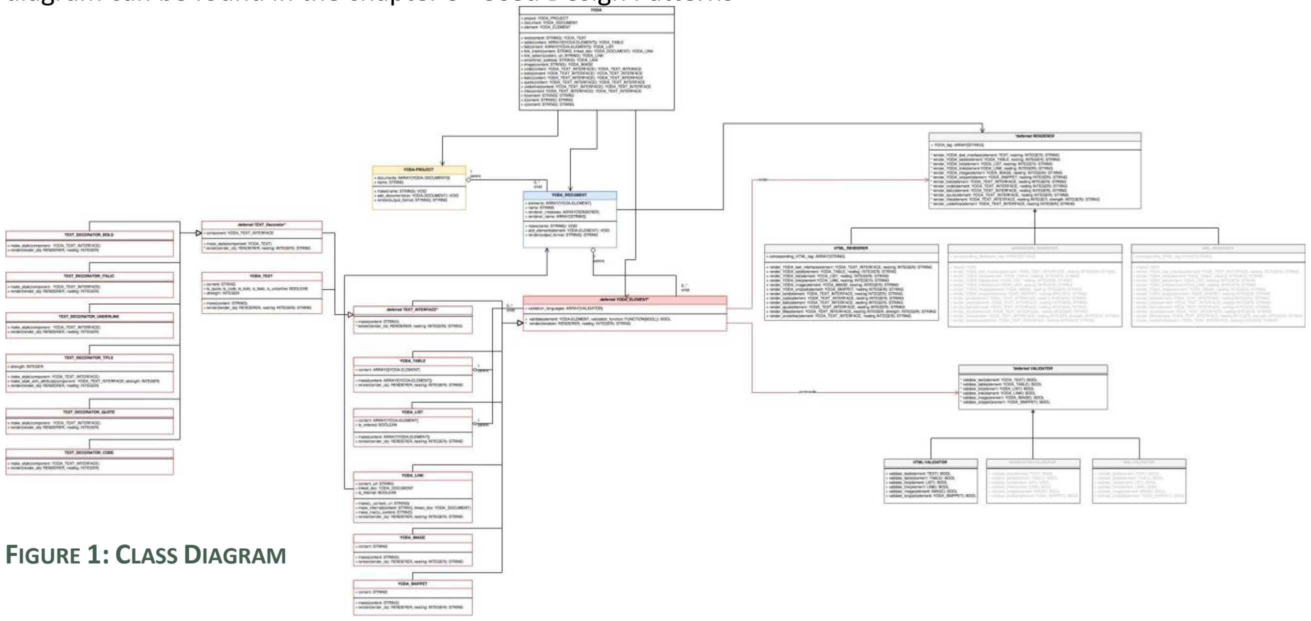


FIGURE 1: CLASS DIAGRAM

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 uniformly treat single objects and compositions.

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 things. This so-called nesting of the elements YODA solves 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.

