Microdown: a clean and extensible markup language to support Pharo documentation

Stéphane Ducasse, Laurine Dargaud, Guillermo Polito

Microdown: a clean and extensible markup language to support Pharo documentation

S. Ducasse
Inria, Univ. Lille, CNRS, Centrale Lille
Lille, France
stephane.ducasse@inria.fr

L. Dargaud
Inria, Univ. Lille, CNRS, Centrale Lille
Lille, France
dargaud.laurine@gmail.com

G. Polito
Univ. Lille, CNRS, Centrale Lille, Inria,
UMR 9189 - CRIStAL - Centre de
Recherche en Informatique Signal et
Automatique de Lille
Lille, France
guillermo.polito@univ-lille.fr

ABSTRACT

Markdown is imposing itself as a defacto standard for wiki-like syntax. However, building a Markdown implementation that correctly interacts with other implementations requires a careful design. First, Markdown clumsily proposes multiple syntaxes for the same elements. Second, there is a Markdown specification but many different sub-specifications and implementations, each of them with different super/sub-sets, where github Markdown is one of the most knowns. The Markdown standard does not support many features that are important for book writing such as anchor definition, figure and code block parameters. Finally, it does not offer coherent and systematic extension mechanisms.

In this article we present Microdown. Microdown is a clean subset of Markdown introducing clean extension mechanisms as proposed in Pillar. The objective of Microdown is to have a syntax compatible with most of the existing markdown implementations, extensible, usable for book writings, and non-ambiguous. Microdown is the foundation for class and package comments in Pharo image as well as full book and documentation.

ACM Reference Format:
S. Ducasse, L. Dargaud, and G. Polito. 2020. Microdown: a clean and extensible markup language to support Pharo documentation . In *International Workshop on Smalltalk Technologies'20.* ACM, New York, NY, USA, 8 pages. https://doi.org/xxxxxxx

1 INTRODUCTION

There are many different markup languages to describe documents such as ASCIIDoc, Pandoc, Markdown, and the less known Pillar [ADCD16] (inspired from the original Wiki via the SmallWiki implementation [DRW05] and in use since 2002). Among all these options, Markdown is imposing itself as a defacto standard for wiki-like syntax. Markdown is a lightweight markup language declined in many implementations and supported by many tools, for example with Github Markdown and its related online editing and render tools. The fact that Github uses markdown for its online documentation is massively promoting Markdown.

However, Markdown is clumsy. It proposes multiple syntaxes for the same elements. In addition, it does not support many important features mandatory for books or structured documents such as explicit anchor definition, references to figures and code elements. Finally, it does not offer coherent and systematic extension mechanisms. This leads to Markdown dialects such as MarkDeep[1] that is a kitchen sink of features.

In this article, we present Microdown: a markup language based on Markdown. Microdown takes the good pieces of Markdown so that developers write class comments with a familiar syntax, and this familiar syntax renders well in existing tools such as Github online renderers. Also, Microdown proposes extensions and a flexible extension mechanisms to produce books, slides and websites such as compilation chains like Pillar (http://github.com/pillar-markup/).

Contrary to Markdown extensions such as MarkDeep, the design of Microdown is to have a small core in the spirit of Smalltalk design, but with the extra constraint of supporting the main elements of Markdown. Such minimality but also the familiarity with Markdown lower the user cognitive load but also of the maintenance of the project.

Finally Microdown fulfils three important goals for Pharo: (1) first it is the foundation to support the writing of better class and package comments. Pharo uses it as a structured text and renders such elements using rich text rendering, (2) all the Pharo external documentations such as books, booklets will be migrated to Microdown to support on the fly loading and in image browsing, and (3) all documentation written in Pharo will integrate more easily, and up to some extent, with external Markdown-related tools. With Microdown we unify the syntax for comments and for books while keeping familiarity for users.

The article is structured as follows: first we describe some of the limits of Markdown and the interesting features of Pillar. Second we stress the design constraints we followed while designing Microdown. Then we present Microdown from a user perspective and finally we show how Microdown will be used in Pharo 90. At the time of writing this article the Pillar toolchain is under adaptation to use Microdown as syntax for books.

IWST '20, Sept, 2020, Virtual

https://doi.org/xxxxxxx

[1] https://casual-effects.com/markdeep/features.md.html

2 MARKDOWN'S LIMITS

In this section we sketch the key limits of Markdown. Some blogs such as https://medium.com/@Mister_Gold/stop-using-markdown-for-documentation-5bda05ad17e3 identifies similar limits.

