The design and implementation of a userfriendly object-oriented markup language

Titouan Vervack

Supervisor: Prof. Marko van Dooren
Counsellor: Benoit Desouter

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Vakgroep Toegepaste Wiskunde, Informatica en Statistiek
Chair: Prof. dr. Willy Govaerts
Faculty of Engineering and Architecture
Academic year 2015-2016

UNIVERSITEIT GENT

The design and implementation of a userfriendly object-oriented markup language

Titouan Vervack

Supervisor: Prof. Marko van Dooren
Counsellor: Benoit Desouter

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Vakgroep Toegepaste Wiskunde, Informatica en Statistiek
Chair: Prof. dr. Willy Govaerts
Faculty of Engineering and Architecture
Academic year 2015-2016

UNIVERSITEIT GENT

Preface and acknowledgement

Permission for usage

"The author(s) gives (give) permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use. In the case of any other use, the copyright terms have to be respected, in particular with regard to the obligation to state expressly the source when quoting results from this master dissertation."

Titouan Vervack, June 2016

The design and implementation of a userfriendly object-oriented markup language

by

Titouan VERVACK

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2015–2016

Supervisor: Prof. dr. ir. Marko van Dooren
Counsellor: Dr. ir. Benoit Desouter

Faculty of Engineering and Architecture, Ghent University

Department of Applied mathematics, computer science and statistics, Chair: Prof. dr. Willy Govaerts

Abstract

A lot of solutions exist for document creation. These are split up into What You See Is What You Get (WYSIWYG) editors and markup languages. Both of these provide a lot of advantages as well as disadvantages. WYSIWYG editors immediately show the final document and help the user by providing a GUI for all the features. Markup languages put greater emphasis on letting the user structure a document properly, by providing specialised syntax for certain structural elements for example. Because of this some solutions are easy to read and write while others offer a lot of customisability, usually through some programming model. However, a good combination of both is hard to find. As such the goal of this thesis is to design a markup language that is easy-to-read and write as well as providing a modern programming model to allow for full customisation of the document. An introduction to the state-of-the-art solutions for document production is given (Chapter 1) in which the (dis)advantages of said solutions are discussed. This is followed by an introduction to the new markup language, its features and the design decisions that were made (Chapter 2). Then several proofs of concept for document types and libraries build with the language are presented (Chapter 3). The development of the compiler for the language, its features and the faced challenges are then discussed (Chapter 4). The text concludes with possible future work on this work (Chapter 5), as well as a short reflection and summary on the work (Chapter 6).

Keywords

Markup, Object-oriented, LaTeX, Markdown

Contents

List of Figures

Listings

Chapter 1

Introduction

In this thesis we introduce a new markup language that improves upon a few others, namely LaTeX and Markdown, whilst still holding on to their advantages.

The language has been used to write this entire book [21] with the exclusion of the title page and a few images. The source code of the book is available on the UGent GitHub. In the same repository you also find the Neio library and source code for the Neio compiler.

Before we get into the details of this new language it is necessary to introduce some of the most used markup languages and word processors.

1.1 State-of-the-art

Before we discuss the state-of-the-art, we first discuss the difference between saving a document in a human-readable or a binary format.

Markdown, LaTeX and Pandoc use a human-readable format that can be opened in any editor, is quite robust to file corruptions and is well suited for Version Control Systems (VCS) such as Git and Mercury. It is robust in the sense that a if a single bit of the document gets flipped, the document can still be repaired by the user. The user can find the error on his own, or he can use the variety of editors to find help him as every editor can handle errors differently.

When the syntax gets more extensive and less human-readable, the chances to encounter a problem upon compilation increase. This is both good and bad, if a part of the syntax got corrupted, the compiler tells you and you know that the document has been corrupted. Had it not warned the user, the document would be in a corrupt state without the user noticing so. On the other hand the corruption can prevent the document from successfully compiling, forcing the user to find out what happened to the document. In any case however, the error can still be tracked down and repaired.

This is not always possible, when using a binary format, like the ones used by default in What You See Is What You Get (WYSIWYG) editors for example. As a matter of fact, we created a Word document and flipped one bit in a random byte using a hex editor. We then tried to open the document and received the following sequence of errors and were unable to retrieve the contents of the file.

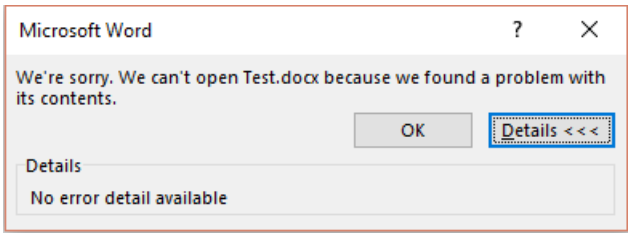


Figure 1.1: The detection of an error

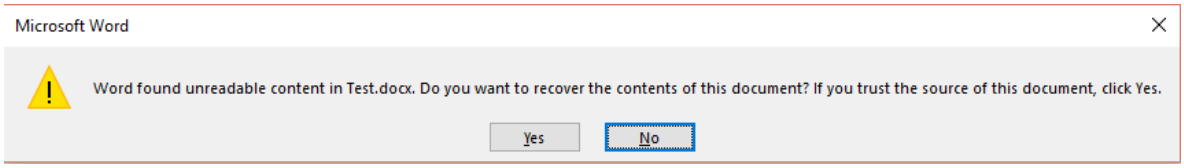


Figure 1.2: The warning before trying to repair the error

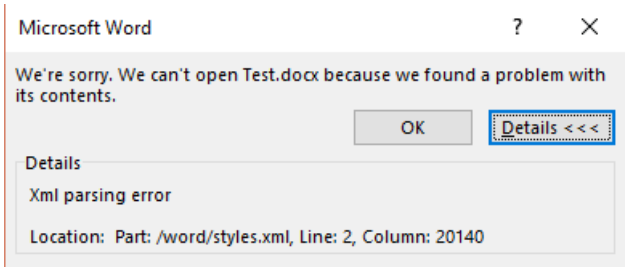


Figure 1.3: The inability to repair the error

A regular user, without technical expertise, is not be able to repair this error as the source code is not human-readable. There are also almost no other tools to help the user as the binary format can only be interpreted correctly by a few editors.

It is to be noted that the error handling depends on what exactly was corrupted, in some cases the corruption can be automatically repaired.

1.1.1 Word processors

The best-known solution for creating documents, are so called WYSIWYG editors, examples include 'Microsoft Word' and 'Pages'. They are called as such because you immediately, without compiling, see what your final document is going to look like. Because of this, the complexity of the tool is significantly decreased allowing anyone to use the tool. It is less complex because you do not have to type in commands and wait for a preview to change. Instead you select the property from a drop-down list or click a button and the document is updated instantly. The GUI essentially hides the complexity.

A WYSIWYG editor gives the author freedom, for example it is very easy to change a font or change the font size of a certain part of the text and you can easily add an image and drag-and-drop it around.

A lot of these editors are grouped into packets, such as Office or LibreOffice. Using one of these packets, it is also easy to insert slides, tables or another document [22]. These are then be editable with the specialized program for it (like Excel), but you do not have to leave the

document. In the example below an Excel chart has been inserted into a Word document.

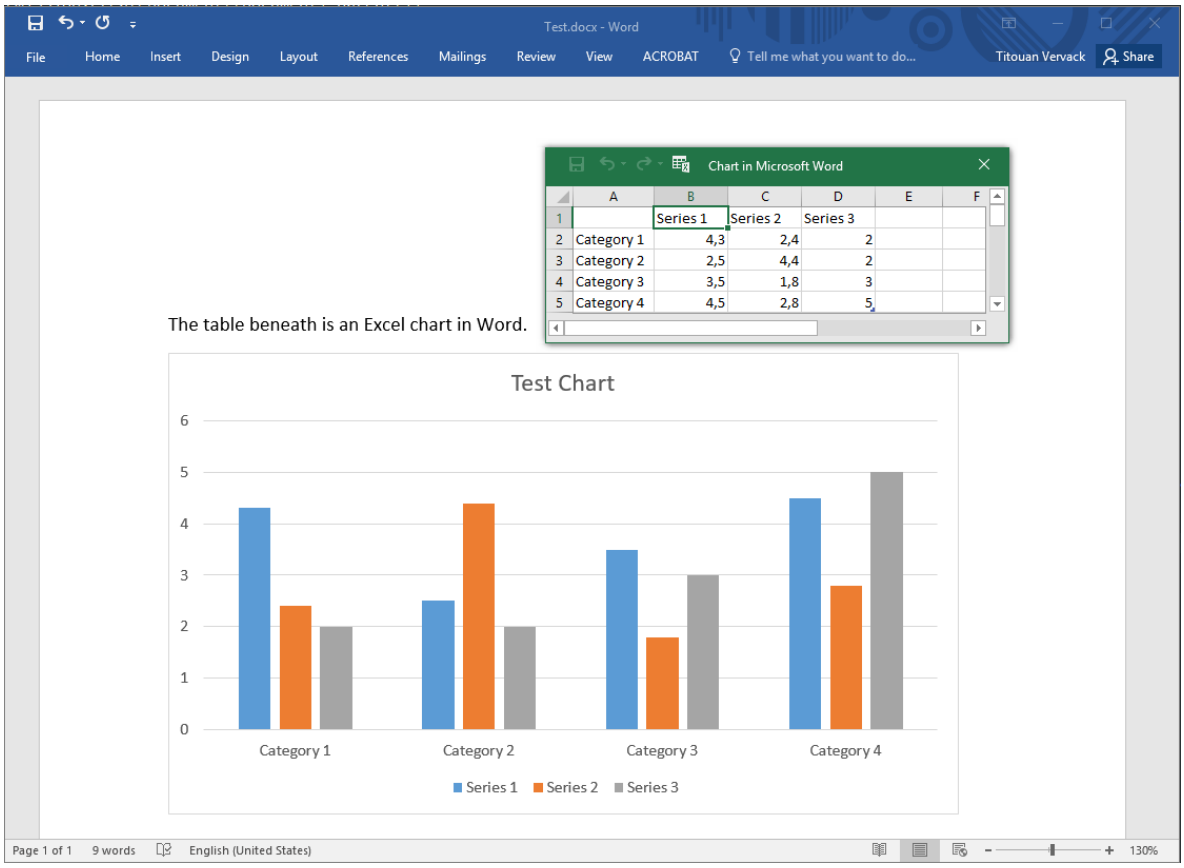


Figure 1.4: An Excel chart in a Word document

Collaboration is also made easy by using online versions of these solutions, such as 'Microsoft Word Online' or 'Google Docs'. These make documents cross-platform and allow multiple people to work on them at the same time. It is also possible to add comments or suggestions on (online) documents. These changes can then be resolved in just one click by an authorized editor.

Some documents are however more complex and have need for a more powerful tool such as LaTeX.

1.1.2 LaTeX

LaTeX provides rich customisability through its Turing complete programming model. The model is different from modern programming models, it does not use regular variables or functions, instead it uses macros. A macro or command looks like the following: `\cmdname[opt-arg]{req-arg}`. Here `cmdname` is the name of the command, `opt-arg` are the optional arguments and `req-arg` are the required arguments. To show how commands work, we show the process involved in processing some input that includes a command.

In LaTeX, every character and command in the input is a token, `a` is a token but `\command` is also a token. The input is transformed into a list of tokens and is then processed token after token. When a command is encountered, it is expanded.

Expansion removes the current token from the list and adds every expandable item in its body to the token list recursively. This is best shown using an example.

We define two commands using `\newcommand` and call the `\y` command.

```
1 \newcommand{\x}{a command}
2 \newcommand{\y}{This is \x}
3 \y
```

The output is `This is a command`. The step-by-step expansion is given below:

1. LaTeX encounters the token `\y` without arguments and expands it.
2. `\y` is removed from the token list and `This is \x` is added to it.
3. `This is` is read and `\x` is encountered, `\x` gets expanded.
4. `\x` is removed from the token list and `a command` is added.

We can also pass arguments to a command by specifying the amount of arguments as an optional argument. We can then access the arguments by using the `hash` symbol followed by a number. For example, the output of code beneath is: `This is an argument`.

```
1 \newcommand{\command}[1]{This is #1}
2 \command{an argument}
```

The braces around the first required argument are optional, thus we can also write this as follows:

```
\newcommand\command[1]{This is #1}
```

To elaborate further on expansion, we have a look at another example. We have the string `Hello` saved in a command and want to concatenate a string to it so that it prints out `Hello world`. To do this, we have to redefine the command, this can be done using `\renewcommand`. We thus get the following:

```
\newcommand\example{Hello}
\renewcommand\example{\example{} world}
```

The {} at the end of `\example` is used as a delimiter, to know when the `\example` command ends, such that the space behind it does not disappear. This example does not work however as `\example` is recursively calling the new definition instead of calling the old one once. To counter this, `\example` in the new definition should be expanded first, we can to that using `\expandafter` as shown below.

```
\newcommand{\example}{Hello}
\expandafter\renewcommand\expandafter\example\expandafter{\example{} world}
\example
```

`\expandafter` removes the first two tokens behind itself from the token list, expands the second token and then adds the expanded token and the first token back to the token list.

The step-by-step expansion for the previous example is given below:

1. `\example` is defined for the first time.
2. The second definition begins.
3. `\expandafter` removes `\renewcommand` and `\expandafter` and expands the latter.
4. The second `\expandafter` removes `\example` and `\expandafter` and expands the latter.
5. The third `\expandafter` removes { and `\example` and expands the latter to `Hello`.
6. `\renewcommand, \example, { and Hello` are placed back on the token list.
7. The token list (`\renewcommandexample{Hello world}`) is now expanded normally.

Using this programming model, libraries, called packages can be created. Over the course of the past 31 years (the initial LaTeX release was in 1985), a lot of them have been created. They offer all kinds of functionalities, ranging from the creation of sheet music and slide shows [27] to providing the base to write letters and books. This is one of the reasons why LaTeX is often used for long and complex documents such as books, scientific papers and syllabi.

The goal of LaTeX [13] is for the user to focus on the structure and content of the document and to let designers worry about the design of the document. To make sure the document looks good afterwards, it is important to structure it using elements such as sections and paragraphs. Because of the freedom given in WYSIWYG word processor, it is often forgotten to use styles, headers and so on. This can cause inconsistencies to appear as the document grows.

Designers can use the very sophisticated typesetting of LaTeX to create designs for well structured LaTeX documents. The typesetting improves readability by supporting kerning, ligatures,... and using the advanced Knuth-Plass line-breaking algorithm [11]. The algorithm sees a paragraph as a whole instead of using a more naive approach and seeing each line individually. Kerning places letters closer together or further away depending on character combinations. A ligature is when multiple characters are joined into one glyph.

Figure 1.5: Kerning on

Figure 1.6: Kerning off

Figure 1.7: Ligature on

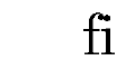


Figure 1.8: Ligature off

LaTeX also knows how to work with bibliographies by using programs such as Bib(La)TeX or Biber. These are reference management programs that are very robust and make certain that all of your references are consistent. They allow you to create the references outside of your main document, decreasing the amount of clutter in the document.