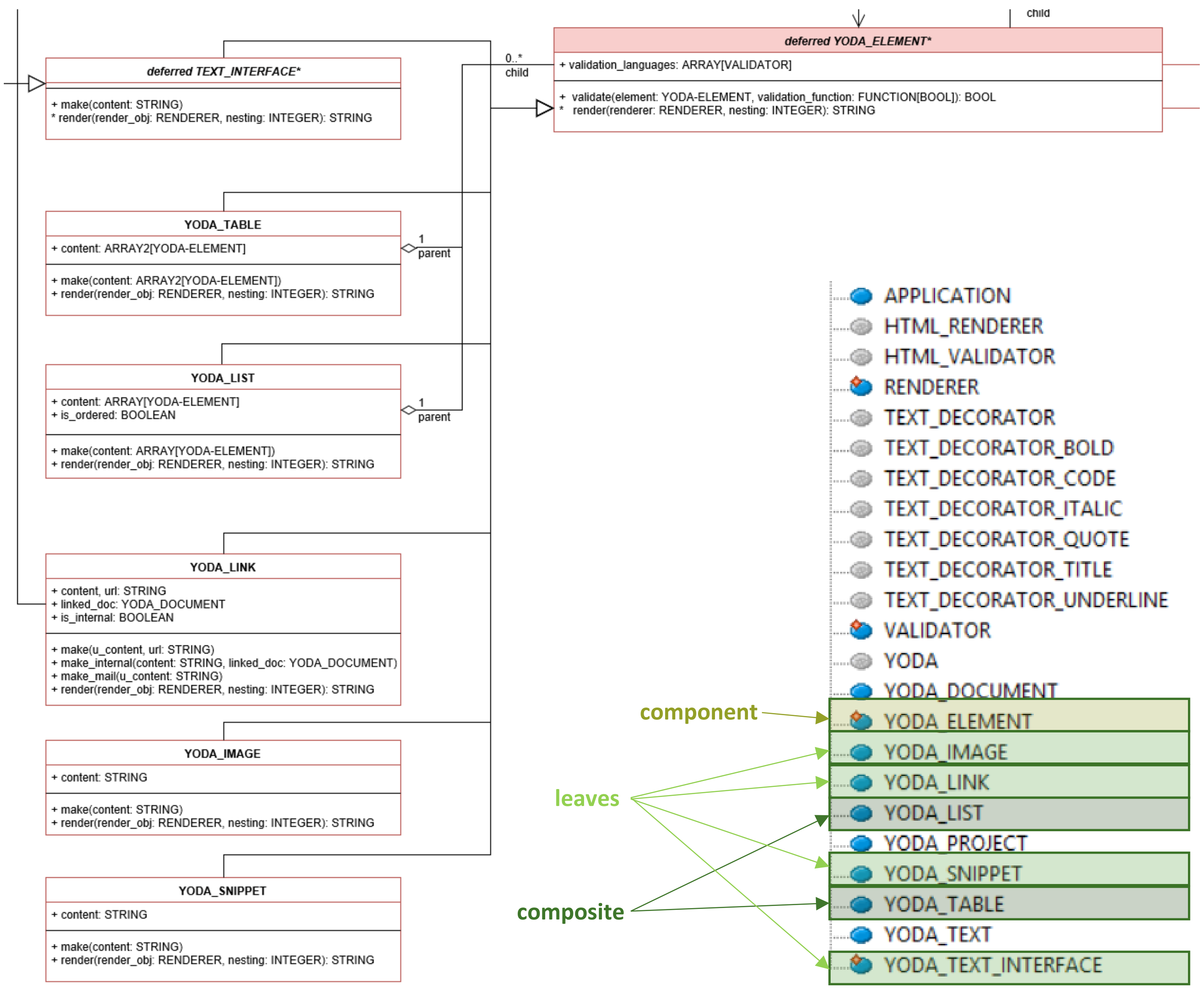

child
deferred YODA_ELEMENT*
+ validation_languages: ARRAY[VALIDATOR]
+ validate(element: YODA-ELEMENT, validation_function: FUNCTION[BOOL]): BOOL
* render(renderer: RENDERER, nesting: INTEGER): STRING
0..*
child
deferred TEXT_INTERFACE*
+ make(content: STRING)
* render(render_obj: RENDERER, nesting: INTEGER): STRING
YODA_TABLE
+ content: ARRAY2[YODA-ELEMENT]
+ make(content: ARRAY2[YODA-ELEMENT])
+ render(render_obj: RENDERER, nesting: INTEGER): STRING
1
parent
YODA_LIST
+ content: ARRAY[YODA-ELEMENT]
+ is_ordered: BOOLEAN
+ make(content: ARRAY[YODA-ELEMENT])
+ render(render_obj: RENDERER, nesting: INTEGER): STRING
1
parent
YODA_LINK
+ content, url: STRING
+ linked_doc: YODA_DOCUMENT
+ is_internal: BOOLEAN
+ make(u_content, url: STRING)
+ make_internal(content: STRING, linked_doc: YODA_DOCUMENT)
+ make_mail(u_content: STRING)
+ render(render_obj: RENDERER, nesting: INTEGER): STRING
YODA_IMAGE
+ content: STRING
+ make(content: STRING)
+ render(render_obj: RENDERER, nesting: INTEGER): STRING
YODA_SNIPPET
+ content: STRING
+ make(content: STRING)
+ render(render_obj: RENDERER, nesting: INTEGER): STRING
APPLICATION
HTML_RENDERER
HTML_VALIDATOR
RENDERER
TEXT_DECORATOR
TEXT_DECORATOR_BOLD
TEXT_DECORATOR_CODE
TEXT_DECORATOR_ITALIC
TEXT_DECORATOR_QUOTE
TEXT_DECORATOR_TITLE
TEXT_DECORATOR_UNDERLINE
VALIDATOR
YODA
YODA_DOCUMENT
component
YODA_ELEMENT
YODA_IMAGE
leaves
YODA_LINK
YODA_LIST
YODA_PROJECT
YODA_SNIPPET
composite
YODA_TABLE
YODA_TEXT
YODA_TEXT_INTERFACE

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.