2.1 Multiple ways to express the same element

Markdown proposes multiple syntaxes to represent an element such as a header. It means that different users may use different variations and get alienated when reading other variations. For example, Listing 1 shows two different syntaxes to define headings of different levels.

```
# Heading level 1
## Heading level 2

Heading level 1
===============
Heading level 2
---------------
```

Listing 1: Different syntax for headers in Markdown

Markdown also supports a lazy mode where some blocks separated by a newline are render as a single one. For example, Listing 2 shows two blockquotes in Markdown that will be merged in a single one because of the lazy mode. Such mode could be good idea but makes users producing documents with mixed types and writing conventions. Some Markdown implementations do even support extensions to disable such behaviour[2].

```
> This is a blockquote.
>     On multiple lines.
That may be lazy.
>
> This is the second paragraph.
```

Listing 2: Lazy mode in Markdown merges two blocks separated by a new line

Having a single syntax without optional (lazy) markup usage for a concept is better since it supports the generation of a more coherent body of documents. In addition it simplifies the parser.

2.2 Fuzzy evaluation semantics

Even if Github's Markdown has a specification as the one available in https://github.github.com/gfm the evaluation semantics of the blocks that compose the language is sometimes unclear. The presence of elements such as blockquotes given its proposed semantics looks like it is available only because the renderer does highlighting or emphasizes the text. However, blockquotes accept nested elements such as a headers or other blockquotes recursively, but the definition of some elements such as references inside nested blockquotes is totally unclear (Listing 3).

```
> This is a paragraph.
>
> > A nested blockquote.
>
> ### Headers work
>
> * lists too
>
> and all other block-level **elements**.
>
> Even code blocks:
>```
>   def hello
>   puts "Hello world!"
>   end
> ```
```

Listing 3: Nested blockquotes in Markdown

For example, should a header nested inside a blockquotes be referenceable and part of a table of contents? Mixing codeblock with blockquotes leads to unexpected results.

2.3 Serious lack of key features

Markdown does not support many important features such as explicit anchor definition. Instead, the writer is forced to change all its references each time a header is changed. Also, figures and code blocks cannot be referred to. They cannot have parameters such as caption for code block or width for figures. This means that the underlying model cannot be efficiently used to generate parametrizable LaTeX.

2.4 No extension mechanisms

Markdowm does not offer coherent and systematic extension mechanisms. In Pillar, as we will show in section 3, there are two simple yet powerful extension mechanisms that plugins can define:

Annotations. Pillar supports the definition of new complex annotations. For example, ${citation=Duca99a|file=rmod.bib}$ is the definition of a citation with arguments in Pillar syntax.

Environments. New environments can be defined.

3 KEY PILLAR FEATURES

Pillar is a document edition toolkit using the a syntax based on the one of Pier (which is the evolution of SmallWiki [DRW05]). The syntax was heavily influenced by the syntax of the original Wiki developed by W. Cunningham. In addition, Pier and Pillar introduced specific extension mechanisms not found in the original Wiki syntax because Pillar was designed to support book writing.

The Pillar syntax supports the traditional features such as: header, list (three kinds), internal links, figures with parameters (to specify anchors, size and any other information), preformatted, paragraph with different formattings, and tables.

In addition Pillar supports features not found in Markdown as listed below and shown in Figure 12:

- *Anchors.* Figures, environments (mathematical environments, environments or code blocks) and headers can define an anchor that can then be referred by links.
- *Annotations.* Annotations, defined using ${ and }$ in text, delimit a tag and a list of parameters. They represent an element not supported by default in the core of Pillar. In Listing 12 inputFile is an annotation for input files. Annotations are used in Pillar to support footnote, slides, columns, citations. A plugin expands its annotations into companion structures that can be handled by various engines (expressed as visitors).

[2]https://github.com/atodorov/Markdown-No-Lazy-Code-Extension

- *Environments.* Environments are similar to annotations but they define document structure. They contain other elements such as headers, paragraph. They are similar to LaTeX environments: they have a start, an end and support parameters to customize them.
- *Meta-data.* Even if annotations could support the definition of metadata Pillar offers a specific syntax for them.

Pillar has been successfully used for wiki, full books, slides, and static web sites.

```
${inputFile:myFile.pillar}$
${inputFile:value=directory/myFile.pillar}$
${footnote:note=Pillar supports the definition annotations.}$


${begin:card}$

! aSection
@anAnchor
A paragraph about the card.
I will be interpreted as part of the card.

${end:card}$
```

Listing 4: Pillar extensibility: annotations and environments with parameters

4 MICRODOWN DESIGN PRINCIPLES

The idea behind Microdown is not to invent yet another markup language but to make the Pillar toolkit more familiar for new users. In other words, Microdown introduces a more powerful and leaner Markdown. Pillar's features and design have proven to be good to support our objectives (in the past slides, books, comments and websites got built with Pillar)[3]. Still, there are several main forces that shape the design of Microdown:

(1) **Strong similarity to Markdown**. It means in particular that Microdown should be subset of Markdown. The idea is that developers should be able to write in a subset of markdown package comments and Pharo browsers can import and display it. The implications are strong. It means that some key elements such as header, links, figures, lists, codeblocks, should be the same syntactically. Even quoteblock whose semantics is unclear should be supported since developers are using them to highlight information. As a general principle we took, even with some reluctance, the Markdown syntax when an element was present in Markdown.
(2) **Small uniform core**. Having a profusion of syntax and markups is possible – MarkDeep is a typical example of this. While such an approach is a possible path, we refrain to take it because we want to make sure that we get the essence of a document so that the same document can be displayed in multiple supports. In addition, we do not want to overwhelm Microdown writers. They also should not be alienated when reading documents written in a specific subset. The uniformity of the corpus is an important design decision. Finally we want to have the simplest implementation possible for maintenance reasons.
(3) **Extensible (supporting books, web sites, comments)**. Microdown should offer evolutions and the design of specific plugins within the same syntax. The experience of Pillar books and slides proved that this is possible to build this on top of a small core.

A tolerant parser to handle Microdown-Markdown mismatches. Although Microdown was designed thinking on Markdown, not all the syntactic alternatives of Markdown are supported by Microdown, and Microdown extensions are not supported by Markdown. This causes a mismatch between both markup languages. We have chosen to handle these cases in the same way Markdown implementations do: use a more tolerant parser.

When developers use features not supported by Microdown, the Microdown parser will parse it as plain text. Microdown writers then get immediate feedback in the in-image tools that do not display them nicely.

5 MICRODOWN IN A NUTSHELL

We present briefly the Microdown syntax. We show the selected Markdown elements and the new syntax introduced for the extensions. In addition we use the following terminology: we name *block* elements those elements that start in a line and may span multiple lines (header, section, paragraph, lists...), and *intext* elements those elements that start and end in the same line such as bold. Block elements elements may contain intext elements, but the inverse does not hold. Intext elements may contain other intext elements. Listing 5 shows an example of all these elements.

Microdown as Markdown or Pillar is based on the identification of a block type based on the first line of the block. But discussed later, the Microdown parsing approach is more robust than Pillar since it follows the Markdown design that scope intext markup termination to block end.

5.1 Mimicking Markdown

The basic principle for Microdown is to follow Markdown when the element is available in Markdown. But it supports only one syntax for one element where possible, and does not support lazy mode.

The following *block* elements are available in Microdown.

Horizontal line. A horizontal line is defined by three stars.

Headers. Headers in Microdown are defined as 1 to 6 # sign followed by the title of the section.

Code blocks. Code blocks are defined using 3 back ticks but in addition support for parameters is available.

Ordered lists. Ordered lists follow the digit period syntax.

Unordered lists. Unordered lists accept the three markdown syntaxes *, - and + but we consider only keeping the first two. Having two options is good for nesting lists but we consider that using multiple nesting level is in general a bad style. The nested list spacing follows the idea that the markup should be aligned of Markdown (it requires two spaces for unordered and three for ordered lists).

Figures. Figures use the Markdown syntax but in addition support parameters as URL parameters, delimited by a ? as shown in Listing 5. There we see that a figure can define a size and a label (anchor) that can explicitly referenced by inline references presented below. We are considering using anchor

[3]Pillar did not reach a strong visibility. In part due to the niche aspect of Smalltalk.

instead of label (which is inherited from LaTeX) to be more regular and close to the anchor block that we introduce (See next Section).

Blockquotes. Blockquotes are syntactically supported using the greater than sign on each line Microdown does not do recursive parsing. In addition, Microdown does render blockquotes closer to preformatted blocks than to quotes.

Tables. Tables follow Github Markdown table extensions but with a stricter form. While in Github Markdown some information in tables is optional, we decided that in Microdown information should be systematically provided. This simplifies the Microdown parser while making all Microdown tables compatible with Github Mardown tables. We plan to support right, left and centred as in Pillar.

The following *intext* elements are kept from Markdown.

Links. Links use the Markdown syntax (...)[url+params] but variables are not supported and will be interpreted as plain text. This point avoids to have a define a scope and other language rules for an optional feature.

Formatting. For in text markup, Microdown supports monospace with one backtick, bold with ** and italic with _. We decided not to support anymore Pillar subscript, superscript and strike since they are barely used.

```
***
A horizontal line
# Microdown Parser and Elements

```language=Pharo&caption=Beautiful&label=code1
Some code
```

[Pharo web site](http://www.pharo.org?alt=Pharo&key2=value2)

\![The famous Pharo Logo](https://files.pharo.org/media/logo/logo.
      png?size=80&label=logo1)

\![This diagram shows...](file:///figures/diagram1.png?width=80&
      label=fig1)

- item1
- item2
  on two lines
- item 3

1. item one
3. item two

> This is
> a quote
```

Listing 5: Markdown selected and extended features.