Whilst on the topic of including other files in your document, in LaTeX you can split up your document into multiple smaller documents, allowing for easier management. A common use of this function is to write every chapter, for example in a book, in a separate document and than all the chapters are imported into the main document. This allows to easily remove or switch out different parts of a document. This is also possible in Pandoc (as you can inline LaTeX) and the WYSIWYG editors [22].

Finally, LaTeX employs a free and open source model, which is a big reason why it is so popular in the open source community. As a result of this model, combined with its popularity and good scientific support, LaTeX has been integrated in many other applications. A few examples are:

- MediaWiki [17] [28]: a free and open-source software package that powers sites as Wikipedia [31] and Wiktionary [32].
- Stack Exchange [30]: a network of Q&A web sites
- JMathTex [9]: a Java library adds functionality to display mathematical formulas in a Java application

LaTeX is powerful but it is also complex and has a steep learning curve, because of this, simpler alternatives such as Markdown have been created.

1.1.3 Markdown

The goal of Markdown [15] is to be easy-to-write and read as a plain text document and as such is actually based on plain text emails. Its appeal lies in its simplicity as it allows to for example easily create a simple document, such as a short report or a blog post.

It achieves this simplicity, by introducing a small and simple syntax [16] that feels natural. For example, to create a title you underline it with `minus` or `equals` characters. To create an enumeration, you use `star` characters as bullet points. An example Markdown document is shown below:

Listing 1.1: document.md

```
Markdown
========

It is:
* Simple
* Easy to read
* Easy to write
```

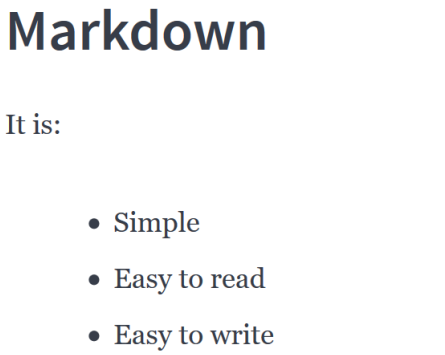


Figure 1.9: The document rendered as HTML by the official Markdown conversion tool

This syntax decreases the amount of possible syntax errors and allows for short compile times and wide editor support. It also allows Markdown to be easily translated into other formats such as PDF or HTML. As such, a tool to convert Markdown to HTML is offered by the designers of the language itself. This tool is also cross-platform and as such widens the target audience of Markdown.

However, for a lot of cases Markdown is too simple. Because of this, solutions such as Pandoc have been created.

1.1.4 Pandoc

Pandoc [23] is a document converter, but it can do much more than just convert a document from one format to another. Notably, it allows you to write inline LaTeX in a Markdown document, combining the power of both. An example is shown below.

Listing 1.2: pandocExample.md

```
# Pandoc
This pandoc document uses \LaTeX{} and
    markdown in one file!

\begin{equation*}
\sqrt[3]{125}=25
\end{equation*}
```

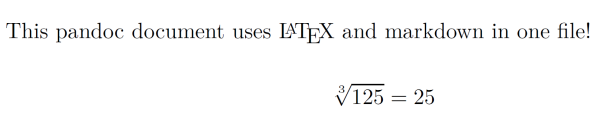


Figure 1.10: The output of the document to the left.

However, outside of the LaTeX one, Pandoc does not really have a programming model and instead works with so called filters. A document in Pandoc can be read and transformed into an Abstract Syntax Tree (AST). The AST is read in a filter, then transformed and passed on to the next filter.

Filters can be written in a multitude of languages such as Haskell, Python, Perl,... . They are then passed as a command line argument to the conversion command. We illustrate this using an example from the Pandoc tutorial on scripting [24].

Listing 1.3: behead2.hs

```
#!/usr/bin/env runhaskell
-- behead2.hs
import Text.Pandoc.JSON

main :: IO ()
main = toJSONFilter behead
  where behead (Header n _ xs) | n >= 2 = Para [Emph xs]
        behead x = x
```

(Pandoc, 2016)

The example above shows a filter written in Haskell. It replaces all the headers of level 2 or more in a Markdown file by paragraphs with the text in italics (hence the `Emph` in the code). The `toJSONfilter` function constructs a JSON representation of the AST. Using this, we find the headers of level of 2 or more and create a new paragraph. The document can be created using the following command `pandoc -f SOURCEFORMAT -t TARGETFORMAT --filter ./behead2.hs`.

This example shows the power of Pandoc: it can do its work directly on the AST. This means that the source document does not have to be a Markdown document, it could also use HTML or any other supported input format.

Using another one of these filters, PanPipe [29], you can execute programs defined outside of a document. The stdout of the program is inserted into the document and the stderr is redirected to the stderr of Pandoc. An example is shown below.

```
This document executes a shell command.

```{pipe="sh"}
echo "Hello world! I am " $(whoami)
```
```

This documents executes a shell command.

Hello world! I am titouan

Figure 1.11: The output of the document to the left.

Using Pandoc, you can also declare templates that allow you to use variables. The value of the variables is predefined (such as `$body$`) or passed in through the command line. These can then also be used in conditionals or loops in this template. You can see an example below.

Listing 1.4: letterTemplate.latex

```
\documentclass{letter}
\signature{Titouan Vervack}
\begin{document}
\opening{Dear $addressee$,}

$body$

\closing{Sincerely,}
\end{letter}
\end{document}
```

Listing 1.5: letter.md

```
How have you been?
I've been longing to see you, but I've been very busy...

My son's getting married next week and I would like to invite you!
Well, you and your wife of course.

If you're planning on coming, please give me a call.
```

The document can then be build with the following command.

```
pandoc letter.md -o letter.pdf --template letterTemplate --variable=addressee:"John Doe"
```

1.2 Problem statement

We can see that these solutions all have their advantages, but there are also quite a few disadvantages. We discuss the most common ones for every solution underneath and afterwards we introduce the problem statement.

1.2.1 Word processors

The UI in WYSIWYG editors allows you to customise the document, you can insert other documents, tables, figures,... and adjust a lot of properties. It is also possible to use macros using for example Visual Basic for Applications in Word to automate some tasks. It can for example be used to remove leading tab characters from paragraphs.

However, it has trouble with inserting variables. Because of this, it is very cumbersome to compute a simple expression like `x + y`. To do this, you first have to open the Visual Basic Editor. In there you create a new module in which you define your new variables as follows.

```
1    ActiveDocument.Variables.Add Name:="x", Value:=27
2    ActiveDocument.Variables.Add Name:="y", Value:=2
3    ActiveDocument.Variables.Add Name:="result", Value:=(Int(ActiveDocument.
         Variables("x")) + Int(ActiveDocument.Variables("y")))
```

The result variable can then be inserted by selecting `Quick parts` in the `Insert` tab, selecting `Field`, then selecting `DocVariable` and finally typing in `result`. This inserts the `result` variable into the document, but the variable is not automatically update. To achieve that you have to right-click it and click `Update field`.

In word processors small changes can also have big side effects on the document. For example, you placed an image exactly where you wanted and later on, you add a newline somewhere higher up. The reflowing of the text underneath it shifts the precisely placed image out of place. These changes are not always immediately clear, which creates inconsistencies in the document.

Another problem that occurs with images (thought it is not an inherent problem of WYSIWYG editors, rather a bad implementation), is that the caption does not move together with the image. In the example below an image is inserted and a caption is (by right-clicking the image and pressing `Add caption`) added. The image is then dragged to the end of the file and we see that the caption remains in place.

This is the first paragraph.

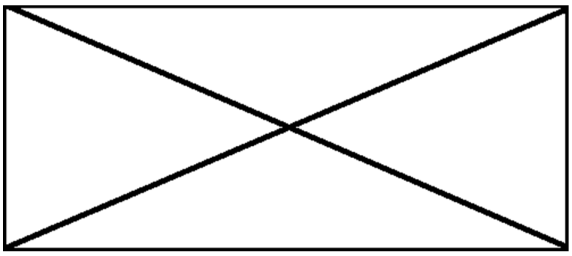


Figure 1 This is the first figure

This is the second paragraph.

Figure 1.12: Caption below the image

This is the first paragraph.

Figure 1 This is the first figure

This is the second paragraph.

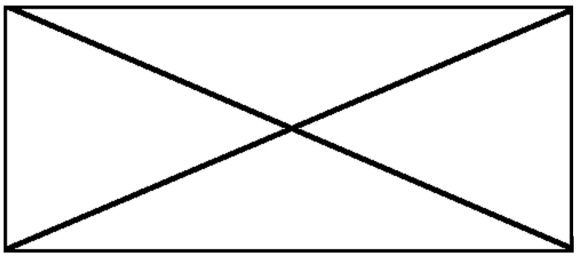


Figure 1.13: The image moved, but the caption did not.

1.2.2 LaTeX

Even though LaTeX offers customisability through a programming model, it is not perfect. The programming model is hard to use and read, creating a steep learning curve. It also makes LaTeX seem complex, scaring away a lot of potential users. A way to counter this complex look, is LyX [14]. This is a GUI around LaTeX offering some of the most common WYSIWYG features, while still utilizing the strength of LaTeX. Even though the complexity can be hidden under a GUI, it is still there and error messages are as bad as they were before, as is discussed beneath.

Having a cumbersome programming model, LaTeX makes it easy to create syntax errors. When these are made LaTeX shows an error message, but these are often unclear. See the excerpt below for an example.

```
1 \documentclass{article}
2 \begin{document}
3 \begin{equation}
4 $a$
5 \end{equation}
6 \end{document}
```

```
1 line 4: Display math should end with $$
    .<to be read again>a $a
2 line 5: You can't use `\eqno' in math
    mode.\endequation ->\eqno\hbox {\
    @eqnnum }$$\@ignoretrue \end{
    equation}
3 line 6: Missing $ inserted.<inserted
    text>$ \end{equation}
```

The message means that $ is not allowed inside of an equation, but that is not clear from the

error message.

As mentioned in Subsection 1.1.2, the macro model does not work as we are used to in imperative and functional languages, we have to think about the order of expansions. Keeping up with these expansions is confusing and requires a lot of work.

Next to this, we also have to watch out for name clashes, because LaTeX does not uses namespaces. The `newcommand` we saw in Subsection 1.1.2, requires that the command has not been defined earlier, if it was, an error is thrown. `renewcommand` requires that the command was defined earlier, it overwrites the previous command and it crashes if the command was not yet defined. Lastly, `providecommand` acts as `newcommand` if the command was not yet defined and does nothing if it was already defined, leaving the old definition intact. For example, we can not define a command called `name` using `newcommand` as this command already exists, we have to use `renewcommand` to this. This should be used with care however, as every command that uses `name` now uses the newly defined `name`.

Lastly, LaTeX does not use a static type system while this could provide us with information needed for refactoring and auto-completion based on the context.

1.2.3 Markdown

Markdown is simple but pays a hefty price for it. All of the syntax elements have been hard-coded into the language meaning there is no room for customisability. The only way to customise Markdown is by embedding inline HTML. This allows to create a solid structured model but it still does not allow for very much customisability. There is no programming model that can provide the customisability.

As an example, we use HTML to create a table, as these are not a part of default Markdown (though there are variations that have syntax for them, like GitHub Flavored Markdown [7]).

```
Below you can find an HTML table
<table border="1">
    <tr>
        <td>
            Element 1
        </td>
        <td>
            Element 2
         </td>
     </tr>
</table>
```

Below you can find an HTML table

Element 1	Element 2

Figure 1.14: The table document rendered as HTML by the official Markdown conversion tool

Adding HTML to Markdown decreases readability and cleanliness of your document significantly, which goes against the goal of Markdown. Using it also introduces the need for a certain technical proficiency to read and edit the source document. Finally, HTML reintroduces the possibility of syntax errors and thus slows down the document creation.

1.2.4 Pandoc

Pandoc allows you to execute exterior programs through the PanPipe filter, but the only result is the stdout and stderr of the program. These do not allow you to change the output afterwards because you have no more structure. The structural elements such as sections and paragraphs have disappeared. For example, you create a binary tree using a 3rd party application. Then you want to insert a new value into this tree, which requires it to be rebalanced. However this is impossible as you lost all of the structure of the tree, it is only represented as a string. You thus have to redraw every instance of the tree instead of creating it once, programmatically transforming it and redisplaying the transformed tree.

To access the structure of the document, the AST can be used. Transforming an AST afterwards however, does not allow for precise control either. In Listing 1.3 we saw how we could replace all of the headers of a level higher than 2, but changing it for only a few would require us to find them. If we could have done this inline or if we could redefine the meaning of the `#` symbol, the problem would be avoided.

Finally, the variables used in Pandoc have some limitations. They can not be defined inline, only through the command line and they can only be used in templates. This means that

you can not easily embed variables and execute simple expressions such as `x + y`.

1.3 Proposed solution: Neio

To counter the problems occurring in the state-of-the-art solutions whilst retaining their advantages, we created a new markup language called `Neio` (read as neo). The name `Neio` was chosen because `neo` means new and we are developing a new language. The spelling however, came from the Colemak keyboard layout [26], the letters on the home row, beneath the right hand, spell `n-e-i-o`.

Based on these solutions, the following goals were created for the Neio markup language:

1. It has to be user-friendly and easy to get started with the language;
2. It has to use a modern programming paradigm;
3. It has to be highly customisable.

The first goal is achieved by using a syntax like Markdown for the documents.

To achieve the second goal we note that a document has a strong structure consisting of a lot of elements like chapters, sections, paragraphs and so on. An object-oriented model, like the one in Java, is able to represent this structure well by using an object for each element and inheritance allows us to easily specialize an element, for example change the numbering of an enumeration. We also chose this model because it is well known and as seen in Subsection 1.1.2 and Subsection 1.1.4, working with macro expansions or filters is not ideal.

Because it offers a powerful programming model that we can translate to, we chose to translate to LaTeX. It has also been used successively for decades and has proven that it can deliver very clean looking documents. Later on more back ends, such as HTML, could be implemented, but that is outside of scope of this thesis.

1.3.1 Target group

Neio targets anyone that is currently using Markdown but wants to customise their document further. For example, using Neio you can create citations and references, which is not possible in Markdown.

By offering a simpler syntax than used in LaTeX, we target LaTeX users. Finally, we target developers by offering a well-known, modern programming model that is more conventional than the macro model in TeX.

Chapter 2

Design of the Neio markup language

In Chapter 1 the state-of-the-art concerning document creation has been discussed and we concluded that improvements can be made. In this chapter the Neio language is presented. We explain how our decisions were reached and how they were affected by the state-of-the-art.

2.1 Neio document

The typical files that a user writes are called Neio documents; they have to be as simple as possible. These files use the `.no` extension. Based on our state-of-the-art analysis, we chose a Markdown-like syntax for Neio documents.

To illustrate some of the basic concepts we present an example Neio document.

Listing 2.1: A simple Neio document

```
// This is a Neio document
[Document]

/* Below we define a chapter
 * and a paragraph.
 */
# Chapter 1
This is the first paragraph.
```

1 Chapter 1

This is the first paragraph.