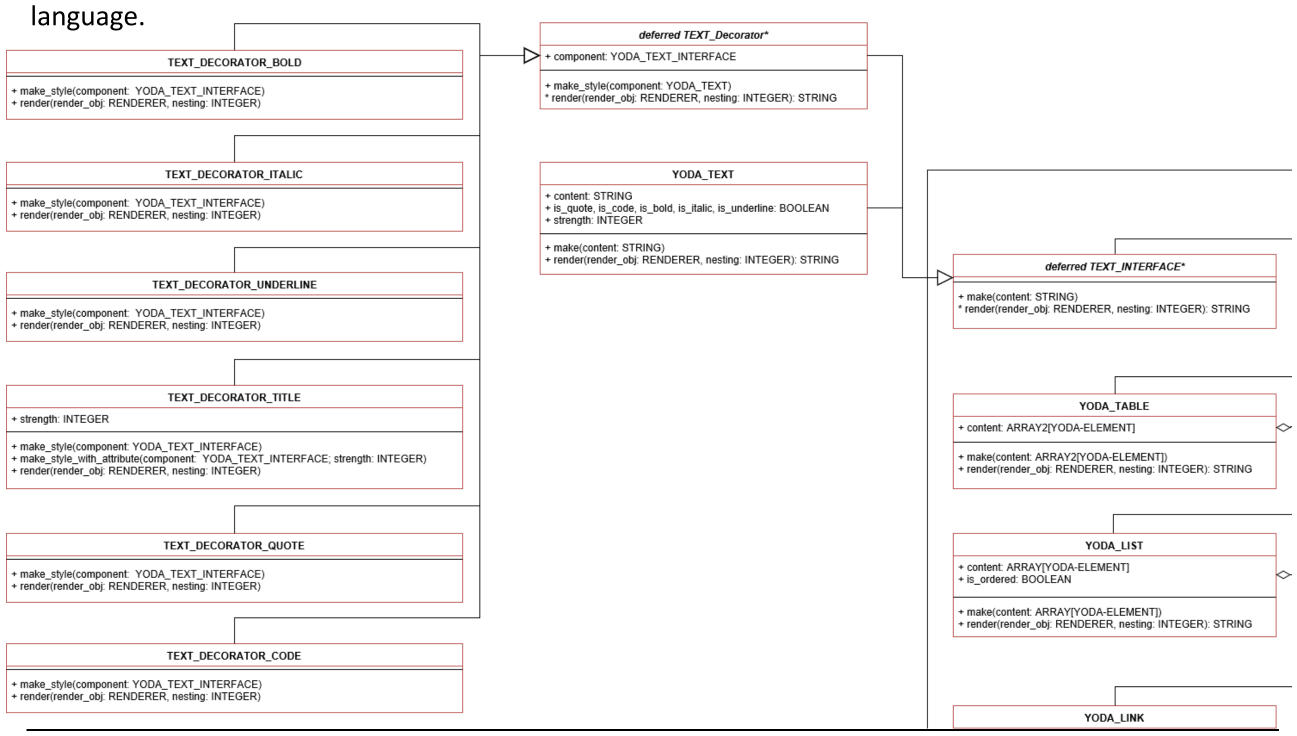
3.2.2 What YODA uses Decorators for

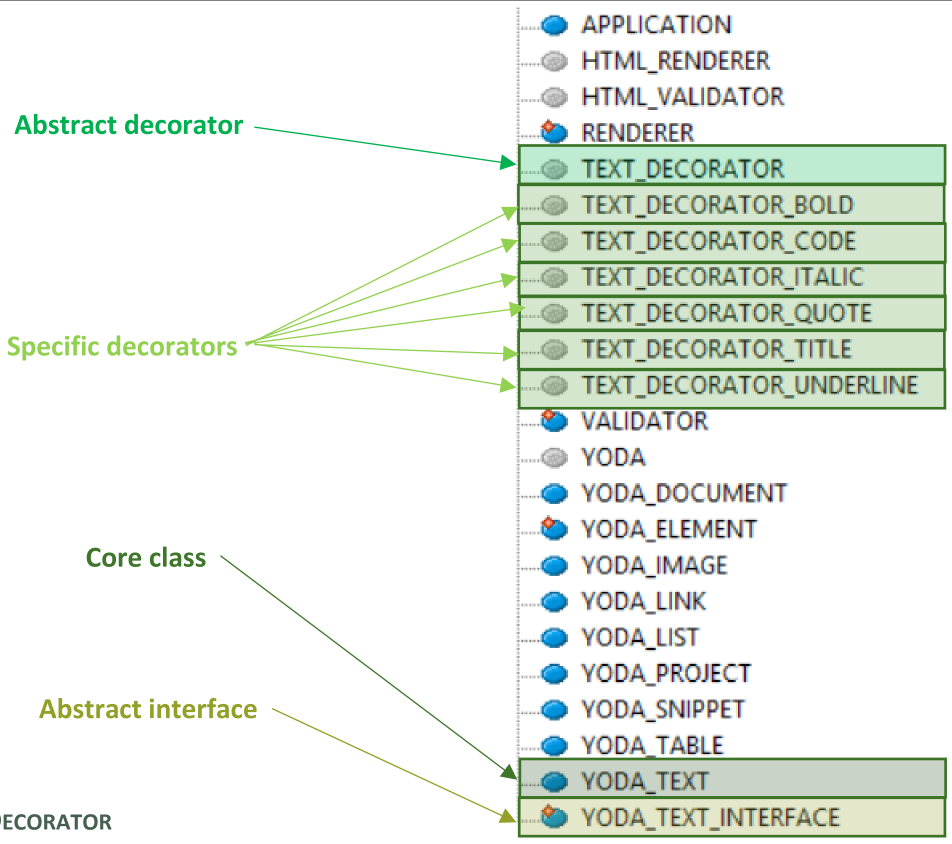
YODA uses Decorators for the styling of the text-element. HTML and Markup languages support some basic text styling such as making the text bold, italic or underlined. For these stylings YODA uses Decorators. In Addition to this basic styling YODA uses the decorators for tiles, quotes and displayed code, 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.

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.


TEXT_DECORATOR_BOLD
+ make_style(component: YODA_TEXT_INTERFACE)
+ render(render_obj: RENDERER, nesting: INTEGER)
TEXT_DECORATOR_ITALIC
+ make_style(component: YODA_TEXT_INTERFACE)
+ render(render_obj: RENDERER, nesting: INTEGER)
TEXT_DECORATOR_UNDERLINE
+ make_style(component: YODA_TEXT_INTERFACE)
+ render(render_obj: RENDERER, nesting: INTEGER)
TEXT_DECORATOR_TITLE
+ strength: INTEGER
+ make_style(component: YODA_TEXT_INTERFACE)
+ make_style_with_attribute(component: YODA_TEXT_INTERFACE; strength: INTEGER)
+ render(render_obj: RENDERER, nesting: INTEGER)
TEXT_DECORATOR_QUOTE
+ make_style(component: YODA_TEXT_INTERFACE)
+ render(render_obj: RENDERER, nesting: INTEGER)
TEXT_DECORATOR_CODE
+ make_style(component: YODA_TEXT_INTERFACE)
+ render(render_obj: RENDERER, nesting: INTEGER)
deferred TEXT_Decorator*
+ component: YODA_TEXT_INTERFACE
+ make_style(component: YODA_TEXT)
* render(render_obj: RENDERER, nesting: INTEGER): STRING
YODA_TEXT
+ content: STRING
+ is_quote, is_code, is_bold, is_italic, is_underline: BOOLEAN
+ strength: INTEGER
+ make(content: STRING)
+ render(render_obj: RENDERER, nesting: INTEGER): STRING
deferred TEXT_INTERFACE*
+ make(content: STRING)
* render(render_obj: RENDERER, nesting: INTEGER): STRING
YODA_TABLE
+ content: ARRAY2[YODA-ELEMENT]
+ make(content: ARRAY2[YODA-ELEMENT])
+ render(render_obj: RENDERER, nesting: INTEGER): STRING
YODA_LIST
+ content: ARRAY[YODA-ELEMENT]
+ is_ordered: BOOLEAN
+ make(content: ARRAY[YODA-ELEMENT])
+ render(render_obj: RENDERER, nesting: INTEGER): STRING
YODA_LINK



APPLICATION
HTML_RENDERER
HTML_VALIDATOR
RENDERER
TEXT_DECORATOR
TEXT_DECORATOR_BOLD
TEXT_DECORATOR_CODE
TEXT_DECORATOR_ITALIC
TEXT_DECORATOR_QUOTE
TEXT_DECORATOR_TITLE
TEXT_DECORATOR_UNDERLINE
VALIDATOR
YODA
YODA_DOCUMENT
YODA_ELEMENT
YODA_IMAGE
YODA_LINK
YODA_LIST
YODA_PROJECT
YODA_SNIPPET
YODA_TABLE
YODA_TEXT
YODA_TEXT_INTERFACE

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 decided for the decorator however since the decorator pattern 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

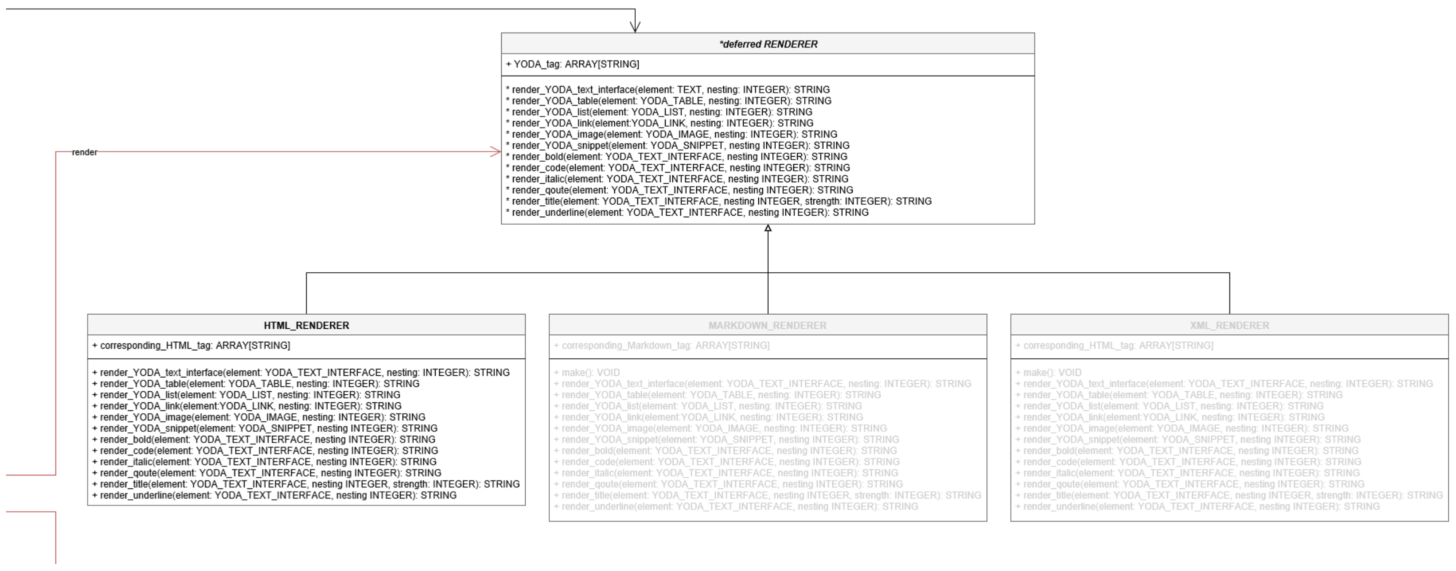
The visitor design pattern describes an architectural scheme that separates an operation from an object structure on which it operates to solve the double dispatch problem. This is possible by the elegant collaboration of the three main participants, the target classes, the client classes and the visitor 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:


*deferred RENDERER
+ YODA_tag: ARRAY[STRING]
* render_YODA_text_interface(element: TEXT, nesting: INTEGER): STRING
* render_YODA_table(element: YODA_TABLE, nesting: INTEGER): STRING
* render_YODA_list(element: YODA_LIST, nesting: INTEGER): STRING
* render_YODA_link(element:YODA_LINK, nesting: INTEGER): STRING
* render_YODA_image(element: YODA_IMAGE, nesting: INTEGER): STRING
* render_YODA_snippet(element: YODA_SNIPPET, nesting INTEGER): STRING
* render_bold(element: YODA_TEXT_INTERFACE, nesting INTEGER): STRING
* render_code(element: YODA_TEXT_INTERFACE, nesting INTEGER): STRING
* render_italic(element: YODA_TEXT_INTERFACE, nesting INTEGER): STRING
* render_qoute(element: YODA_TEXT_INTERFACE, nesting INTEGER): STRING
* render_title(element: YODA_TEXT_INTERFACE, nesting INTEGER, strength: INTEGER): STRING
* render_underline(element: YODA_TEXT_INTERFACE, nesting INTEGER): STRING
render
HTML_RENDERER
+ corresponding_HTML_tag: ARRAY[STRING]
+ render_YODA_text_interface(element: YODA_TEXT_INTERFACE, nesting: INTEGER): STRING
+ render_YODA_table(element: YODA_TABLE, nesting: INTEGER): STRING
+ render_YODA_list(element: YODA_LIST, nesting: INTEGER): STRING
+ render_YODA_link(element:YODA_LINK, nesting: INTEGER): STRING
+ render_YODA_image(element: YODA_IMAGE, nesting: INTEGER): STRING
+ render_YODA_snippet(element: YODA_SNIPPET, nesting INTEGER): STRING
+ render_bold(element: YODA_TEXT_INTERFACE, nesting INTEGER): STRING
+ render_code(element: YODA_TEXT_INTERFACE, nesting INTEGER): STRING
+ render_italic(element: YODA_TEXT_INTERFACE, nesting INTEGER): STRING
+ render_qoute(element: YODA_TEXT_INTERFACE, nesting INTEGER): STRING
+ render_title(element: YODA_TEXT_INTERFACE, nesting INTEGER, strength: INTEGER): STRING
+ render_underline(element: YODA_TEXT_INTERFACE, nesting INTEGER): STRING
MARKDOWN_RENDERER
+ corresponding_Markdown_tag: ARRAY[STRING]
+ make(): VOID
+ render_YODA_text_interface(element: YODA_TEXT_INTERFACE, nesting: INTEGER): STRING
+ render_YODA_table(element: YODA_TABLE, nesting: INTEGER): STRING
+ render_YODA_list(element: YODA_LIST, nesting: INTEGER): STRING
+ render_YODA_link(element:YODA_LINK, nesting: INTEGER): STRING
+ render_YODA_image(element: YODA_IMAGE, nesting: INTEGER): STRING
+ render_YODA_snippet(element: YODA_SNIPPET, nesting INTEGER): STRING
+ render_bold(element: YODA_TEXT_INTERFACE, nesting INTEGER): STRING
+ render_code(element: YODA_TEXT_INTERFACE, nesting INTEGER): STRING
+ render_italic(element: YODA_TEXT_INTERFACE, nesting INTEGER): STRING
+ render_qoute(element: YODA_TEXT_INTERFACE, nesting INTEGER): STRING
+ render_title(element: YODA_TEXT_INTERFACE, nesting INTEGER, strength: INTEGER): STRING
+ render_underline(element: YODA_TEXT_INTERFACE, nesting INTEGER): STRING
XML_RENDERER
+ corresponding_HTML_tag: ARRAY[STRING]
+ make(): VOID
+ render_YODA_text_interface(element: YODA_TEXT_INTERFACE, nesting: INTEGER): STRING
+ render_YODA_table(element: YODA_TABLE, nesting: INTEGER): STRING
+ render_YODA_list(element: YODA_LIST, nesting: INTEGER): STRING
+ render_YODA_link(element:YODA_LINK, nesting: INTEGER): STRING
+ render_YODA_image(element: YODA_IMAGE, nesting: INTEGER): STRING
+ render_YODA_snippet(element: YODA_SNIPPET, nesting INTEGER): STRING
+ render_bold(element: YODA_TEXT_INTERFACE, nesting INTEGER): STRING
+ render_code(element: YODA_TEXT_INTERFACE, nesting INTEGER): STRING
+ render_italic(element: YODA_TEXT_INTERFACE, nesting INTEGER): STRING
+ render_qoute(element: YODA_TEXT_INTERFACE, nesting INTEGER): STRING
+ render_title(element: YODA_TEXT_INTERFACE, nesting INTEGER, strength: INTEGER): STRING
+ render_underline(element: YODA_TEXT_INTERFACE, nesting INTEGER): STRING