5.2 Microdown

In addition to the essential Markdown elements mentioned above that are extended to support parameters (codeblocks, figures and links), Microdown introduces several Pillar elements: comments, anchors, references, annotations in text, environments, metadata and math block elements as well as in text as shown in Listing 6. These elements support building advanced documents such as books and slides, and will be usually ignored and rendered as plain text by other Markdown tools.

The following *block* elements are available in Microdown.

Comments. % as first element identifies that the line is a comment.

Anchors. An anchor is a way to make a block element the target of a reference. The main basic usage is for sections. An anchor is a block element and follows the Pillar definition @anchor

Mathematical environment. A mathematical environment is a block accepting LaTeX math syntax and parameters. They are delimited by $.

Meta-data. Microdown express meta-data as STON expressions [CDF+15]. Syntactically they are delimited by { and }.

Environments. An environment is a named block containing potentially other blocks. Microdown syntax for environment is <? and ?>. Environments are not fully stabilised and will probably slightly be changed when the slide support will be migrated from Pillar to Microdown. Currently the use of <? has been favored because it is similar to the one of intext annotations which are the equivalent to environments in intext. Environments are an important extension point. New elements can be supported without introducing new syntax.

The following *intext* elements are kept from Markdown. Since blocks and intext markups live in different scope we decided to reuse when possible the markups. This is the case for math intext and annotations.

References. A reference, syntactically identified by *@mysection@* (we used @ for looks closer to anchors definition), refers to anchors, figures, code blocks, environment and math environments.

Annotations. An annotation is a tagged markup with parameters. They are delimited by <? and ?> as shown by the footnote in Listing 6. With annotations, Microdown can be extended without the need for introducing new syntactical elements.

Intext math. Intext math uses the same markups than the math environment one, i.e., $ and accepts plain math as in LaTeX.

```
%This comment allows us to say that @level1 is an anchor for the
      section.

# This is a level 1 section
@level1

Microdown supports annotations <?footnote | value=A foonote is an
      annotation.?> in text.
Microdown supports reference to anchor *@level1@* and figures (see
      Fig. *@logo1@*)  or environments.

<?slide | title=This is a great slide&key2=value2
   type your content

<?column | width=45
This is a column
?>


?>

{
 "author" : "S. Ducasse"
}
```

```
Microdown supports in text math for example $V_i = C_0 - C_3$ and
      math environment
with parameters.

$key1=value1&key2=value2
V_i = C_0 - C_3
$
```

Listing 6: Microdown specific features.

6 MICRODOWN USES

This section shows the current use of Microdown. Such use is not specific to Microdown and another markup language could be used. Nevertheless we show it since in Pharo 9.0, class and package comments will use Microdown as description language. We expect that having better looking comments will encourage developers to write better comments – This hypothesis will take several years to be validated and it is not the focus of the current article.

6.1 Supporting Nicer Class/Package Comments

Figure 1 shows a class comment in Microdown. Browsers have been extended to support a nicer rendering of such comments with hyperlink to web resources but also in image entities such as classes, methods or packages. Figure 2 shows the same comments displayed to the user.

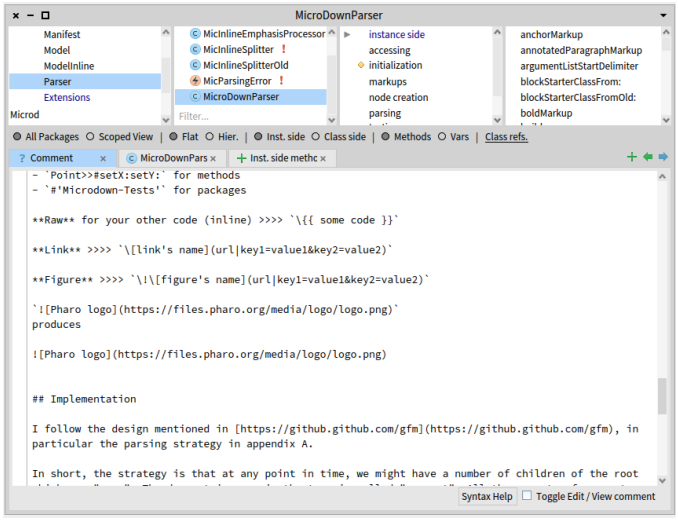

Figure 1: Editing comments using Microdown.

Figure 3 shows the rendering of a comment using mathematical expressions.