Figure 2.1: The rendered document

Even if you are not familiar with Markdown, you can immediately tell what this document represents. It creates a document, a chapter (through the `#` symbol) and a paragraph, and it contains two comments.

A difference with Markdown, but a resemblance with LaTeX, is that every Neio document starts out with document class, it tells us what kind of document we are building. This is done to improve customisability and to be able to support multiple kinds of documents.

In Neio there are two sets of syntax called text- and code mode. Listing 2.1 is written in text mode, the code files in next section are written in code mode.

2.2 Code files

Code files are very similar to class files in Java. They also use the `.no` extension. As means of an introduction we have a look at the `Paragraph` class that is used to represent the paragraph in Listing 2.1.

Listing 2.2: Paragraph.no

```
namespace neio.stdlib;

import neio.lang.*;

/**
 * Represents a paragraph
 */
class Paragraph extends Content;

// private
Text text;

```

```
13 /**
14  * Intialises a paragraph with some text
15  *
16  * @param parText The initial text of the paragraph
17  */
18 Paragraph(Text text) {
19     this.text = text;
20 }
21
22 /**
23  * Appends some text on a new line within the same paragraph
24  *
25  * @param t The text to add after the newline
26  * @return This paragraph with a newline and {@code t} added to it
27  */
28 Paragraph appendLine(Text t) {
29     return appendText(new Text("\n").appendText(t));
30 }
31
32 /**
33  * Appends Text to this Paragraph
34  *
35  * @param t The Text to add
36  * @return This paragraph with {@code t} added to it
37  */
38 private Paragraph appendText(Text t) {
39     if (text == null) {
40         text = t;
41     } else {
42         text = text.appendText(t);
43     }
44     return this;
45 }
46
47 /**
48  * Creates a newline handler to handle newlines following a Paragraph
49  *
50  * @return A new {@code ParNLHandler} with this Paragraph as parent
51  */
52 ParNLHandler newline() {
53     return text;
54 }
55
56 /**
57  * Returns the LaTeX representation of this Paragraph.
58  * It is the empty string in case there is no Text in this paragraph.
59  *
60  * @return The LaTeX representation of this Paragraph
61  */
62 String toTex() {
63     if (text != null) {
64         return "\\par " + text.toTex();
65     } else {
66         return "";
67     }
68 }
```

Except for some variations on the syntax, such as using `namespace` instead of `package`, and the lack of access level modifiers (which can be specified), this is valid Java code. The absent

access level modifiers are automatically set to `private` for members and `public` for methods. They do not have to be written explicitly as the default value is usually what the developer wants and it makes the code file just a little clearer and more concise.

To keep code files simple and to maximize reusability (as well as making the parsing somewhat easier), some functionality from the Java language was dropped. It is for example not possible to create anonymous classes as these are not reusable and do not provide any new functionality. It is also not possible to create multiple classes in a single file.

As we see in the next section, code files are used to build a Neio document.

2.3 Call chain

A problem with Markdown is that the semantics for the syntax are hard-coded in the language, a `#` always represents a chapter for example. But if you are creating a slide show for example, you want `#` to create a new slide instead of a chapter. For this reason, text mode in Neio has no hard-coded semantics.

Instead, the semantics of the syntax are defined in code files. For example, the `#` in a `Document` is defined as follows:

Listing 2.3: Document.no

```
1  /**
2   * Creates a Chapter and adds it to this
3   *
4   * @param title The title to use for the new Chapter
5   * @return The newly created Chapter
6   */
7  Chapter #(Text title) {
8      Chapter chapter = new Chapter(title, 1);
9      addContent(chapter);
10
11     return chapter;
12 }
```

Now we can see that a Neio document is actually a sequence of method calls. This is referred to as the `call chain`. An example of a document and its call chain is shown below.

```
[Document]

# Chapter 1
This is the first paragraph.
```

```
new Document()
    .#("Chapter 1")
    .text("This is the first paragraph
        .");
```

Note that the call chain is not code that the user has to write, its just a representation of a Neio document.

The user is thus actually creating an object model. The object model for the previous document is shown below. The calls that create every object are shown on the edges.

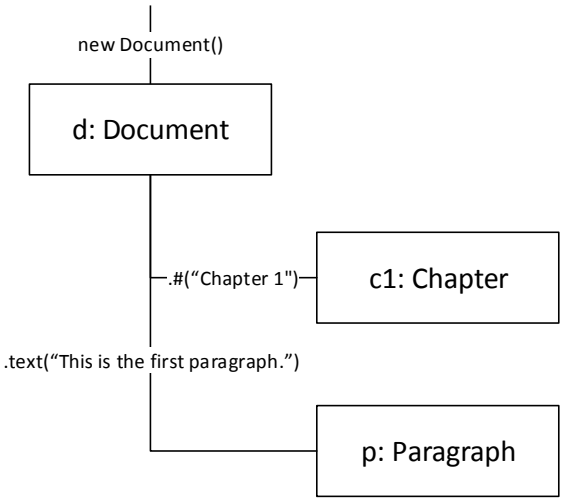


Figure 2.2: The object model of the previous document.

This object model is separated as much as possible from the visualisation, which happens later on. We say "as much as possible" because it is not completely separated. For example, sometimes it is specified that two elements should be placed next to each other in the object model.

Every element that is visualised in the final is a `Content`. `Content` is the base object of Neio, it is like `Object` in Java.

To be able to customise the semantics of the syntax in text mode, the Java syntax in code files has been modified to allow for special method identifiers. These are discussed in the next

section.

2.4 Method modifications

To be able to write the Neio documents we have seen so far, the regular Java methods used in code files had to be modified slightly. We also had to reserve two method names that are not reserved in Java. The modifications and the reasons for the modifications are given below.

2.4.1 Symbol methods

In Listing 2.3 we saw that the code file, used a hash symbol as a method name. These methods are called `symbol methods`. All the symbols that can be used as method names are `#, -, *, _, $, |, =, ^, ‘`.

These methods have either no arguments or a `Text` argument. This is done to keep the method calls invisible for the user as more or different arguments would have to passed explicitly. The `Text` that is passed to the method is the text following the call in text mode. An example of both, and the call chain, is shown below.

```
1 [Document]
2 # Chapter 1
3 #
```

```
1 new Document()
2     .#("Chapter 1")
3     .#()
```

Next to symbols, there are two method names that have been reserved. The first of them is `text` and is explained in the next section.

2.4.2 Text

Whenever text in a Neio document is encountered on its own, for example a paragraph but not the text behind a `#` symbol, the `text` method is called with the text as an argument. It is a separate method to allow continuation of the call chain and because not every block of text is necessarily a paragraph, it can for example be a quote.

The reason we use `Text` instead of just a Java `String` is because text can be marked up. How this can be done is explained in the next section.

2.4.3 Surround methods

Like in Markdown, we want to be able to markup text by placing it between a pair of characters. For example, putting text in bold by surrounding it with a pair of `star` characters. With the features that we have seen so far this is not yet possible.

To achieve this surround methods were developed. A surround method is a method that is annotated with the `surround` modifier. When text in text mode is surrounded with a pair of symbols, then a surround method is called. A few examples of such methods are given below.

Listing 2.4: Text.no

```
/**
* Creates a new {@code BoldText} and appends it to this
*
* @param t The text to make bold
* @return The newly created BoldText
*/
public surround Text *(Text t) {
    return appendText(new BoldText(t));
}

/**
* Creates a new {@code ItalicText} and appends it to this
*
* @param t The text to make italic
* @return The newly created ItalicText
*/
public surround Text _(Text t) {
    return appendText(new ItalicText(t));
}

/**
* Creates a new {@code MonospaceText} and appends it to this
*
* @param t The text to show in a monospace font
* @return The newly created MonospaceText
*/
public surround Text `(Text t) {
    return appendText(new MonospaceText(t));
}
```

The text `This is *bold* text` is thus transformed into the following call chain `text("This is ").text(*("bold")).text(" text")`. Throughout the text the surround method ` is

used a lot, it shows the text in a monospaced font.

As said in Subsection 2.4.1, the argument of symbol methods is a Text, this means we can use surround methods in a symbol method. A more extensive example of surround methods is shown below.

Listing 2.5: surroundEx.input

```
1 [Document]
2 # `_Chapter 1_`
3 This `Chapter` is written in italic and
      monospace font.
4
5 # _Chapter 2_
6 This *Chapter* is written in italic.
```

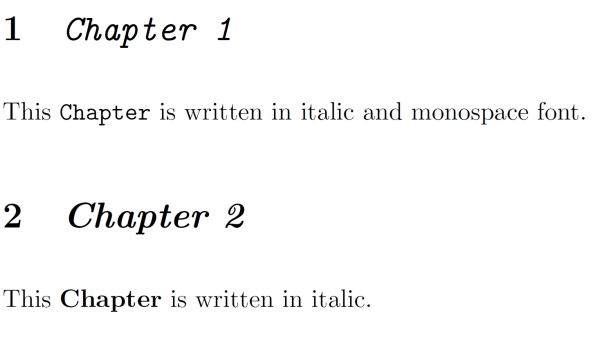


Figure 2.3: The rendered version of the document to the left

To not confuse surround methods with the symbol methods, the first and last characters inside a surround call can not be a space. The text after a symbol method on the other hand, is always separated from the symbol by at least one space.

2.5 Nested methods

Documents often have recursive elements such as sections or enumerations. With what we have seen thus far, we would have to create a new method for every level of recursion. For example, to create sections, subsections and subsubsections, we have to create the following methods: `#, ##, ###`.

This is of course very cumbersome and even impossible if there is no limit on the recursion. The behaviour of such recursive elements is also very similar. Usually, only one property, such as the numbering and indentation in an enumeration, is affected by the level of recursion. This is why nested methods were developed.

A method can be annotated with the `nested` modifier. Such a method implicitly takes an

extra argument, an `Integer` that reflects the depth of this recursive method. The depth of this recursive method is the number of times the symbol has been used. If we have a look at the `Chapter` class, we see that it defines a nested method `#`.

Listing 2.6: Chapter.no

```
/**
 * Creates a new Chapter and adds it to this.
 * If the level of the new Chapter is lower or equal
 * to the level of this, the new Chapter is added to
 * the nearest TextContainer above this.
 * This is a nested call thus only ##+ will match.
 *
 * @param title  The title of the new Chapter
 * @param level The level of nesting of the new Chapter
 */
nested Chapter #(Text title, Integer level) {
    if (level <= this.level) {
        Chapter c = nearestAncestor(Chapter.class);
        if (c != null) {
            return c.hash(title, level);
        } else {
            return nearestAncestor(Document.class).hash(title);
        }
    }
    Chapter chapter = new Chapter(title, level);
    addContent(chapter);

    return chapter;
}
```

If we now call `chapter.##("Chapter 1.1")` for example, it translates to the following call: `chapter.#("Chapter 1.1", 2)`.

A nested method only matches with at least two symbols, the single symbol has to be defined separately. The first reason for this is because the first level often requires some initialisation. A second reason is given once context types have been discussed. An example of such initialisation is shown below.

Listing 2.7: TextContainer.no

```
/**
 * Creates a new Itemize and adds an ItemizeItem to it
 * The Itemize is added to this.
 *
 * @param text The text to use for the ItemizeItem
 * @return The created ItemizeItem
 */
Itemize *(Text text) {
    Itemize itemize = new Itemize();
```

```
10      ItemizeItem item = new ItemizeItem(text, itemize, 1);
11      itemize.*(item);
12
13      addContent(itemize);
14      return itemize;
15  }
```

When a `*` call is encountered for the first time, an `Itemize` has to be created The following `*` calls then simply add an `ItemizeItem` to it.

The last method name that was reserved is `newline`. It is explained in the next section.

2.5.1 Newlines

Users often use varying amount of newlines to express different structures. For example, the following two examples do not have the same meaning.

Listing 2.8: Two lists

```
1  * Item 1
2  * Item 2
3
4  * Item 3
```

Listing 2.9: One list

```
1  * Item 1
2  * Item 2
3  * Item 3
```

In Listing 2.8 there are two lists while in Listing 2.9 there is only one list. The same is done to separate paragraphs.

Because of this, and to include as much information as possible in the call chain, even the newline character is a method. The newline character is defined by the following W3C Extended Backus-Naur Form (EBNF) [33]:

```
1  newline ::= "\r?\n"
```

An example of a document and its call chain including the newline method is shown below.

```
[Document]
The first line
of the first paragraph.

The first line
of the second paragraph.
```

```
new Document()
    .newline()
    .text("The first line")
    .newline()
    .text("of the first paragraph.")
    .newline()
    .newline()
    .text("The first line")
    .newline()
    .text("of the second paragraph.")
```

This example shows that we never explicitly create a paragraph, this is done depending on the context. We explain the example step-by-step to understand how the document is build.

The first `newline()` is defined in `TextContainer`, the super class of `Document`, and creates an intermediate object that is an instance of `NLHandler`. The code for this method is shown below.

Listing 2.10: TextContainer.no

```
/**
 * Handles newlines
 *
 * @return Returns a new NLHandler
 */
NLHandler newline() {
    return new NLHandler(this);
}
```

`NLHandler` defines the `text` method (shown below), which creates a paragraph and adds it to the parent, in this case the `Document`.

Listing 2.11: NLHandler.no

```
/**
 * Creates a Paragraph and adds it to the parent.
 *
 * @param text The text for the Paragraph
 * @return        The newly created Paragraph
 */
Paragraph text(Text text) {
    new Paragraph(text) par;
    parent().addContent(par);
    return par;
}

/**
 * Returns this
 *
 * @return this
```

```
17  */
18 NLHandler newline() {
19     return this;
20 }
```

The second `newline()` is defined in Paragraph. It creates a new `ParNLHandler` which also defines a `text` method. This `text` method (shown below) however does not create a new `Paragraph` but appends to the existing one.

Listing 2.12: ParNLHandler.no

```
1 /**
2  * Appends some Text to an existing Paragraph
3  *
4  * @param text The Text to add
5  * @return The existing Paragraph with {@code text} appended to it
6  */
7 Paragraph text(Text text) {
8     return parent().appendLine(text);
9 }
```

Next there are two `newline()` calls. As we saw the first one creates a new `ParNLHandler`, but `ParNLHandler` also defines a `newline` method (given below). This method returns the `NLHandler` defined by the parent (in this case `Document`) of the existing `Paragraph`.

Listing 2.13: ParNLHandler.no

```
1 /**
2  * Calls the newline method of the parent of the existing Paragraph
3  * Newline is only called in text mode thus Paragraph certainly has a
       TextContainer parent
4  *
5  * @return The NLHandler of the TextContainer that is parent of the existing
       Paragraph
6  */
7 NLHandler newline() {
8     return (TextContainer) (parent().parent()).newline();
9 }
```

The next `text()` is thus called on an instance of `NLHandler` and as said before, creates a new `Paragraph`. The following `newline()` and `text()` append some `Text` to the new `Paragraph` using a `ParNLHandler`, as seen before.