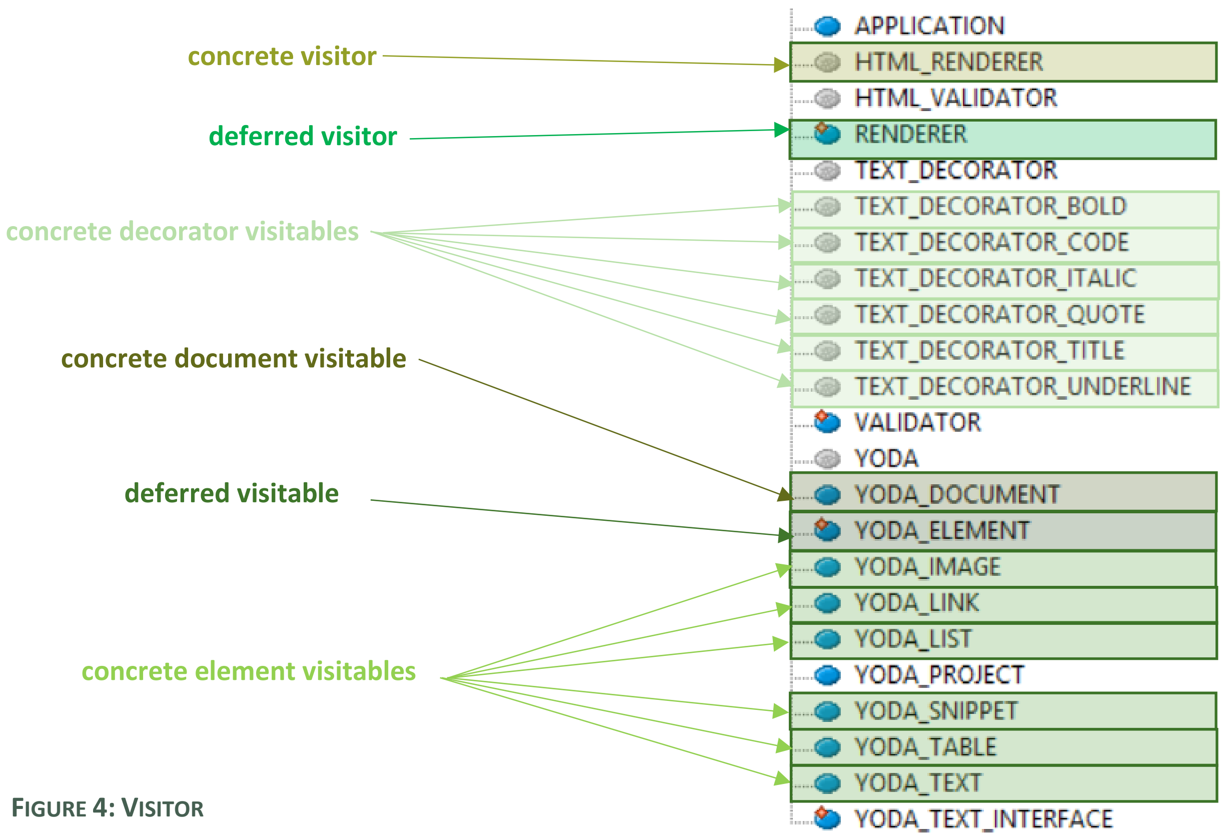
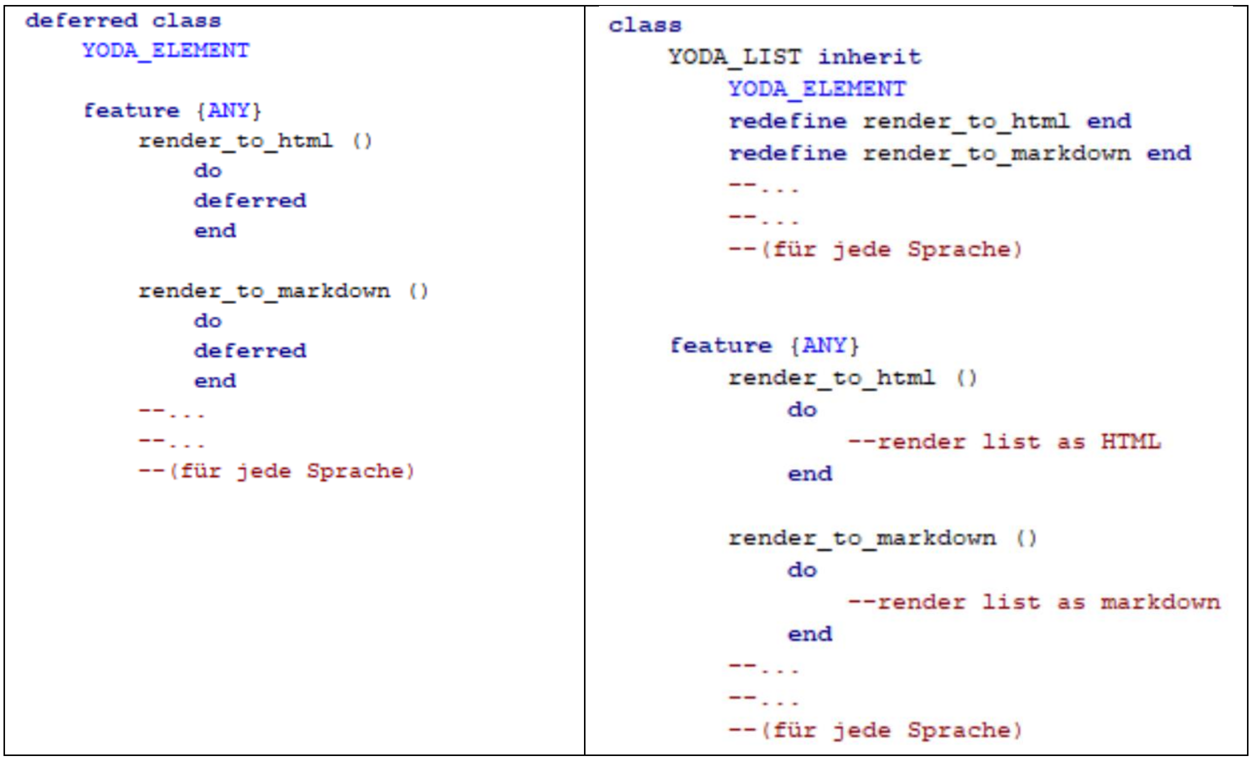


FIGURE 4: VISITOR

3.2.4 Alternative approaches and why YODA does not use them

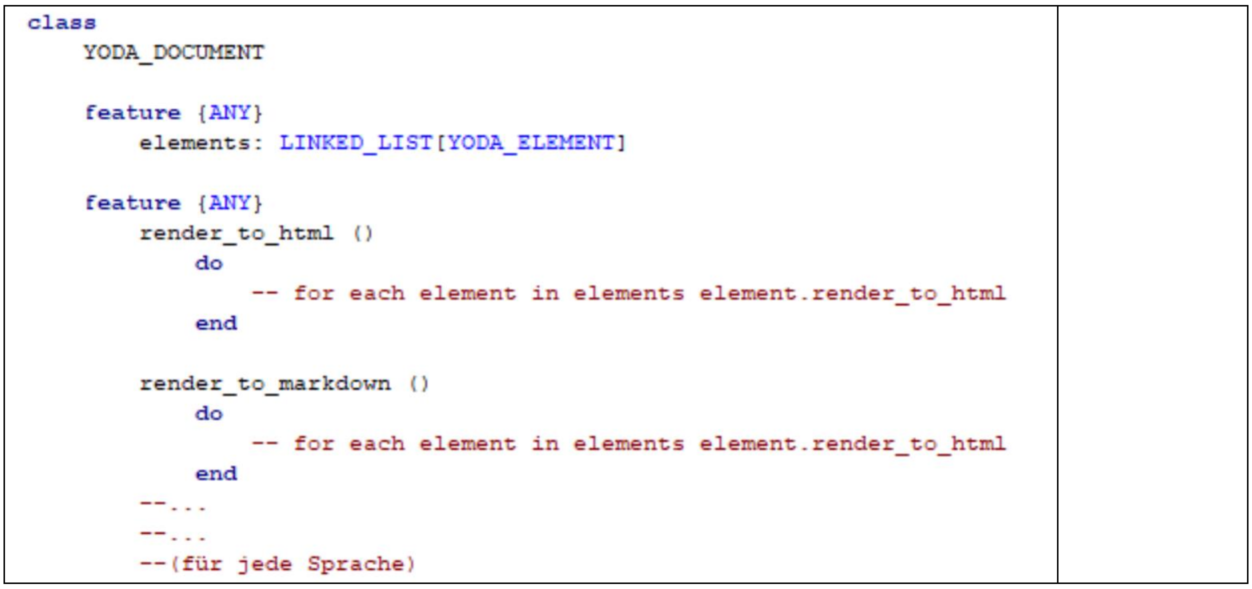
The most straightforward, but very naive, implementation to render the different YODA-Elements in the different supported languages would be a deferred procedure called render_to_X() for each language in the YODA-ELEMENT class and redefined procedures in all of its children, the concrete elements, as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