6.2 About Hyperlinks to in Image Entities

In addition to displaying comments with a nicer output, the rendering engine takes care to support hyperlinks for navigation. To the question of the introduction of a specific markup or markup family at the level of Microdown, we decided not to modify Microdown but to enhance the rendering engine to analyse monospace in text

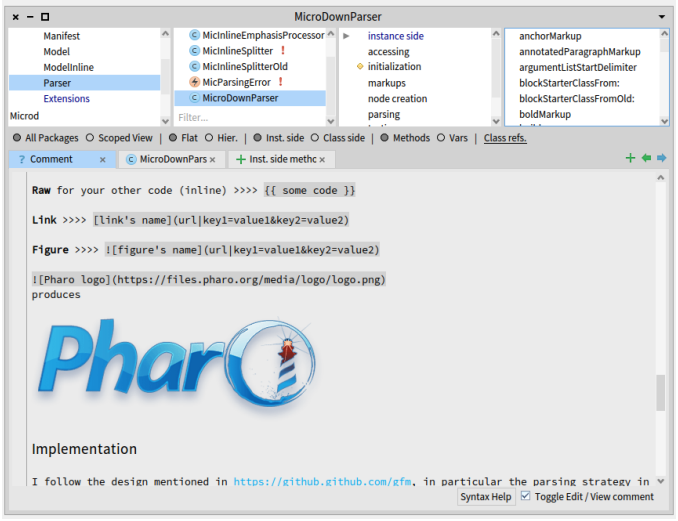

Figure 2: Rendering comments.

and identify when the marked text corresponds to a class, package, method name.

The following conventions are used:

- For class, the analysis is simple: it checks the class name. Hence 'Point' is a link to the Point class. Metaclasses are similarly supported: 'Point class' is a link to the Point class.
- For methods, the analysis supports the exact syntax of direct compiled method access: Point class » #r:degrees:. Clicking 'Point class » #r:degrees:' will bring a message browser on the method r:degrees:.
- For the package, Pharo does not propose any specific syntax. We decided to support the syntax #'Refactoring-Core' to represent navigation to the corresponding package.

When the analysis does not find any of the patterns above, the text is displayed using the monospace conventions.

The net result is that:

- We limit the number of extensions.
- Pharo users can navigate fast and get benefit from Microdown use.
- When the text is edited or rendered with another engine, we do not end up with an orphan (not treated) element.

6.3 Class and Package Level Templates

Even if packages are represented as a class, their comments are important. Since comments are attached to classes representing different concepts such as test cases, project definitions (Metacello Baselines), traits or packages, we designed a templating mechanism. At the meta-level, packages, classes, traits, baselines and test classes define specific comment templates. Users then can extend those templates with comments specific to the commented class. Figure 4 shows that the comments of a test case shows immediately the list of tests. Figure 5 shows that in case of Metacello project Baselines (and similarly for packages containing a baseline) the template displays the baseline definition since it is the most important information. The fact that the displayed information is queried from the actual