By introducing these special method identifiers we actually extended fluent interfaces [6]. Because of this, the user is actually programming without noticing it.

In Chapter 3 we see that we can for example create tables as the one below.

Listing 2.14: A table created using a DSL.

```
|      Student club       | Rounds | Seconds/Round | Dist (km) | Speed km/h |
-------------------------------------------------------------------------------
| HILOK                   |  1030  |       42      |   298,70  |   24,89    |
| VTK                     |  1028  |       42      |   298.12  |   24.84    |
| VLK                     |   841  |       51      |   243.89  |   20.32    |
| Wetenschappen and VLAK  |   819  |       53      |   237,51  |   19.79    |
| VGK                     |   810  |       53      |   234.90  |   19.58    |
| Hermes and LILA         |   793  |       54      |   229.97  |   19.16    |
| HK                      |   771  |       56      |   223.59  |   18.63    |
| VRG                     |   764  |       57      |   221.56  |   18.46    |
| VEK                     |   757  |       57      |   219.53  |   18.29    |
| VPPK                    |   689  |       63      |   199.81  |   16.65    |
| SK                      |   647  |       67      |   187.63  |   15.64    |
| Zeus WPI                |   567  |       76      |   164.43  |   13.70    |
| VBK                     |   344  |      126      |    99.76  |    8.31    |
```

Student club	**Rounds**	**Seconds/Round**	**Dist (km)**	**Speed km/h**
HILOK	1030	42	298,70	24,89
VTK	1028	42	298.12	24.84
VLK	841	51	243.89	20.32
Wetenschappen and VLAK	819	53	237,51	19.79
VGK	810	53	234.90	19.58
Hermes and LILA	793	54	229.97	19.16
HK	771	56	223.59	18.63
VRG	764	57	221.56	18.46
VEK	757	57	219.53	18.29
VPPK	689	63	199.81	16.65
SK	647	67	187.63	15.64
Zeus WPI	567	76	164.43	13.70
VBK	344	126	99.76	8.31

This table is not an image that was included, but is actually created as shown in Listing 2.14. We do not need to create a special reader or to add new syntax to be able to implement this table. Instead we create a Domain Specific Language (DSL) to build tables using the features we have shown so far.

To avoid clutter, the `newline` method is not shown in the further examples unless it is needed

to understand the example.

2.6 Context types

Every method in the call chain returns an object, on which the rest of call chain is called. However, we do not want to define the same symbol methods for all of these objects, for example defining `#` in a `Document`, a `Chapter`, a `Paragraph` and so on. We also do not want to specify on what object every method in the call chain is called.

For this reason context types were developed. They use the object model that has been built so far to find out which object roots the next method call.

We illustrate this using an example.

```
[Document]
# Chapter 1
This is the first paragraph.
# Chapter 2
```

```
new Document()
    .newline()
    .#("Chapter 1")
    .newline()
    .text("This is the first paragraph
        .")
    .newline()
    .#("Chapter 2")
```

As we saw before, `Document` creates a `NLHandler` through the `newline` method. However, `NLHandler` does not have a `#` method. `#` in this example is called on `Document` instead and creates a `Chapter`.

`Chapter` also creates a `NLHandler` when `newline` is called on it and as we saw `NLHandler` defines `text`, which creates a `Paragraph`.

`Paragraph` creates a `ParNLHandler` when `newline` is called on it, but `ParNLHandler` has no `#` method. The last `#` is again called on `Document`. The object model for this example is shown below.

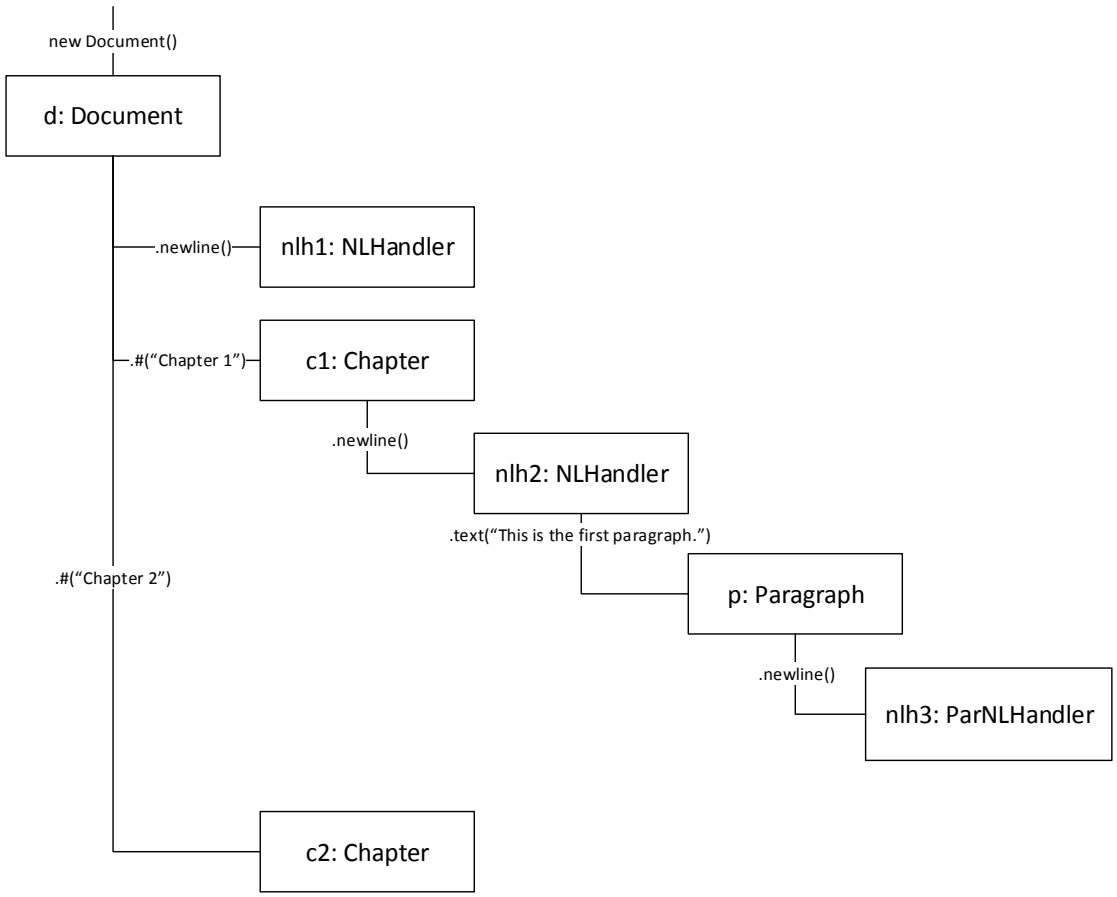


Figure 2.4: The object model of the previous example.

Sometimes the new method is called on the previous object, sometimes its called on an object that was made a lot earlier. Context types are used to find the object that roots the next method call. To understand how they do this, we go over the example step-by-step .

A context type is created every time a new method call (in the call chain) returns. It has a context, which is the context type in which the root object of the method call was found, and an actual type. The latter is the object that was returned by the new method call.

When we encounter `new Document(), the first context type, called ct0`, is created. There are no context types yet, so there is no context and the actual type is `Document`. The context type is shown below.

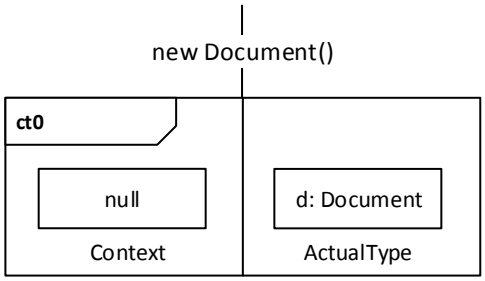


Figure 2.5: The first context type: ct0.

When we now encounter `newline()` we have to do a `context lookup`. This lookup recursively searches through previous context types for the object that roots the method call.

The context lookup starts by checking if the actual type of the previous context type, `ct0`, defines a method `newline`. As it does, we do not have to check the context of `ct0`.

The context of the new context type, `ct1`, is `ct0` and because `newline()` creates a `NLHandler` the actual type is `NLHandler`. The object model, with context types, now looks as follows.

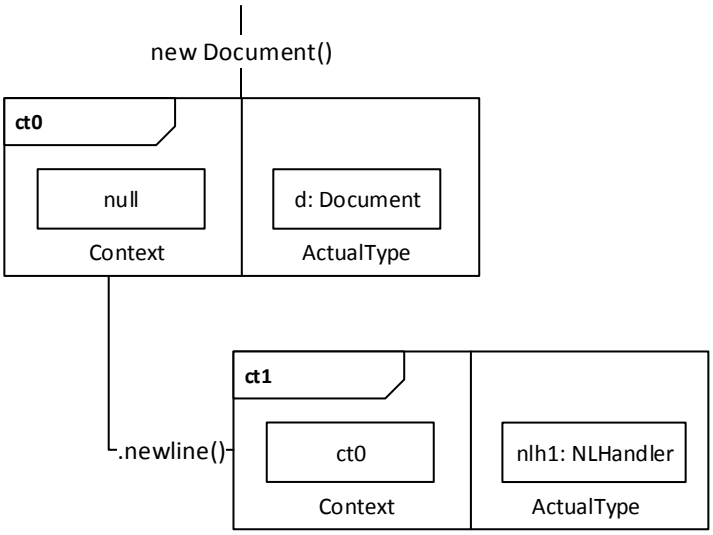


Figure 2.6: The object model of ct0 and ct1.

The next call is `#("Chapter 1")` and the previous context type is `ct1`. To know on what to call `#` we first check if the actual type of `ct1` defines a `#` method. It does not, thus we check if the context of `ct1` does so instead.

The context of `ct1` is `ct0`, thus we first check if the actual type of `ct0` defines `#`. The actual type is `Document` and this defines `#`. The new context type, `ct2`, is thus created with `ct0` as context. Because the `#` in `Document` returns a `Chapter`, the actual type is `Chapter`. The object model is shown below.

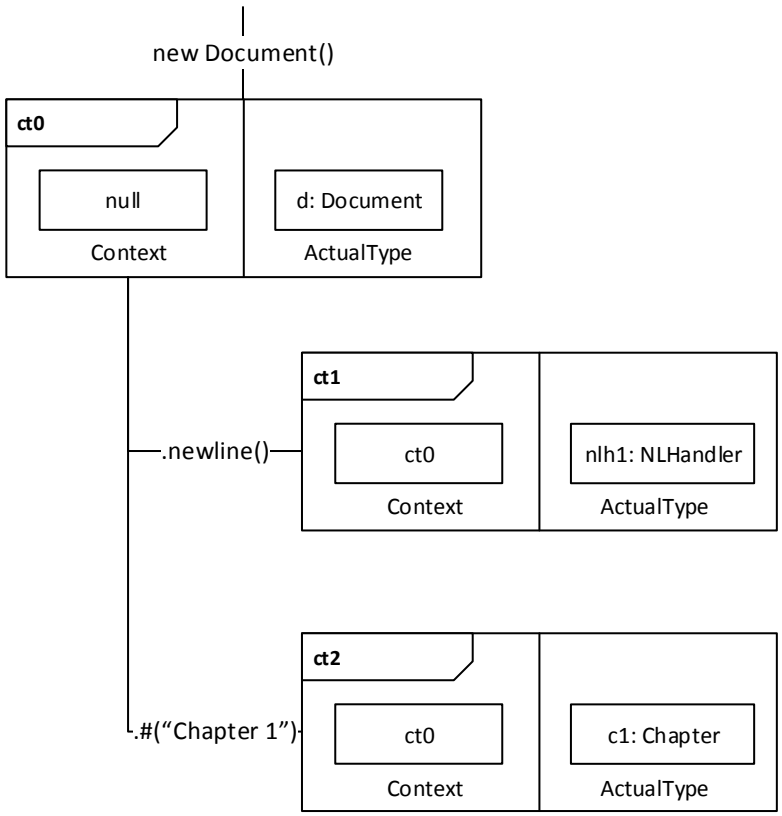


Figure 2.7: The object model of ct0 up to ct2.

The following call is `newline()`, we check the actual type of `ct2`, `Chapter`, and we know that it defines `newline`. We thus create `ct3` with `ct2` as context and `NLHandler` as actual type. `ct4` and `ct5` are created with respectively `ct3` and `ct4` as context and respectively `Paragraph` and `ParNLHandler` as actual type. The object model up to `ct5` is shown below.

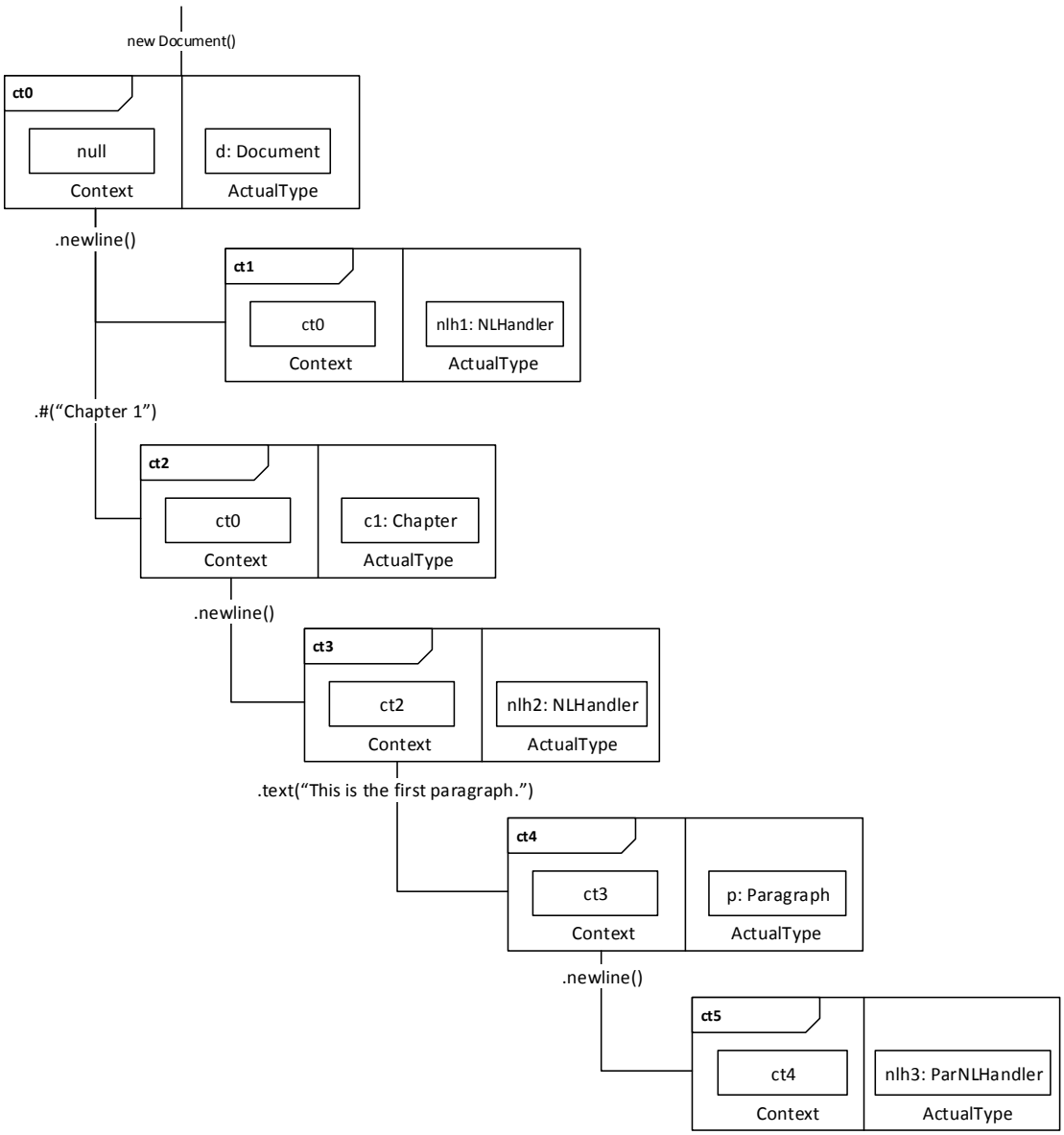


Figure 2.8: The object model of ct0 up to ct5.