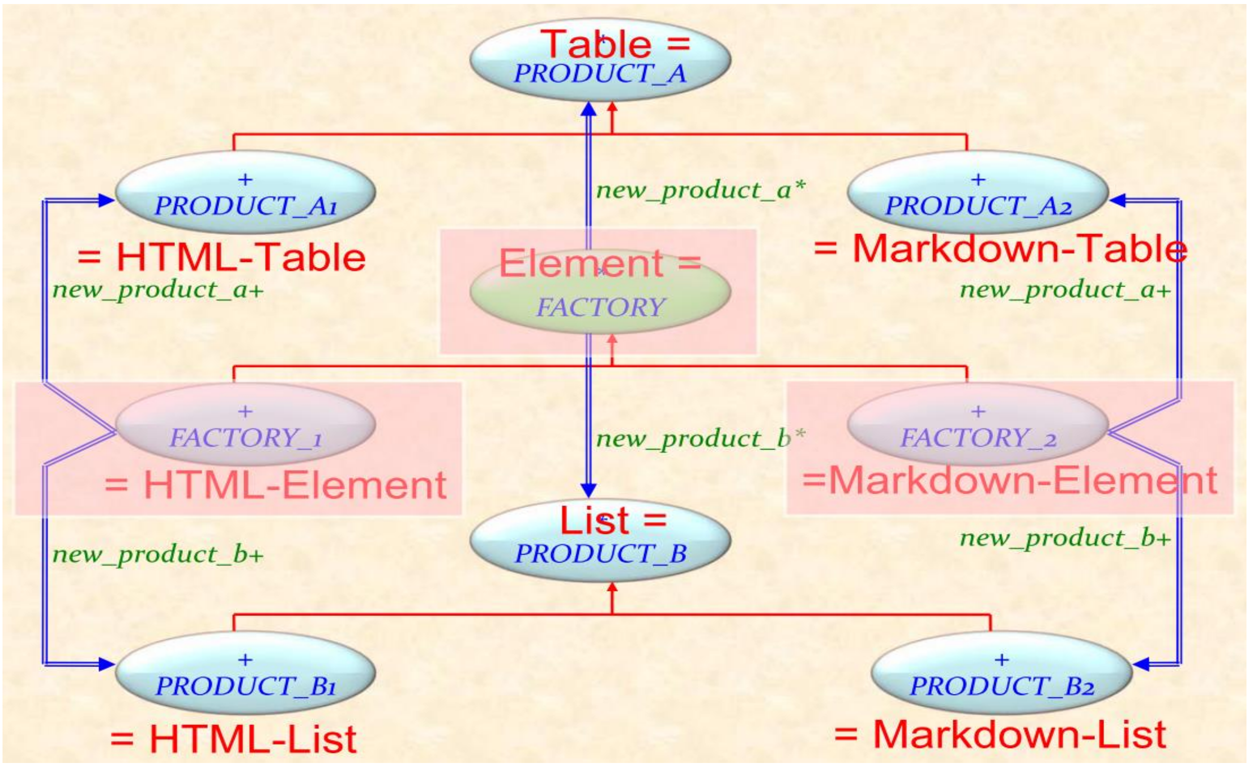


FIGURE 7: ALTERNATIVE DESIGN: ABSTRACT FACTORY

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.