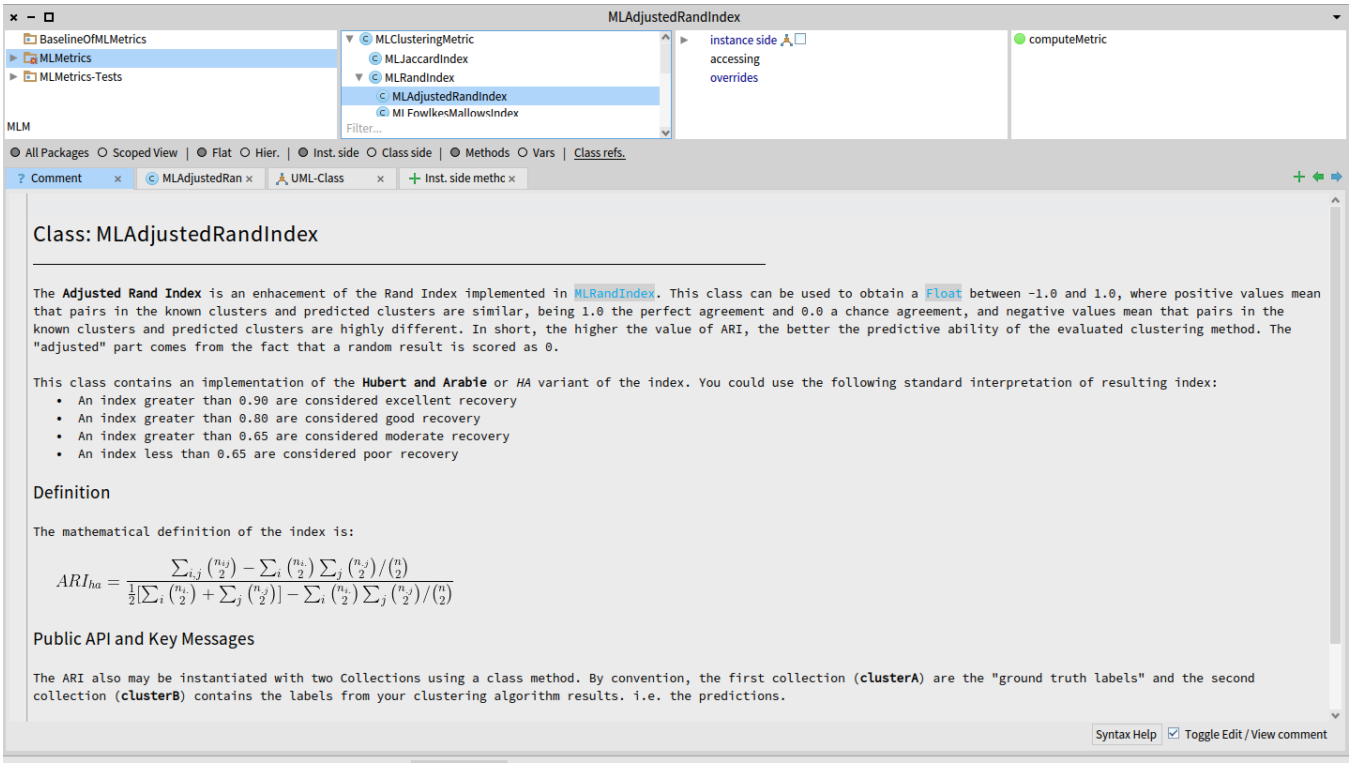

Figure 3: An example of mathematical rendering.

implementation ensures that we do not get obsolete information displayed as part of the comment.

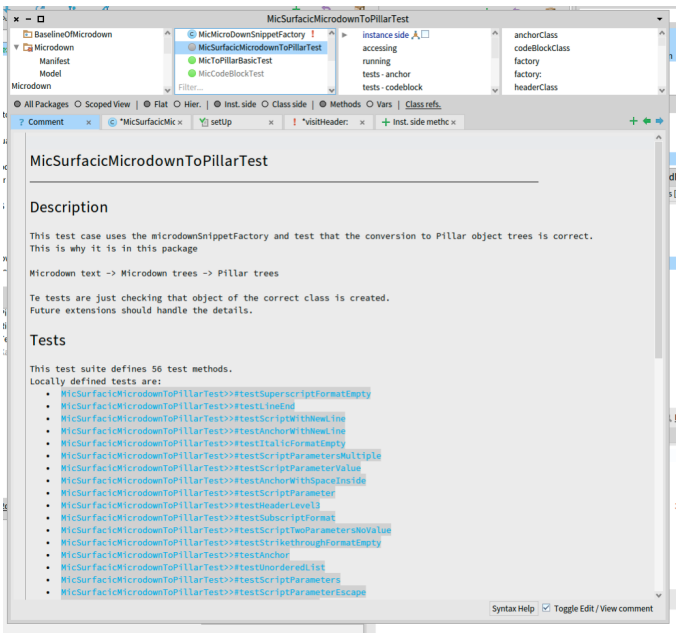

Figure 4: Test classes shows their tests.

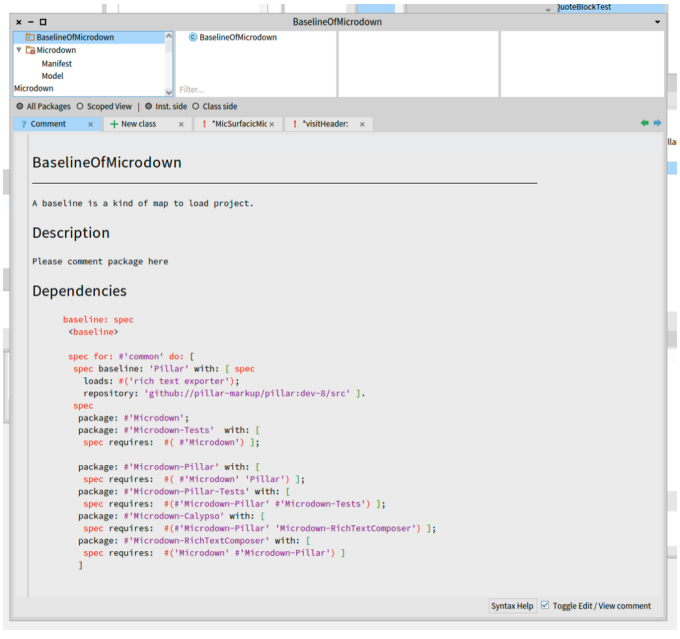

Figure 5: BaselineOf classes and packages show their actual baselines to avoid to have to click to access the information.