Finally, `#("Chapter 2")` is called. For this call we recursively search back up to `ct0` as it is the first context type in which `#` is defined. `(Par)NLHandler, Paragraph` and `Chapter` have no definition of `#`. The last context type is thus build with ct0 as context and `Chapter` as a actual type. The final object model is shown below.

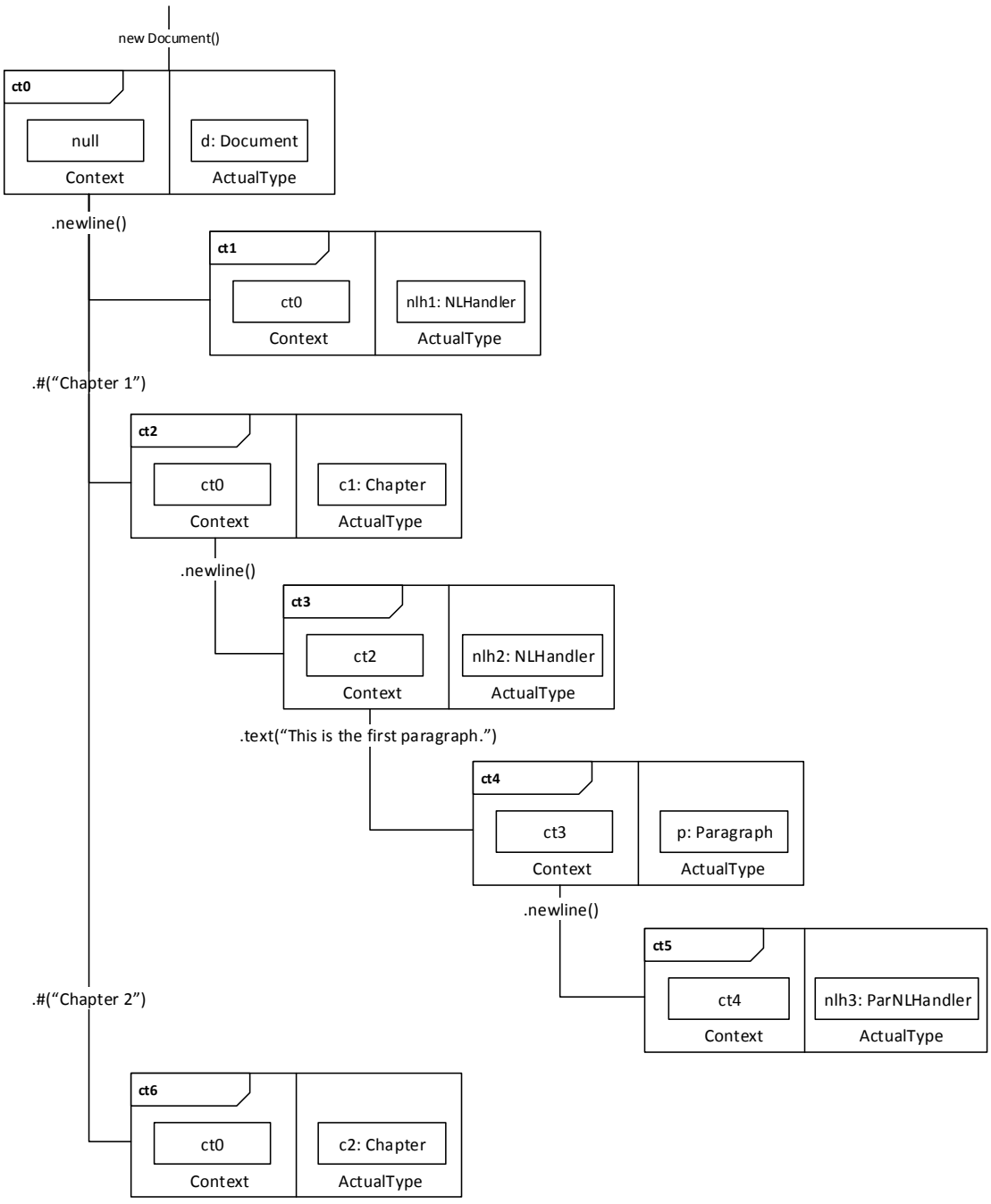


Figure 2.9: The object model of ct0 up to ct6.

It is important to note that we do not carry the entire context along for the entire document. In the last step of the previous example, we did a `context reset`. After a context reset a

part of the object model becomes inaccessible for future method calls in the call chain. This is needed to correctly create the object model.

To show why context resets are important we give another example:

Listing 2.15: An example for context resets

```
[Document]
* Item 1
* Item 2
Paragraph 1
* Item 3
```

```
new Document()
    .*("Item 1")
    .*("Item 2")
    .text("Paragraph 1")
    .*("Item 3")
```

The example has the following object model. The methods that cause a context reset have been coloured red.

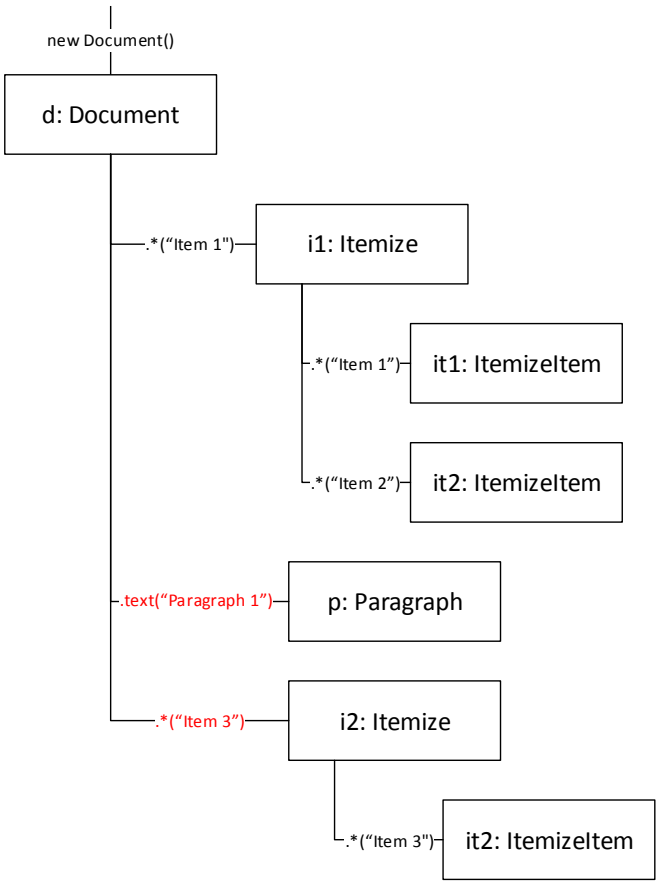


Figure 2.10: The object model of the previous example.

The * is defined in `Document` creates `Itemize i1`. The * defined in `Itemize` creates `ItemizeItems`. Because of context resets, the last * is called on `Document`. This document thus contains an itemize of two items, a paragraph and an itemize of one item. The correct context lookup for the last * is visualised on the call chain below, the arrows depict one step in the context lookup.

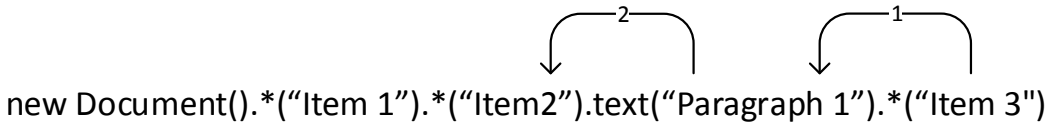


Figure 2.11: Correct context lookup for *

If there were no context resets and the entire context was thus kept in every step, `Item 3` would find the * defined in `Item 2` before it reached `Document`. This means that there would only be one list, that contains all three items, instead of two lists as was meant in the document. This wrong context lookup for the last * is visualised on the call chain below.

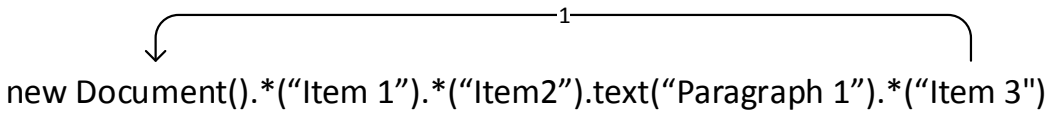


Figure 2.12: Wrong context lookup for *

Nested methods

Now that we know how context types work, we can explain why the first level of a nested method is separated from the rest. We explain why this is needed using the following example:

```
1 [Document]
2 # Chapter 1
3 ## Chapter 1.1
4 # Chapter 2
```

```
1 new Document()
2     .#("Chapter 1")
3     .#("Chapter 1.1", 2)
4     .#("Chapter 2")
```

We want `Chapter 1` and `Chapter 2` to be children of `Document` and `Chapter 1.1` should be a child of `Chapter 1`. The context type representation of the document is shown below.

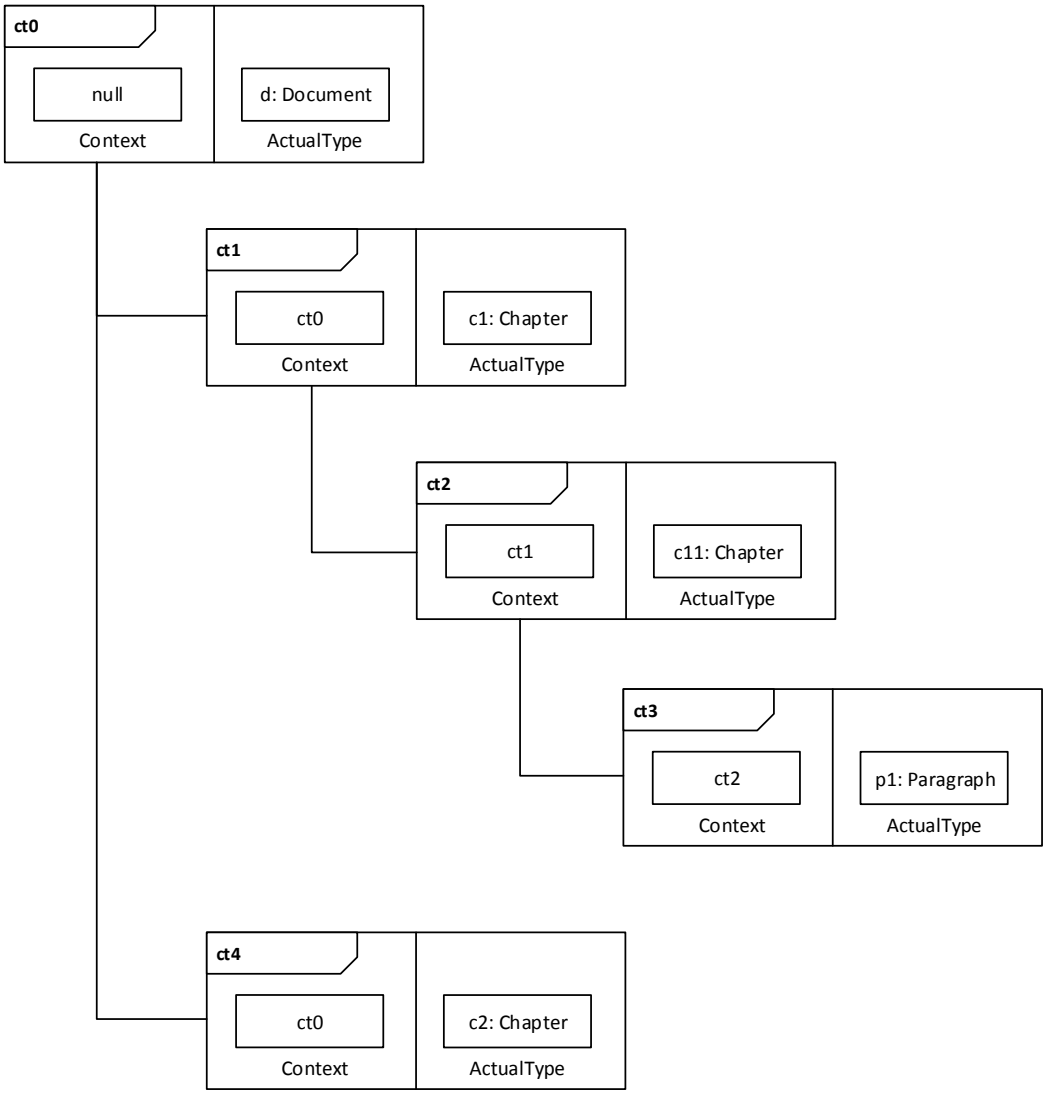


Figure 2.13: The correct ContextType representation

If we did not separate the first level, the aforementioned structure is still built correctly, **but** the ContextType representation is different. This is because we did not reset to the same point, the context reset jumped up to `Chapter 1` instead of `Document`. The context type representation of this document is shown below.