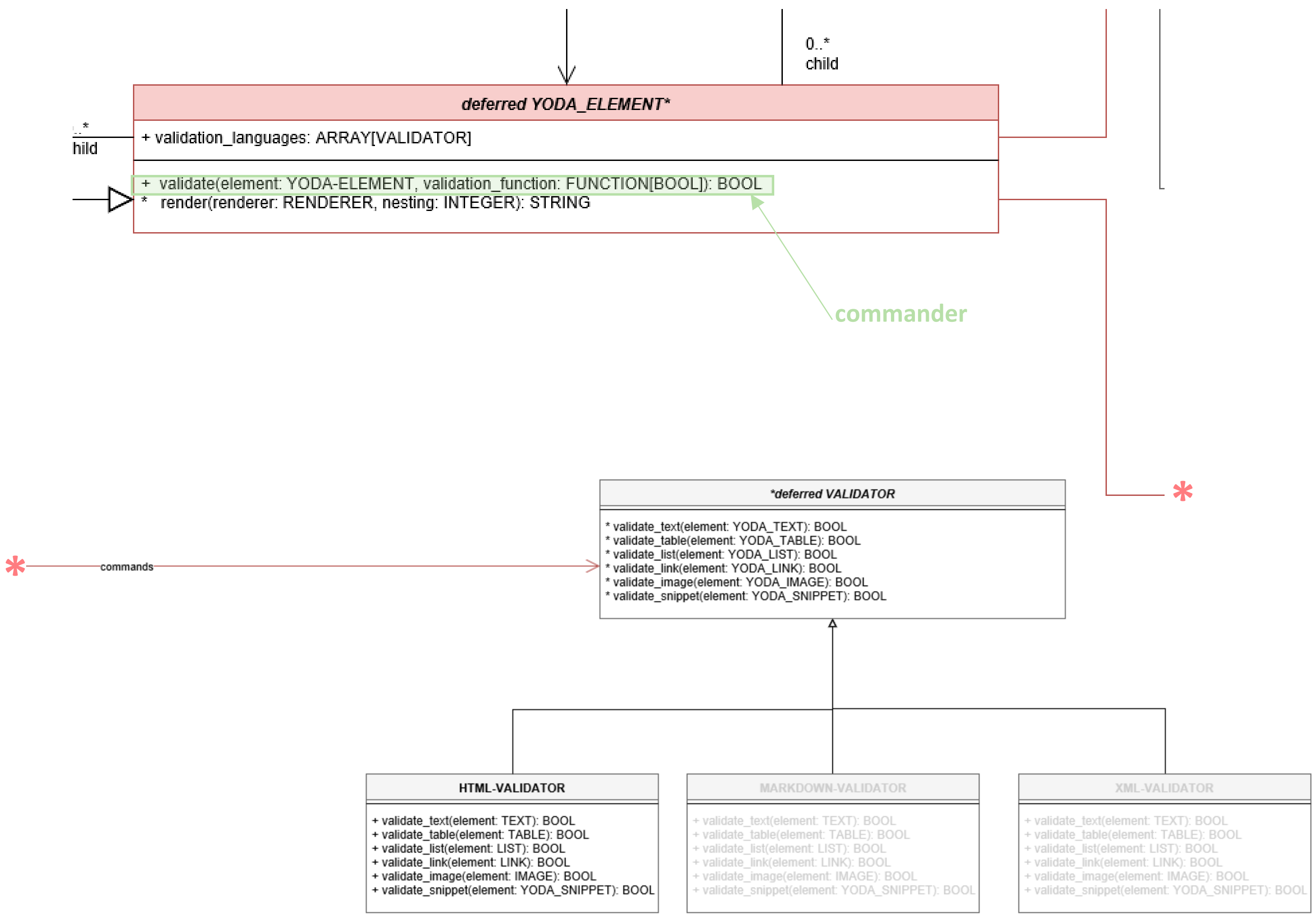
3.4.3 How YODA uses Commands

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_TABLE may be: The handed in two-dimensional array has no YODA_ELEMENT entry of type YODA_TABLE. In words, YODA does not support tables inside of tables, which is a constraint we defined by ourselves. 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 validator_commander do validate the object: It passes the current object and the corresponding validation-function to the commander. The commander then iterates through all the supported language validators (HTML_VALIDATOR, MARKDOWN_VALIDATOR, XML_VALIDATOR), 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 validate(current, validate_table)

The commander validate 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 validate received True or False and, accordingly, lets the creation of a Table pass or gives an Error.

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 validator_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 validation function.


0..*
child
..*
hild
deferred YODA_ELEMENT*
+ validation_languages: ARRAY[VALIDATOR]
+ validate(element: YODA-ELEMENT, validation_function: FUNCTION[BOOL]): BOOL
* render(renderer: RENDERER, nesting: INTEGER): STRING
commander
*
*deferred VALIDATOR
* validate_text(element: YODA_TEXT): BOOL
* validate_table(element: YODA_TABLE): BOOL
* validate_list(element: YODA_LIST): BOOL
* validate_link(element: YODA_LINK): BOOL
* validate_image(element: YODA_IMAGE): BOOL
* validate_snippet(element: YODA_SNIPPET): BOOL
*
commands
HTML-VALIDATOR
+ validate_text(element: TEXT): BOOL
+ validate_table(element: TABLE): BOOL
+ validate_list(element: LIST): BOOL
+ validate_link(element: LINK): BOOL
+ validate_image(element: IMAGE): BOOL
+ validate_snippet(element: YODA_SNIPPET): BOOL
MARKDOWN-VALIDATOR
+ validate_text(element: TEXT): BOOL
+ validate_table(element: TABLE): BOOL
+ validate_list(element: LIST): BOOL
+ validate_link(element: LINK): BOOL
+ validate_image(element: IMAGE): BOOL
+ validate_snippet(element: YODA_SNIPPET): BOOL
XML-VALIDATOR
+ validate_text(element: TEXT): BOOL
+ validate_table(element: TABLE): BOOL
+ validate_list(element: LIST): BOOL
+ validate_link(element: LINK): BOOL
+ validate_image(element: IMAGE): BOOL
+ validate_snippet(element: YODA_SNIPPET): BOOL

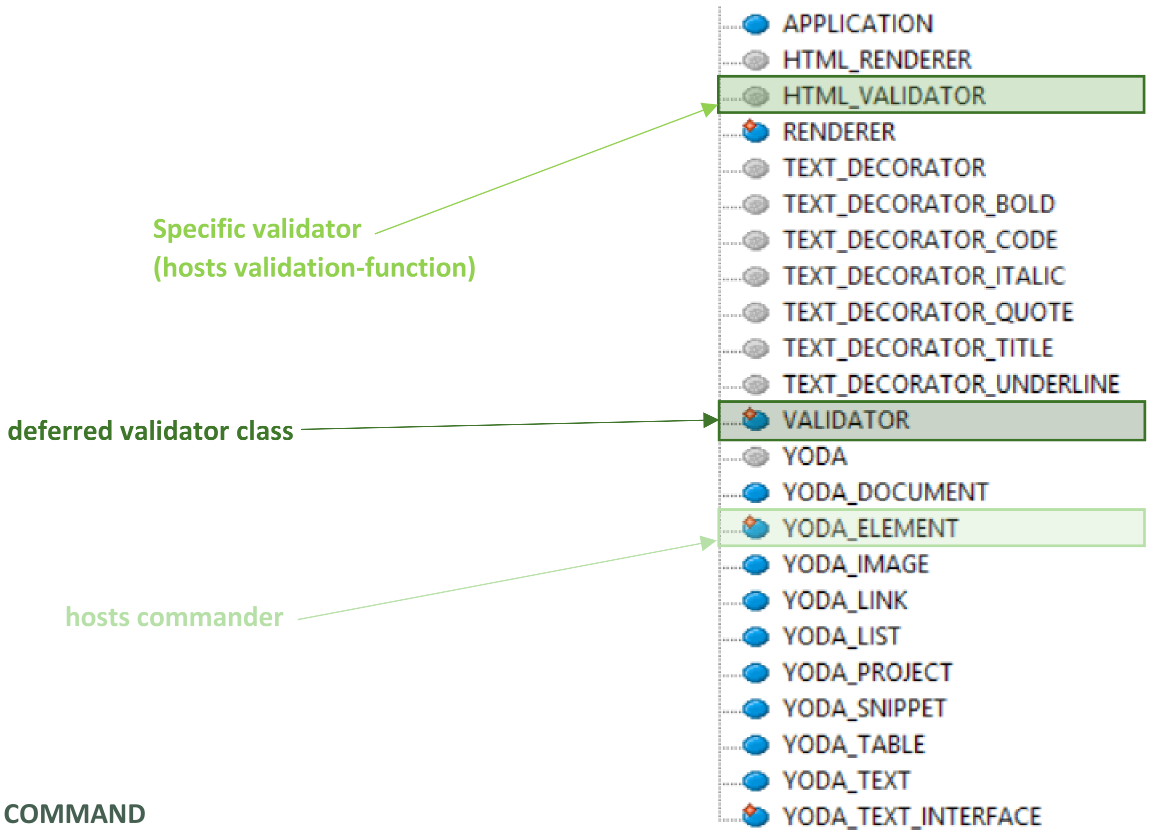


FIGURE 8: COMMAND

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 exactly the same thing as the validator_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 differes the validation process as an argument: the validation function.