6.4 A System Customisation Hook

A Microdown builder is available to let user define and extend the templates without having to be concerned about the production of correct Microdown text. The builder encapsulates the textual representation of the elements.

In addition, we introduced the hook buildMicroDownUsing: aBuilder withComment: aMicrodownString and customized it on packages, classes, baselines, traits and test cases to be able to customize the information that they display. Listing 7 shows the template definition, in particular a baseline shows its baseline automatically as shown in Figure 5. Similarly Listing 8 shows that when browsing a package, the template looks for the companion baseline to display the baseline as shown in Figure 5.

```
BaselineOf class >> buildMicroDownUsing: aBuilder withComment:
      aString
   aBuilder
      header: [ :b | b text: self name ] withLevel: 1;
      horizontalLine;
      text: 'A baseline is a kind of map to load project.';
      header: [ :b | b text: 'Description' ] withLevel: 3;
      text: aString;
      header: [ :b | b text: 'Dependencies' ] withLevel: 3;
      codeblockTag: 'pharo'
      withBody: (self instanceSide sourceCodeAt: #baseline:)
```

Listing 7: Microdown baseline template

```
RPackage >> buildMicroDownUsing: aBuilder withComment: aString

   self class environment
      at: self name
      ifPresent: [ :cls |
         aBuilder
            header: [ :b | b text: self name ] withLevel: 1;
            horizontalLine;
            text: 'A baseline is a kind of map to load project.';
            header: [ :b | b text: 'Description' ] withLevel: 3;
            text: aString;
            header: [ :b | b text: 'Dependencies' ] withLevel: 3;
            codeblockTag: 'pharo'
               withBody:
               (cls
                  sourceCodeAt: #baseline:
                  ifAbsent: [ 'No baseline! Houston we have a
     problem' ]) ]
      ifAbsent: [ aBuilder
            header: [ :b |
               b
                  text: 'Package: ';
                  text: self name ]
               withLevel: 1;
            horizontalLine;
            text: aString ]
```

Listing 8: Microdown package template

7 IMPLEMENTATION

Microdown[4] is implemented as a parser generating a Pillar document model. The Microdown parser produces a Microdown tree which is then transformed into Pillar document elements. Such elements are then later on rendered using a rich text renderer.

[4]http://pillar-markup/microdown

7.1 Microdown AST

The Microdown AST is decomposed in two main groups: the *block level* elements such as comment, lists, and the *intrablock* elements such as bold, link...

Block-Level Elements. Listing 9 shows the hierarchy of the Microdown block-level element. Only the leaves of the inheritance tree are instantiated and represent a document.

```
MicAbstractBlock #(#parent #children #parser)
   MicAbstractAnnotatedBlock #(#label #isClosed #body #firstLine)
      MicAnnotatedBlock #()
   MicContinuousMarkedBlock #(#text)
      MicCommentBlock #()
      MicQuoteBlock #()
      MicTableBlock #(#rows #hasHeader)
   MicListBlock #(#indent)
      MicOrderedListBlock #(#startIndex)
      MicUnorderedListBlock #()
   MicListItemBlock #(#text)
   MicParagraphBlock #(#text)
   MicRootBlock #()
   MicSingleLineBlock #()
      MicAnchorBlock #(#label)
      MicHeaderBlock #(#level #header)
      MicHorizontalLineBlock #()
   MicStartStopMarkupBlock #(#isClosed #body #firstLine)
      MicEnvironmentBlock #(#arguments #name)
      MicMetaDataBlock #()
      MicSameStartStopMarkupBlock #(#arguments)
         MicCodeBlock #(#firstTag)
         MicMathBlock #()
```

Listing 9: Microdown block-level AST.

We introduced several parsing abstractions to factor out the behavior of an element accepting a new child or not. Such abstractions are handy to factor out the logic.

- MicContinuousMarkedBlock is for elements whose any line starts with the same markup such as table using |.
- MicSingleLineBlock is for element which are only composed of a single line such as horizontal line or header.
- MicStartStopMarkup and MicSameStartStopMarkupBlock are respectively for element delimited by a starting and closing markup that is either different or the same.

IntraBlock Elements. Listing 10 describes the hierarchy of intrablock elements. Each one corresponding to one (e.g. ‘, $ or two markups (e.g. `![]()`)

```
MicAbstractInlineBlock #(#start #end #kind #children #substring)
   MicAbstractInlineBlockWithUrl #(#url)
      MicFigureInlineBlock #(#parameters)
      MicLinkInlineBlock #()
   MicAnchorReferenceInlineBlock #()
   MicAnnotationInlineBlock #(#name #arguments)
   MicBasicInlineBlock #()
   MicBoldInlineBlock #()
   MicItalicInlineBlock #()
   MicMathInlineBlock #()
   MicMonospaceInlineBlock #()
   MicRawInlineBlock #()
   MicStrikeInlineBlock #()
```

Listing 10: Microdown intrablock AST.