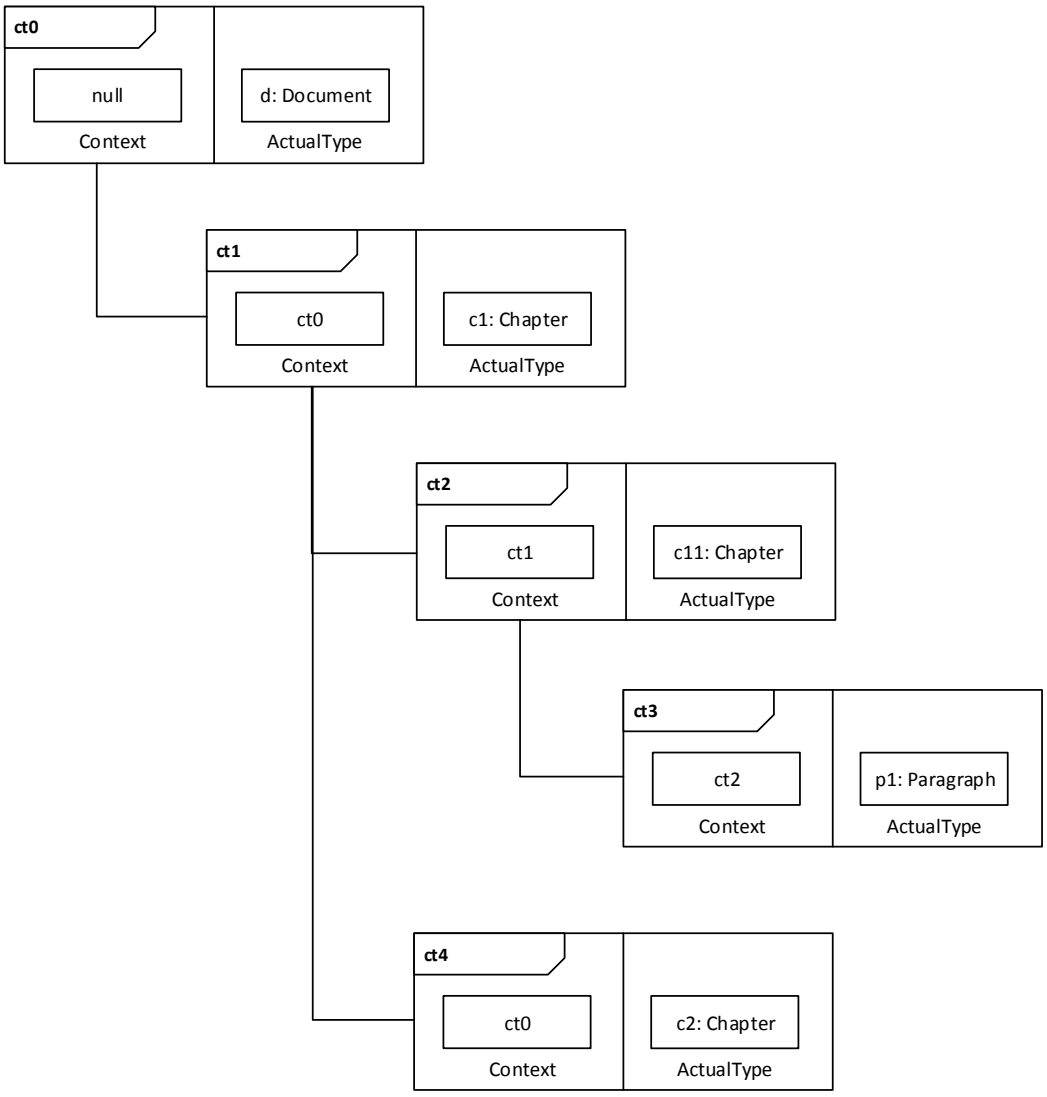


Figure 2.14: The wrong ContextType representation

2.7 Code blocks

We have seen that we can use code in code files and customise our document this way. But this does not allow us to do everything we want to. We want to be able to define variables and execute short expressions such as `x + y` We do not want to create a new document class

every time we want to do something out of the ordinary.

We want to be able to change properties of content in a Neio document without touching code files. It also has to be possible to insert content in such a document without using a symbol method. For example, we can create a class for pie charts but as there is no symbol method that creates it in the existing document classes, we can not insert it with the features discussed thus far.

For this reason, it is possible to add code blocks in a Neio document. The three different kinds of code blocks are explained below.

2.7.1 Non-scoped code

The first kind of code blocks is the non-scoped code block. We show how it is used with an example.

```
[Document]
# Chapter 1
{
    Chapter chapter2 = new Chapter("
        Chapter 2")
}
```

```
new Document()
    .#("Chapter 1");
Chapter chapter2 = new Chapter("Chapter
    2");
```

In the example above a code block is opened and a new `Chapter` is created. A non-scoped code block is defined by the following EBNF:

```
nl ::= "\r?\n"
non-scopedCodeBlock ::= "{" nl (statement ";" nl*)* nl* statement nl* ";"? nl*
    "}" "nl"
```

The statements in such a block are injected directly into the document, without introducing a scope. This means that we can use variables defined in a non-scoped block, later on in the document. However, this means that we should be careful when choosing names for our variables as they will be added to the global namespace of the Neio document.

We have now created a `Chapter` but it is not part of the document yet. We could add it manually by calling `addContent()`, but there is an easier way as this is a pattern that occurs a lot.

If the last statement in a non-scoped code block returns an object, this object is appended to the current document. We can thus also use a return statement as last statement to add an object to the document. These two possibilities are shown below.

Listing 2.16: Automatic return from code block

```
[Document]
# Chapter 1
{
    new Chapter("Chapter 2")
}
```

Listing 2.17: Manual return from code block

```
[Document]
# Chapter 1
{
    Chapter chapter2 = new Chapter("
        Chapter 2");
    return chapter2;
}
```

The object returned in the last statement is added to the document and it becomes part of the context. It is added to the document by calling `appendContent` on `this`. To further explain code blocks we explain `this` in the next section.

2.7.2 This

Like every object-oriented language, Neio has an expression to refer to the current object. Because the document is actually a chain of method calls, it is set to the last object returned before a code block. When an object is returned from a code block and added to the document, this object becomes the new `this`.

We need `this` as we do not know how to refer to the objects that were created in text mode. The example in Listing 2.16 can be seen as the following call chain.

```
new Document()
    .#("Chapter 1");
Chapter chapter 2 = new Chapter("Chapter 2")
this.appendContent(chapter 2);
```

A separate statement is created for the call chain before a code block. The object returned from the code block is placed in a variable and passed to the `appendContent` method call. When Listing 2.16 is actually translated, `this` is be filled in and it produces the following code:

```
Document $var0 = new Document();
Chapter $var1 = $var0.#("Chapter 1");
```

```
3 Chapter $var2 = new Chapter("Chapter 2");
4 $var1.appendContent($var2);
```

`this` is filled in with the last object that was returned in the call chain before the code block. To do this the compiler has knowledge of the `appendContent` method.

We show another example to show how context types and `this` work together. For this example we use the following document:

```
1 [Document]
2 # Chapter 1
3 {
4     Chapter chapter2 = #("Chapter 2")
5 }
```

```
1 new Document()
2     .#("Chapter 1");
3 Chapter chapter2 = this.#("Chapter 2");
```

In this example we create a chapter as we do in text mode, by using the `#` method. `Chapter` has no `#` method, but has `Document` does. The method already adds the created `Chapter` to the `Document`. Thus we assign a value to the result of the `#` method to prevent it from being automatically appended like it was in the previous examples.

Because of context types `this` is be filled in by the variable that holds `Document` instead of the last object that was returned in text mode. The final translation is given below.

```
1 Document $var0 = new Document();
2 Chapter $var1 = $var0.#("Chapter 1");
3 Chapter chapter2 = $var0.#("Chapter 2");
```

2.7.3 Scoped code

Sometimes all we want to do is execute some arbitrary code without corrupting the namespace. For this, we use a scoped code block. Its use is illustrated below.

```
1  [Document]
2  # Chapter 1
3  {{
4  List<String> l = new ArrayList<String>();
5  l.add("upquote");
6  l.add("pdfpages");
7  l.add("url");
8  for (int i = 0; i < l.size(); i = i + 1) {
9      addPackage(l.get(i));
10 }
11 }}
```

In this example a `Document` is created, but to suit our needs, it needs a few more packages that are not included by default in `Document`. We also do not want to think of very descriptive identifiers for our variables as it is just a simple operation, the name `l` should suffice.

A scoped block does the same as a non-scoped code block but it injects all of the code in a new scope. The last statement is not automatically appended to the document, to be certain that nothing leaves the scope. A scoped block can thus be used as a safe way to execute code without adding anything to the document, changing the current `this` or polluting the global namespace.

The call chain of this example is given below.

```
1  new Document()
2      .#("Chapter 1");
3  {
4      List<String> l = new ArrayList<String>();
5      l.add("upquote");
6      l.add("pdfpages");
7      l.add("pdfpages");
8      for (int i = 0; i < l.size(); i = i + 1) {
9          this.addPackage(l.get(i));
10     }
11 }
```

Scoped blocks have been used frequently while creating this document. One of its prominent uses has been to add images. This is because `TextContainer` defines an `image` method that already adds the image to the document. As we do not want to add the object twice, we use scoped code blocks to call this method. The image method and and example of how images are typically included is shown below.

2.7.4 Inline code

The the last type of code blocks are inline code blocks. As the name implies, this is code that is meant to be used inline. As such it can only contain one statement and the opening and closing bracket have to be on the same line.

Inline code blocks can be used anywhere that text can be used. In the example below we show how inline code blocks can be used in combination with a non-scoped code blocks to create a template for a letter.

Listing 2.18: template.no

```
[Document]
{
    Text addressee = "Thomas Vanhaskel
        ";
    Text help = "my math class";
    Text helpSubject = "math test";
    Text closings = "Kind regards,";
    Text name = "Titouan Vervack";
}

Dear {addressee},

Thank you for helping me out with {help
    }.

I was able to do great at the {
    helpSubject} thanks to you.

{closings}

{name}
```

Dear Thomas Vanhaskel,

Thank you for helping me out with my math class.

I was able to do great at the math test thanks to you.

Kind regards,

Titouan Vervack

Figure 2.15: The rendered form of the example to the left.

An inline code block returns a text and is appended to the rest of the document by calling the `text` method on `this`.

2.8 Text in code mode

In the previous sections we created objects such as `Chapter`s as follows `new Chapter("Chapter 1")`. However, `"Chapter 1"` is not a Java `String` it is a `Text`. `Text` can thus be written in code mode by enclosing text with a pair of doubles quotes (`"`).

A Java `String` is can still needed and can therefore be created using a pair of triple, single-quotes (`'''`). One of the most prominent reason for using `String` instead of `Text` is when you have to pass a path for example to pass code to a code listings.

The inspiration for using three quotes came from the multi-line String literal in Python. The reason we use three single quotes, instead of only, is because a pair of single quotes still represents a Java `Character`. We do not use two quotes because that that would be very confusing when used in combination with double quotes in a non-monospace font.

As we saw in Subsection 2.7.4, you can use code blocks in text mode. Since we can also use text mode in code mode, this means we can endlessly nest text and code mode.

An example demonstrating the strength of this nesting is shown below.

Listing 2.19: dice.no

```
[Document]

We roll the dice!

{
    Random rng = new Random();
    Integer limit = 12;
    Integer random = rng.nextInt() %
        limit;
    if(random > (limit / 2)) {
        "Ouch, we lost {random - (limit
            / 2)}!"
    } else {
        if (random == (limit / 2)) {
            "Oof, we broke even!"
        } else {
            "Great! We won {(limit / 2)
                - random}!"
        }
    }
}
```

We roll the dice!

Great! We won 8€!

Figure 2.16: The rendered form of the example to the left.

In this example we play a game where we roll two dice. In case we roll more than half of the total possible amount (12 in this case), we win.

In every case of the if statement, a piece of `Text` is found. As we saw in Subsection 2.4.2, this is translated to a `text` call. Since we placed and empty newline before the code block, this `text` is called on a `NLHandler` and a new paragraph containing the results of the game is created.

2.9 Navigating through the lexical structure

We have discussed all the features of Neio and we know how the object models are build. The lexical structure represented by the object model can now be navigated using a few methods in `Content`.

For example, we can use the `nearestAncestor` and `directDescendants` methods to count the number of top-level chapters in a document. An example of this is shown below as well as the code for the two methods.

```
[Document]
# Chapter 1
## Chapter 2
# Chapter 2

This document contains {nearestAncestor
    (Document.class).directDescendants(
    Chapter.class).size()} top-level
    chapters.
```

1 Chapter 1

1.1 Chapter 2

2 Chapter 2

This document contains 2 top-level chapters.

Figure 2.17: The rendered version of the example to the left.

Listing 2.20: Content.no

```
/**
 * Recursively checks for a parent of type {@code c}
 *
 * @param c Type of the demanded Content
 * @return The first lexical parent of type {@code c}
 */
<T> T nearestAncestor(Class<T> c) {
    Content p = parent();
    while ((p != null) && (!c.isInstance(p))) {
        p = p.parent();
    }

    return (T) p;
}
/**
 * Returns all direct descendants (the ones right beneath this) of
 * class {@code k}
 *
 * @param k The class of the descendants we are looking for
 * @return A list of direct descendants of class {@code k}
 */
<T extends Content> List<T> directDescendants(Class<T> k) {
    List<T> descendants = new ArrayList<T>();
    for (int i = 0; i < contentSize(); i = i + 1) {
        Content c = content(i);
        if (k.isInstance(c)) {
            descendants.add((T) c);
        }
    }

    return descendants;
}
```

2.10 Considerations

Whilst constructing Neio, a few other things were considered but not implemented. They are discussed below.

2.10.1 Static typing

Neio has been designed to use static typing. The reason for this is, that while we are writing a Neio document in Code mode, or we are writing a Neio class, we can be notified of possible type mismatches. This decreases the time spent debugging and spent waiting on the compiler as you weed out one type mismatch after another.

It also allows you to create a safer program, as type errors are caught at compile time. On the other hand when using dynamic typing, type errors only show up at runtime. Using auto-completion in an IDE, we also do not have that much more typing to do than when we would be using dynamic typing.

The types have been incorporated into the Neio compiler, but the necessary checks to create comprehensive error messages are not yet in place. In the current compiler, a `LookupException` is thrown whenever there is a type mismatch, undefined variable,... as the method or variable that we are looking for can not be found.

2.10.2 Security

Another issue that we have considered is security. This is an issue because it is possible to directly execute Java code from the Neio document. However, security is also an issue in LaTeX, as has been shown in several articles [3] [4].

Since Neio is more readable than LaTeX, malicious code can be found more easily than in LaTeX.

Adding a layer of security on top of Neio would require a lot of effort and research and is not in scope of this thesis. Thus we conclude by saying that we known there is a security risk,

but dismissing it is seen as future work.

Chapter 3

Supported document types and libraries

This chapter goes into some more details about what document types can be handled by Neio, and how to specifically build these kind of documents. We also discuss which libraries have been recreated in Neio to allow for a wide use of the language.

3.1 Document classes

It is important to note that to create complete document classes would require a lot more work than was available for this thesis. If we just have a look at the user guides of Memoir [18] or KOMA-Script [12], we see that they consist of 609 and 419 pages respectively. This is not something we could hope to reproduce in the scope of this thesis.

The simplest of documents can be created using the `Document` class. It provides syntax to create anything that Markdown can create, and on top of that, it allows you to create tables, LaTeX math, uml,... It also allows you to use the code blocks defined in Section 2.7, to customise your document. For example, it is possible to place two `Contents` next to each other by using the `leftOf` and `rightOf` methods that are defined in the `Content` class. The code of these methods is shown below. This code takes the parent from one of the two

Contents, the base, and links the other Content to the base. The implementation creates a root LaTeX Minipage [20] if these are the first two objects in a row to be placed next to each other, otherwise it just wraps the two Contents in a minipage. The root minipage takes up the total width, while the minipages that wrap the content take up 1/children of the width.