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!"))

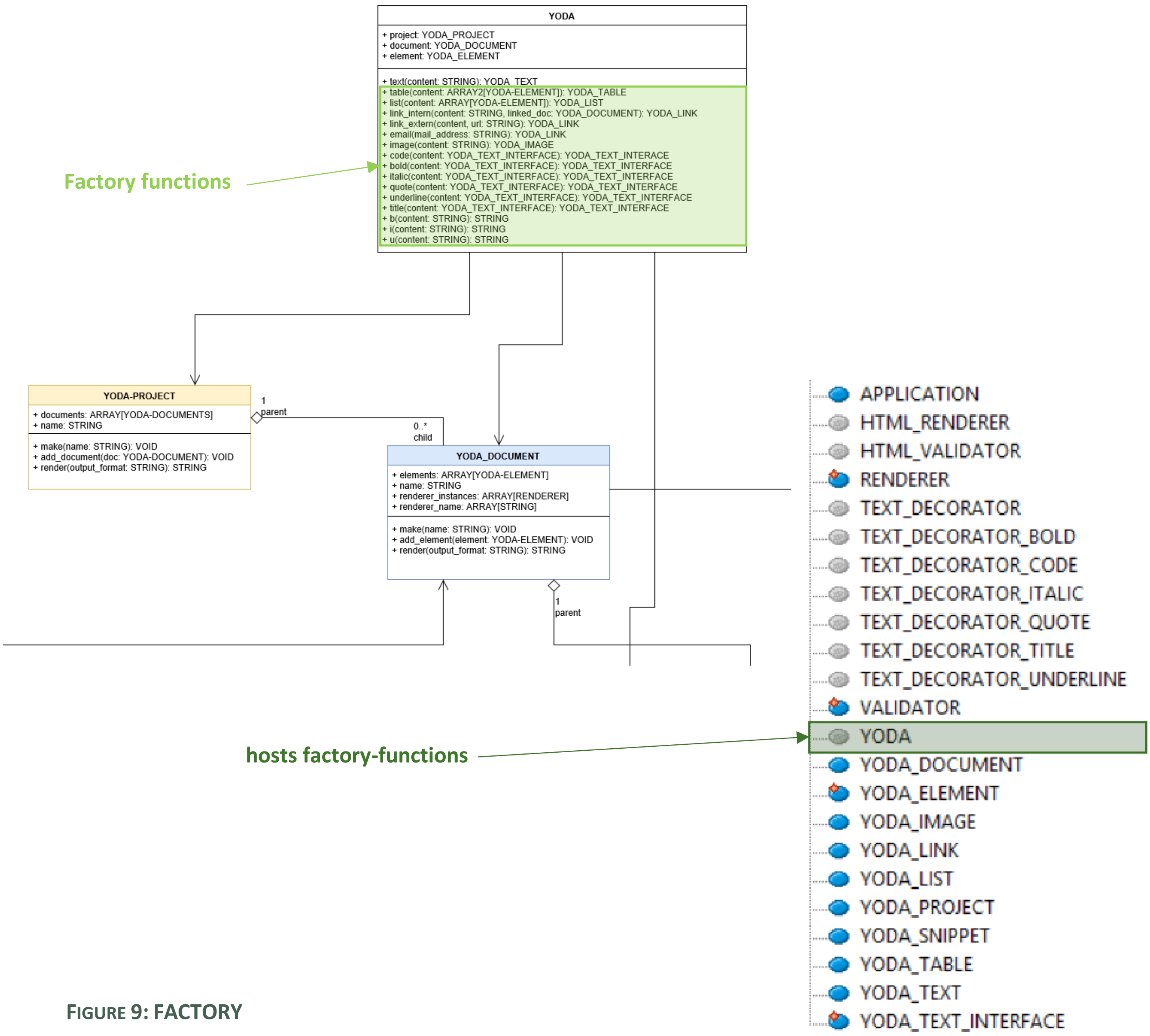

YODA
+ project: YODA_PROJECT
+ document: YODA_DOCUMENT
+ element: YODA_ELEMENT
+ text(content: STRING): YODA_TEXT
+ table(content: ARRAY2[YODA-ELEMENT]): YODA_TABLE
+ list(content: ARRAY[YODA-ELEMENT]): YODA_LIST
+ link_intern(content: STRING, linked_doc: YODA_DOCUMENT): YODA_LINK
+ link_extern(content, url: STRING): YODA_LINK
+ email(mail_address: STRING): YODA_LINK
+ image(content: STRING): YODA_IMAGE
+ code(content: YODA_TEXT_INTERFACE): YODA_TEXT_INTERACE
+ bold(content: YODA_TEXT_INTERFACE): YODA_TEXT_INTERFACE
+ italic(content: YODA_TEXT_INTERFACE): YODA_TEXT_INTERFACE
+ quote(content: YODA_TEXT_INTERFACE): YODA_TEXT_INTERFACE
+ underline(content: YODA_TEXT_INTERFACE): YODA_TEXT_INTERFACE
+ title(content: YODA_TEXT_INTERFACE): YODA_TEXT_INTERFACE
+ b(content: STRING): STRING
+ i(content: STRING): STRING
+ u(content: STRING): STRING
Factory functions
YODA-PROJECT
+ documents: ARRAY[YODA-DOCUMENTS]
+ name: STRING
+ make(name: STRING): VOID
+ add_document(doc: YODA-DOCUMENT): VOID
+ render(output_format: STRING): STRING
1
parent
0..*
child
YODA_DOCUMENT
+ elements: ARRAY[YODA-ELEMENT]
+ name: STRING
+ renderer_instances: ARRAY[RENDERER]
+ renderer_name: ARRAY[STRING]
+ make(name: STRING): VOID
+ add_element(element: YODA-ELEMENT): VOID
+ render(output_format: STRING): STRING
1
parent
hosts factory-functions
APPLICATION
HTML_RENDERER
HTML_VALIDATOR
RENDERER
TEXT_DECORATOR
TEXT_DECORATOR_BOLD
TEXT_DECORATOR_CODE
TEXT_DECORATOR_ITALIC
TEXT_DECORATOR_QUOTE
TEXT_DECORATOR_TITLE
TEXT_DECORATOR_UNDERLINE
VALIDATOR
YODA
YODA_DOCUMENT
YODA_ELEMENT
YODA_IMAGE
YODA_LINK
YODA_LIST
YODA_PROJECT
YODA_SNIPPET
YODA_TABLE
YODA_TEXT
YODA_TEXT_INTERFACE

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.