The Microdown parser does not accept lazy definition of element parts. In addition, there is only one way to do something, except for bulleted list where * and - are accepted for compatibility with Markdown.

The Microdown parser follows the Markdown parsing approach in which the first line of a block allows the identification of a block. In addition, Microdown takes advantage of the Markdown parsing approach in which a block delimits a context in which intext elements are scoped. This approach limits error propagation, makes the parser more robust and provides an overall nice user experience.

7.2 Block-based contexts

Microdown implementation follows the design described in https://github.github.com/gfm, in particular the parsing strategy.

In short, the strategy is that at any point in time, we might have a number of children of the root which are open. The deepest open one in the tree is called current. All the parents of current are open.

When a new line is read we do the following:

(1) Check if the new line can be consumed by current. As part of this, a child of current can be made which can consume the new line. For example, when consuming "‘ the root block node will create, a new code block that will become current and consume the body of the "‘ element then will close.
(2) If current cannot consume the new line, we close current, move current to its parent, and repeat 1.
(3) The root node can consume anything, for instance by making new nodes for storing the new line.
(4) The root node is not closed until input is exhausted.

The following listing 11 is the core logic. Each block level element redefines canConsumeLine: and in combinaison with addLineAndReturnNextNode: implements a kind of state machine of the acceptance of new children by the current element.

```
handleLine: line

   | normalized |
   normalized := line copyReplaceAll: String tab with: String space.
   [ (current canConsumeLine: normalized)
      ifTrue: [ ^ current := current addLineAndReturnNextNode:
      normalized ]
      ifFalse: [ current closeMe ].
   current := current parent.
   self handleLine: normalized ] on: Error do: [ self
     handleErrorInLine: line ]
```

Listing 11: Microdown parsing logic.

7.3 Handling of broken formatting

The logic for handling broken formatting *e.g.,* opened but not closed or closed without an opening is an important point for simple markups. It limits the propagation of an error to a block. An error cannot span further down elements and provides a stability in document processing by handling locally error.

Let us illustrate this point by comparing with Pillar parsing approach. The Pillar parser does not handle well the non closing of formatting. For example, listing 12 produces a paragraph with unclosed monospace part that spans over two paragraphs. This leads to LaTeX generation errors that are difficult to handle at the level of the LaTeX exporter.

```
A part without monospace ==beginning

Another paragraph
```

Listing 12: Example of broken document.

In Microdown as well as in Markdown the same situation does not produce a broken document. Listing 13 produces a valid document composed of two paragraphs each with a part having an uninterpreted monospace text.

```
A part without monospace `beginning

Another paragraph closing `end
```

Listing 13: Example of non broken document

8 CONCLUSION

In this article we presented Microdown a clean and lean Markdown supporting extensibility mechanisms that make it suitable to write full books such as Pharo by Example [BDN+09]. We show some of the drawbacks of Markdown and key features of Pillar. Microdown takes a good pieces of both using the syntax of Markdown to lower the learning curve and improving familiarity. We show that Pharo 9.0 will use Microdown for rendering comments in rich text as well as navigation facilities. Future work is to perform another pass on the infrastructure to make sure that all the Pharo documentation, books and slides can be expressed in Microdown. The ultimate goal being to be able to load full chapters as documentation within Pharo.

Acknowledgments. The authors are grateful to Kasper Osterbye first proof of concept of Markdown parsing and Rich text rendering. Thanks a lot, you help us! We want to thank Damien Pollet and Oleksandr Zaitsev for their discussions.

REFERENCES

[ADCD16] Thibault Arloing, Yann Dubois, Damien Cassou, and Stéphane Ducasse. Pillar: A versatile and extensible lightweight markup language. In *International Workshop on Smalltalk Technologies IWST'16*, Prague, Czech Republic, August 2016.

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example.* Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[CDF+15] Damien Cassou, Stéphane Ducasse, Luc Fabresse, Johan Fabry, and Sven Van Caekenberghe. *Enterprise Pharo: a Web Perspective.* Square Bracket Associates, 2015.

[DRW05] Stéphane Ducasse, Lukas Renggli, and Roel Wuyts. SmallWiki — a meta-described collaborative content management system. In *Proceedings ACM International Symposium on Wikis (WikiSym'05)*, pages 75–82, New York, NY, USA, 2005. ACM Computer Society.