Listing 3.1: Content.no

```
 * Sets {@code c} right of this Content and sets the parent of
 * {@code c} if we have a parent or sets this parent if {@code c}
 * has a parent. One of the Contents should have a parent!
 *
 * @param c The Content to our right
 * @return The Content to our right
 */
<T extends Content> T leftOf(T c) {
    Minipage mp = prepareParent(this, c);
    Integer siblings = ((Minipage) (mp.content(0))).siblings();
    Minipage mp2 = new Minipage(siblings);
    mp2.addContent(c);
    mp.addContent(mp2);

    return c;
}

/**
 * Sets {@code c} right of this Content and sets the parent of
 * {@code c} if we have a parent or sets this parent if {@code c}
 * has a parent. One of the Contents should have a parent!
 *
 * @param c The Content to our left
 * @return The Content to our left
 */
<T extends Content> T rightOf(T c) {
    c.leftOf(this);
    return c;
}


/**
 * Prepares the parent of {@code left} or {@code right} to have
 * horizontally aligned children (by using Minipage).
 *
 * @param left  The left element whose parent has to be prepared
 * @param right The right element whose parent has to be prepared
 * @return The root Minipage
 */
private Minipage prepareParent(Content left, Content right) {
    Content child = null;
    if (left.parent() == null) {
        child = right;
    } else {
        child = left;
    }

    if (Minipage.class.isInstance(child.parent())) {
        List<Minipage> descendants = child.parent().directDescendants(Minipage
            .class);
        for (int i = 0; i < descendants.size(); i = i + 1) {
```

```
51              Minipage m = descendants.get(i);
52              m.incrementSiblings();
53          }
54          return (Minipage) (child.parent().parent());
55      }
56
57      Minipage rootPage = new Minipage(0);
58      child.replaceWith(rootPage);
59
60      Minipage mpl = new Minipage(1);
61      mpl.addContent(left);
62      rootPage.addContent(mpl);
63
64      return rootPage;
65  }
```

Another way of implementing this, that would probably have worked better, was through `TikZ` pictures. This would have been better because it allows you to easily set any two things next to each other or above/beneath each other. `Minipage`s require a lot more manual work, they might also split at a page-end which is not what we really want.

The `Document` document class can thus be used to write simple documents, such as small reports on various tasks.

3.1.1 Book

A prime subject of writing a book, is this document itself. The entirety of this book has been created in a single Neio document. The way we do this is by creating a new `Document` class that represents our LaTeX template. In this case the template is implemented in the `Thesis` class given below.

Listing 3.2: Thesis.no

```
1  namespace neio.thesis;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  import neio.stdlib.BlankPage;
7  import neio.stdlib.Chapter;
8  import neio.stdlib.Document;
9  import neio.stdlib.FiguresList;
10 import neio.stdlib.ListingsList;
11 import neio.stdlib.Packages;
12 import neio.stdlib.Pdf;
13 import neio.stdlib.Toc;
```

```
14
15 /**
16  * A document class to be used in for a Thesis
17  */
18 class Thesis extends Document;
19
20 /**
21  * Intialises the document class, adding the required packages and adding a
        ToC
22  */
23 Thesis() {
24     addPackage("graphicx")
25     .add("fancyhdr")
26     .add("amsmath")
27     .add("float")
28     .add("listings")
29     .add("tocloft")
30     .add("color")
31     .add("pdfpages")
32     .add("hyperref");
33
34     List<String> options = new ArrayList<String>();
35     options.add("english");
36     addPackage(options, "babel");
37
38     addClassMapping(Chapter.class, ThesisChapter.class);
39     addContent(new Pdf("TitlePage"));
40     addContent(new BlankPage());
41     addContent(new Pdf("TitlePage"));
42     addContent(new FrontMatter());
43 }
44
45 /**
46  * Sets up the main matter
47  */
48 void mainMatter() {
49     addContent(new Toc());
50     addContent(new FiguresList());
51     addContent(new ListingsList());
52     addContent(new MainMatter());
53 }
54
55 /**
56  * Builds the preamble for this document class
57  *
58  * @return The preamble for this document class
59  */
60 String preamble() {
61     StringBuilder preamble = new StringBuilder("\\documentclass[11pt,a4paper,
          oneside,notitlepage]{book}\n")
62     .append("\\setlength{\\topmargin}{2cm}\n")
63     .append("\\setlength{\\headheight}{14.49998pt}\n")
64     .append("\\setlength{\\headsep}{1cm}\n")
65     .append("\\setlength{\\oddsidemargin}{2.5cm}\n")
66     .append("\\setlength{\\evensidemargin}{2.5cm}\n")
67     .append("\\setlength{\\textwidth}{16cm}\n")
68     .append("\\setlength{\\textheight}{23.3cm}\n")
69     .append("\\setlength{\\footskip}{1.5cm}\n")
70     .append(packages().toTex())
71     .append("\\DeclareGraphicsRule{*}{mps}{*}{}")
72     .append("\\pagestyle{fancy}\n")
73     .append("\\renewcommand{\\chaptermark}[1]{\\markright{\\MakeUppercase
          {#1}}}\n")
```

```
74      .append("\\renewcommand{\\sectionmark}[1]{\\markright{\\thesection~#1}}\n
            ")
75      .append("\\newcommand{\\headerfmt}[1]{\\textsl{\\textsf{#1}}}\n")
76      .append("\\newcommand{\\headerfmtpage}[1]{\\textsf{#1}}\n")
77      .append("\\fancyhf{}\n")
78      .append("\\fancyhead[LE,RO]{\\headerfmtpage{\\thepage}}\n")
79      .append("\\fancyhead[LO]{\\headerfmt{\\rightmark}}\n")
80      .append("\\fancyhead[RE]{\\headerfmt{\\leftmark}}\n")
81      .append("\\renewcommand{\\headrulewidth}{0.5pt}\n")
82      .append("\\renewcommand{\\footrulewidth}{0pt}\n")
83      .append("\\fancypagestyle{plain}{\n")
84      .append("\\fancyhf{}\n")
85      .append("\\fancyhead[LE,RO]{\\headerfmtpage{\\thepage}}\n")
86      .append("\\fancyhead[LO]{\\headerfmt{\\rightmark}}\n")
87      .append("\\fancyhead[RE]{\\headerfmt{\\leftmark}}\n")
88      .append("\\renewcommand{\\headrulewidth}{0.5pt}\n")
89      .append("\\renewcommand{\\footrulewidth}{0pt}\n")
90      .append("}\n")
91      .append("\\renewcommand{\\baselinestretch}{1.5}\n")
92      .append("\\hyphenation{ditmagnooitgesplitstworden dit-woord-splitst-hier}\
            n")
93      .append("\\setlength{\\parindent}{0em}\n")
94      .append("\\setlength{\\parskip}{1em}\n")
95      .append("\\definecolor{comments}{RGB}{98, 151, 85}\n")
96      .append("\\lstset{\n")
97      .append("belowcaptionskip=1\\baselineskip,\n")
98      .append("breaklines=true,\n")
99      .append("frame=single,\n")
100     .append("comment=[l]{//},\n")
101     .append("morecomment=[s]{/**}{*/},\n")
102     .append("commentstyle=\\color{comments},\n")
103     .append("xleftmargin=\\parindent,\n")
104     .append("showstringspaces=false,\n")
105     .append("basicstyle=\\footnotesize\\ttfamily,\n")
106     .append("keywordstyle=\\bfseries,\n")
107     .append("stringstyle=\\ttfamily,\n")
108     .append("numbers=left,\n")
109     .append("numbersep=7pt,\n")
110     .append("numberstyle=\\tiny,\n")
111     .append("rulecolor=\\color{black},\n")
112     .append("lineskip={-1.5pt}\n")
113     .append("}\n");
114
115     return preamble.toString();
116 }
```

It also says that any `Chapter` that is automatically created, typically by the `#` method in `Document` or `Chapter`, should be an instance of `ThesisChapter`. This is done through the following line.

```
1 addClassMapping(Chapter.class, ThesisChapter.class);
```

A `ThesisChapter` just redefines a chapter so that it translates into the LaTeX `chapter` macro when it is level 1 instead of directly calling the LaTeX section macro.

Listing 3.3: ThesisChapter.no

```
namespace neio.thesis;

import neio.lang.Text;
import neio.stdlib.Chapter;

class ThesisChapter extends Chapter;

ThesisChapter(Text title, Integer level) {
    super(title, level);
}

String subs(Integer lvl) {
    String subs = "";
    for (int i = 2; i < lvl; i = i + 1) {
        subs = subs + "sub";
    }

    if (lvl < 2) {
        texName = "chapter";
    } else {
        texName = "section";
    }

    return subs;
}
```

It extends the `Document` class and adds its own packages and creates its own LaTeX preamble. It also provides a method that initialises the main matter, which contains a Table of Contents (ToC) using the following line. The code for a ToC is shown below.

Listing 3.4: Toc.no

```
namespace neio.stdlib;

import neio.lang.Content;

class Toc extends Content;

String toTex() {
    return "\\newpage\n\\pagestyle{fancy}\n\\tableofcontents\n";
}
```

We can also add things to a document class ourselves which might allow us to reuse other document classes For example, to add an abstract to the `Thesis` document class, we just have to create an `Abstract` class that extends `Content` and add it to the document at the right time. This allows us to add more functionality to a document class, without having to modify the document class itself, or without having to create a new one, for example `Thesis`

and `ThesisWithAbstract`.

To add an abstract , we could initialise an `Abstract`, this `Abstract` might then overwrite the `newline` method to create an `AbstractNLHandler`. The `AbsttractNLHandler` can then override the `text` method to initialize itself, as an abstract is just a block of text. Overriding the `newline` method in the `AbstractNLHandler` allows us to escape this "Abstract mode" by typing an empty line. The code to for `Abstract` and `AbstractNLHandler` are shown below, an example of what the output looks like can be found at the beginning of this document.

3.1.2 Other document classes

To be able to create an article, a letter, or anything else, the only thing we would have to do is create a new Neio class for this document class. This Neio class then implements its own LaTeX preamble and decide what packages to include by default and it might have to redefine what types of objects should be used using `addClassMapping`. It might also add methods to be filled in, using a code block, or by overwriting methods and making use of the newline handlers.

For example, you might create a `LetterHeader` class that represents the header of a formal letter. This header requires some information about the writer, the name, the addressee,.... One way to fill these in is just by passing them to the constructor of `LetterHeader` or by using setters defined in `LetterHeader`. Another way to do this, is by overwriting some methods in `LetterHeader`. We could overwrite the `#` method to represent the name of the author, and `-` to represent the addressee for example. Another possibility is to used nested methods for this, `#` represents the author, `##` represents the addressee and so on.

Yet another way would be to create objects for the parameters we need (grouping some together if that makes sense) such as `Author` and `Addressee`. We then implement a method, let us take `#` once more, in `LetterHeader` that creates a new `Author` and returns it. We then override a method, the same or a different one, in `Author` that creates the `Addressee` and returns it. The advantage of this last step is that the structure of the letter is well split up and is enforced on the user. Forgetting to define the `Author` should be immediately noticeable

as the method to create the `Addressee` should not be known yet without creating the `Author` first. On the other hand this is also more prune to error as people tend to forget things all the time.

3.1.3 Slides

3.2 Libraries

To show what the language is capable of a few libraries were created or reimplemented. All of the currently available libraries in Neio are explained below. Note that none of these libraries are complete and would require a lot more effort to reach a state of completion. Just having a look at the manuals of a few popular libraries such as TikZ[25] we can see that more time is needed to write such a library than there is available for this thesis. The libraries that were created, are meant for illustrative purpose, to show what Neio is capable of.

3.2.1 BibTeX

The library created for BibTeX is one of the easiest ones created for Neio. The implementation is just a binding to LaTeX and does not really allow for a lot of customisation as we do not have an object model of the BibTeX file. We chose to implement BibTeX this way to illustrate that binding to LaTeX is not very difficult, and because we would have had to create an entire parser for BibTeX to be able to create the object structure needed to have full control over the BibTeX files. As the BibTeX format is quite complicated we felt that our time was better spent elsewhere. This is mainly because the only thing we require of BibTeX for this thesis, is to allow us to easily quote articles, websites and so on. The code for our implementation of BibTeX can be found below.

Listing 3.5: Bibtex.no

```
1 namespace neio.stdlib;
2
3 import neio.lang.Content;
4
5 /**
```

```
 * Implements a binding to the LaTeX's BibTeX
 */
class Bibtex extends Content;

String name;

/**
 * Initializes the BibTeX
 *
 * @param name The path to the BibTeX file
 */
Bibtex(String name) {
    this.name = name;
}

/**
 * Cites something defined in this BibTeX
 *
 * @param cname The keyword of the citation
 * @return A Citation object representing the citation of {@code cname}
 */
Citation cite(String cname) {
    return new Citation(cname);
}

/**
 * Creates a TeX representation of this BibTeX, without citations
 *
 * @return The TeX representation of this
 */
String toTex() {
    StringBuilder result = new StringBuilder("\n\\bibliographystyle{plain}\n")
        ;
    result.append("\\bibliography{").append(name).append("}\n");

    return result.toString();
}
```

A BibTeX can now be added through the `addBibtex` method in `Document` and we can cite an entry by using the proxy method, `cite`, in `Document`.

Listing 3.6: Document.no

```
/**
 * Sets a BibTeX file for this Document
 *
 * @param The name of the BibTeX file
 */
void addBibtex(String name) {
    this.bibtex = new Bibtex(name);
}

/**
 * A proxy method for {@code BibTeX}'s {@code cite}
 * Returns a Cite that is linked to {@code key}
 *
 * @param key The key to access the citation
 * @return The Cite corresponding to {@code key}
 */
```

```
Citation cite(String key) {
    if (bibtex != null) {
        return bibtex.cite(key);
    } else {
        return null;
    }
}
```

Lastly we show the code for the `Citation` class.

Listing 3.7: Citation.no

```
namespace neio.stdlib;

import neio.lang.Text;

/**
 * Represents something that has been cited.
 * It is a subclass of Text to allow for it to be used in inline code blocks.
 */
class Citation extends Text;

String name;

/**
 * Initialises a Citation
 *
 * @param name The name of the Citation
 */
Citation(String name) {
    this.name = name;
}

/**
 * Creates a TeX representation of this
 *
 * @return The TeX representation of this
 */
String thisToTex() {
    return "\\cite{" + name + "}";
}
```

The `Citation` class is a subclass of `Text` as we call it through an inline code block. All the cites in this thesis have been provided through these classes and are called as below. `key` is a Java String as it is used to do a lookup on the BibTeX entries, it should not be allowed to be marked up.

```
[Document]
This is how we cite something {cite('''key''')}.
```

3.2.2 References

The way references have been implemented is quite straightforward. Any class that implements the `Referable` interface, such as `Chapter` or `Image`, can be referenced by calling the `ref` method. The way we typically go about a reference is as follows.

```
[Document]
# Chapter 1
{Chapter chap = (Chapter) parent()}

In {chap.ref()} we explain references.
```

What we use here is a little trick to be able to get the `Chapter` into a variable. What happens is as follows, a `Chapter` is created using the `#` method and then the `newline` method is called upon it (this method has been inherited from `TextContainer`). A `NewlineHandler` always takes the object that created it as a parameter in its constructor so that it can refer to it later on. The `parent` call in the inline code block is thus called on the NewlineHandler, more specifically the `ContainerNLHandler`. As the parent of a `ContainerNLHandler` is a `TextContainer`, a cast to `Chapter` is needed. The code block is allowed to be an inline code block instead of a non-scoped code block because the statement in it is an assignment and thus returns nothing. To reference an object LaTeX needs a unique label, we use the hashcode of an object for this as it is easy to retrieve at any time and it is unique. The newline method from `TextContainer` as well as the `ref` method in `Chapter` and the `Referable` interface are shown below.

Listing 3.8: TextContainer.no

```
/**
 * Handles newlines
 *
 * @return Returns a new NLHandler
 */
NLHandler newline() {
    return new NLHandler(this);
}
```

Listing 3.9: Chapter.no

```
 * Returns a string containing the right amount of "sub"'s to create a valid
 * LaTeX section.
 *
```

```
 * @param lvl The level of nesting of this Chapter
 * @return      The string containing all the needed "sub"'s
 */
String subs(Integer lvl) {
    String subs = "";
    for (int i = 1; i < lvl; i = i + 1) {
        subs = subs + "sub";
    }

    return subs;
}

/**
 * Creates a reference to this Chapter and adds a correct prefix to it
 * such as Chapter, Section or Subsection.
```

Listing 3.10: Referable.no

```
namespace neio.lang;

/**
 * Declares that a class can be referred to
 */
interface Referable;

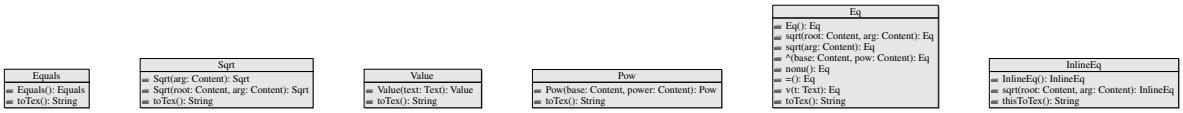
/**
 * References an object
 *
 * @return A reference to this object
 */
Reference ref();
```

As we need a variable to reference something, this means that the object we want to refer to has to be available when we want to refer to it. This means that we can not refer to things that are defined later on in the project, which LaTeX can do. LaTeX is able to do this because it compiles more than once and is able to gather the information needed to create a reference in one of the earlier runs. For Neio to be able to do this, we would have to do something similar as we can not simply let the compiler create the object we want to reference in advance. This is not possible because the creation of an object is dependent on its context.

3.2.3 LaTeX math and amsmath

As mentioned in Subsection 1.1.2, LaTeX is very good at typesetting mathematical formulas. As such it is something that we could not forget about. Two ways to make use of the LaTeX math and the amsmath package have been created. The UML below shows the currently

implemented set of objects in the Neio math library.



To create an inline math formula, in LaTeX this is done by surrounding some text with a pair of $'s, we create an instance of the `InlineEq` class. The only method that has been implemented for it thus far is `sqrt` so we show how we can create an inline square root.

```
1 {
2     Integer base = 2;
3     Integer arg = 10;
4 }
5 ${ieq().sqrt("{base}", "{arg}")}$
```

As you can see we also make use of the `$` operator, in our case it is a surround method defined in `Text`. What we do next is a bit tricky, inside the surround method, we create an inline code block, and in it we create an `InlineEq` on which we call the `sqrt` method. The reason we have to open this code block is because the contents of a surround method has to be a text and in normal text we can not call methods. The arguments that are passed to the `sqrt` method are also special, each of the arguments is an inline code block wrapped in a Neio text. Once again, this has been done because we need to be able to pass `Text` to the `sqrt` method, not just an integer. A solution for this is proposed in Section 5.3.

Continuing the example, it is also possible to explicitly create an equation, in LaTeX this is done using the `equation` environment.

```
1 {{
2     eq().nonu().sqrt("{base}", "{arg}")
          ;
3     eq().^("{base}", "{arg}").=().v("{
          Math.pow(base, arg)}");
4 }}
```

$$\sqrt[2]{10}$$
$$2^{10} = 1024.0 \tag{1}$$

Figure 3.1: The rendered form of the example to the left

The code above creates two `Equation`, on the first one it disable numbering through the `nonu` method and calls sqrt on the equation, as shown before. In the second equation we use the `^`

method to create an exponentiation and use the `=` method to create an equation from it. The `v` method creates a `Value` and calculates the actual value of what has been typeset before it. To be clear, everything except for the `Math.pow()` typesets something, but computes nothing. This can be clearly seen in Figure 3.1. As most of the code in this library is straightforward, does nothing new and is a direct translation to LaTeX, we only show the code for `Eq` class.

Listing 3.11: Eq.no

```
namespace neio.stdlib.math;

import neio.lang.Content;
import neio.lang.Text;

/**
 * Creates an explicit equation
 */
class Eq extends Content;

String texname;

/**
 * Initialises the equation
 */
Eq() {
    texname = "equation";
}

/**
 * Creates a square root and adds it to this equation, returning this to allow
     further chaining
 *
 * @param root The base of the square root
 * @param arg  The value to take the square root of
 * @return This equation
 */
Eq sqrt(Content root, Content arg) {
    addContent(new Sqrt(root, arg));
    return this;
}

/**
 * Creates a square root with no explicit base
 *
 * @param arg The value to take the square root of
 * @return This equation
 */
Eq sqrt(Content arg) {
    return sqrt(null, arg);
}

/**
 * Creates an exponentiation and appends it to this equation
 *
 * @param base The base of the exponentiation
 * @param pow The exponent of the exponentiation
 * @return This equation
 */
```

```
49 Eq ^(Content base, Content pow) {
50     addContent(new Pow(base, pow));
51     return this;
52 }
53
54 /**
55  * Disables numbers for this equation
56  *
57  * @return This equation
58  */
59 Eq nonu() {
60     texname = texname + "*";
61     return this;
62 }
63
64 /**
65  * Appends an equals sign to this equation
66  *
67  * @return This equation
68  */
69 Eq =() {
70     addContent(new Equals());
71     return this;
72 }
73
74 /**
75  * Appends a value to this equation
76  *
77  * @return This equation
78  */
79 Eq v(Text t) {
80     addContent(new Value(t));
81     return this;
82 }
83
84 /**
85  * Creates the TeX representation of this equation
86  *
87  * @return The TeX representation of this equation
88  */
89 String toTex() {
90     return "\\begin{" + texname + "}" + super.toTex() + "\n\\end{" + texname +
          "}";
91 }
```

3.2.4 LaTeX tables

Tables are one of the main things missing in Markdown and is one of the things people have the most problems with in LaTeX. As such we tried to make a table that is simple to create, that is readable in source code and that allows you to easily edit values in the table later on, making it so we do not have to inline all of the changes, making it hard to read how the table. The table is created like you would create an ASCII table, with pipes, spaces, dashes and values for the elements of the table. An example is shown below.

```
[Document]
In the table below you can see the results of the 33rd 12urenloop of the
    University of Ghent:

|      Student club      | Rounds | Seconds/Round | Dist (km) | Speed km/h |
-------------------------------------------------------------------------
| HILOK                  |  1030  |       42      |   298,70  |    24,89   |
| VTK                    |  1028  |       42      |   298.12  |    24.84   |
| VLK                    |   841  |       51      |   243.89  |    20.32   |
| Wetenschappen and VLAK |   819  |       53      |   237,51  |    19.79   |
| VGK                    |   810  |       53      |   234.90  |    19.58   |
| Hermes and LILA        |   793  |       54      |   229.97  |    19.16   |
| HK                     |   771  |       56      |   223.59  |    18.63   |
| VRG                    |   764  |       57      |   221.56  |    18.46   |
| VEK                    |   757  |       57      |   219.53  |    18.29   |
| VPPK                   |   689  |       63      |   199.81  |    16.65   |
| SK                     |   647  |       67      |   187.63  |    15.64   |
| Zeus WPI               |   567  |       76      |   164.43  |    13.70   |
| VBK                    |   344  |      126      |    99.76  |     8.31   |
```

In the table below you can see the results of the 33rd 12urenloop of the University of Ghent:

Student club	**Rounds**	**Seconds/Round**	**Dist (km)**	**Speed km/h**
HILOK	1030	42	298,70	24,89
VTK	1028	42	298.12	24.84
VLK	841	51	243.89	20.32
Wetenschappen and VLAK	819	53	237,51	19.79
VGK	810	53	234.90	19.58
Hermes and LILA	793	54	229.97	19.16
HK	771	56	223.59	18.63
VRG	764	57	221.56	18.46
VEK	757	57	219.53	18.29
VPPK	689	63	199.81	16.65
SK	647	67	187.63	15.64
Zeus WPI	567	76	164.43	13.70
VBK	344	126	99.76	8.31

Figure 3.2: The rendered form of the table given above

First of all, we note that it does not matter how many spaces are placed in side the elements of the table. The way this table is build is by mixing the use of normal method calls of `|`, a nested call of `-` and by making use of a NewlineHandler. The table is created by the `|` method in `TextContainer`, so that it van be used in `Chapters` as well as `Documents` (they are both subclasses of `TextContainer`).

Listing 3.12: TextContainer.no

```
/**
 * Creates a new Href
 *
 * @param text The text to show instead of the url
 * @param url  The url we are referring to
 * @return the new Href
 */
Href href(Text text, String url) {
    return new Href(text, url);
}

/**
 * Creates a new Table and adds a new TableRow to it.
 * The Table is added to this.
```

The `TableRow` that is returned form the the `|` method is `TableRow` that is stil under construction. We continue building it by repeatedly calling the `|` method on it, this time this method is located in `TableRow`. The code for the `TableRow` and `TableElement` are shown below.

Listing 3.13: TableRow.no

```
namespace neio.stdlib;

import java.util.List;
import java.util.ArrayList;

import neio.lang.Content;
import neio.lang.Text;

class TableRow extends Content;

TableElement fte = null;
List<TableElement> elements = new ArrayList<TableElement>();
boolean bold = false;

TableRow() {
}

TableRow(Text text) {
    elements.add(new TableElement(text));
}

void setBold() {
    bold = true;
}

TableRow |(Text text) {
    elements.add(new TableElement(text));
    return this;
}

TableRow |() {
    fte = new FinalTableElement();
    return this;
}

Table parent() {
```

```
37      return (Table) (super.parent());
38  }
39
40  TableRow appendRow(TableRow row) {
41      return parent().appendRow(row);
42  }
43
44  Hline appendHline() {
45      return parent().appendHline();
46  }
47
48  TableNLHandler newline() {
49      return new TableNLHandler(this);
50  }
51
52  String header() {
53      StringBuilder header = new StringBuilder();
54      for (int i = 0; i < elements.size(); i = i + 1) {
55          TableElement te = elements.get(i);
56          header.append(" ").append(te.header()).append(" | ");
57      }
58
59      return header.toString();
60  }
61
62  String toTex() {
63      if (elements.isEmpty()) {
64          return "";
65      }
66
67      if (bold) {
68          elements.get(0).setBold();
69      }
70      StringBuilder result = new StringBuilder(elements.get(0).toTex());
71      for (int i = 1; i < elements.size(); i = i + 1) {
72          TableElement element = elements.get(i);
73          if (bold) {
74              element.setBold();
75          }
76          result.append(" & ").append(element.toTex());
77      }
78
79      if (fte != null) {
80          result.append(fte.toTex());
81      }
82
83      return result.toString();
84  }
```

Listing 3.14: TableElement.no

```
1  namespace neio.stdlib;
2
3  import java.util.List;
4  import java.util.ArrayList;
5
6  import neio.lang.BoldText;
7  import neio.lang.Content;
8  import neio.lang.Text;
9
10 /**
```

```
11  * Represents an element in a Table
12  */
13 class TableElement extends Content;
14
15 protected String header;
16 Text text;
17
18 /*
19  * Initialises the TableElement and alligns it to the left
20  */
21 TableElement() {
22     header = "l";
23 }
24
25 /*
26  * Initialises the TableElement with a given value
27  *
28  * @param text The text to be used as value of this element
29  */
30 TableElement(Text text) {
31     this();
32     this.text = text;
33 }
34
35 /**
36  * Enables the bold property, if this is set the element will be
37  * printed in bold
38  */
39 void setBold() {
40     text = new BoldText(text);
41 }
42
43 /**
44  * Returns this element should be aligned
45  *
46  * @return A string representing how this element should be aligned
47  */
48 String header() {
49     return header;
50 }
51
52 /**
53  * Build the TeX representation of this
54  *
55  * @return The TeX representation of this
56  */
57 String toTex() {
58     return text.toTex();
59 }
```

As you can see in the `TableRow` class, in the DSL we created, a `|` method without any arguments, denotes the end of a `TableRow`. To continue building the table, we use a `TableNLHandler` that has been created in the `newline` method of the `TableRow`. We use the nested '-' method in this handler to be able to insert newlines. In our DSL the length of the nested method is ignored, but it could be used to measure the width of a table for example. The handler also offers the `|` method which appends a new `TableRow` to the `Table`.

Finally we can write an empty newline to get out of the DSL for creating a `Table` because of the `newline` method in the handler.

The following libraries that were implemented are more domain specific and less generally usable. However this shows that we can use Neio to typeset a wide variety of different documents.

3.2.5 Red black trees

Red black trees are trees that are constantly update according to a defined algorithm. When we want to show a red black tree in LaTeX, we can typeset an instance of such a tree using `TikZ`. Or we could create it in a third party program and include an image of an instance of such tree in our document, this is a solution that would also work Markdown.

However, since an algorithm is used, and we have a programming model, could not we just build the tree and then display it? It turns out that we can. All that we have to do is take some code that can generate red black trees and add a display method to it. We show an example of the red black trees in use.

Listing 3.15: rbtDocument.no

```
[Document]

{
    List<Integer> tree = new ArrayList<Integer>();
    tree.add(33);
    tree.add(15);
    tree.add(10);
    tree.add(5);
    tree.add(20);
    tree.add(18);
    tree.add(47);
    tree.add(38);
    tree.add(36);
    tree.add(39);

    String numbers = ''''''' + tree.get(0);
    for (int i = 1; i < tree.size(); i = i + 1) {
        numbers = numbers + ''', ''' + tree.get(i);
    }
    Integer a = 5;
}

Given the red black tree of {numbers}
{
```

```
25      RedBlackTree rbt = new RedBlackTree().insert(tree);
26      return rbt;
27  }
28
29  Add 49
30  {
31      RedBlackTree rbt2 = rbt.insert(49);
32      return rbt2;
33  }
34
35  Add 51
36  {
37      RedBlackTree rbt3 = rbt2.insert(51);
38      return rbt3;
39  }
```

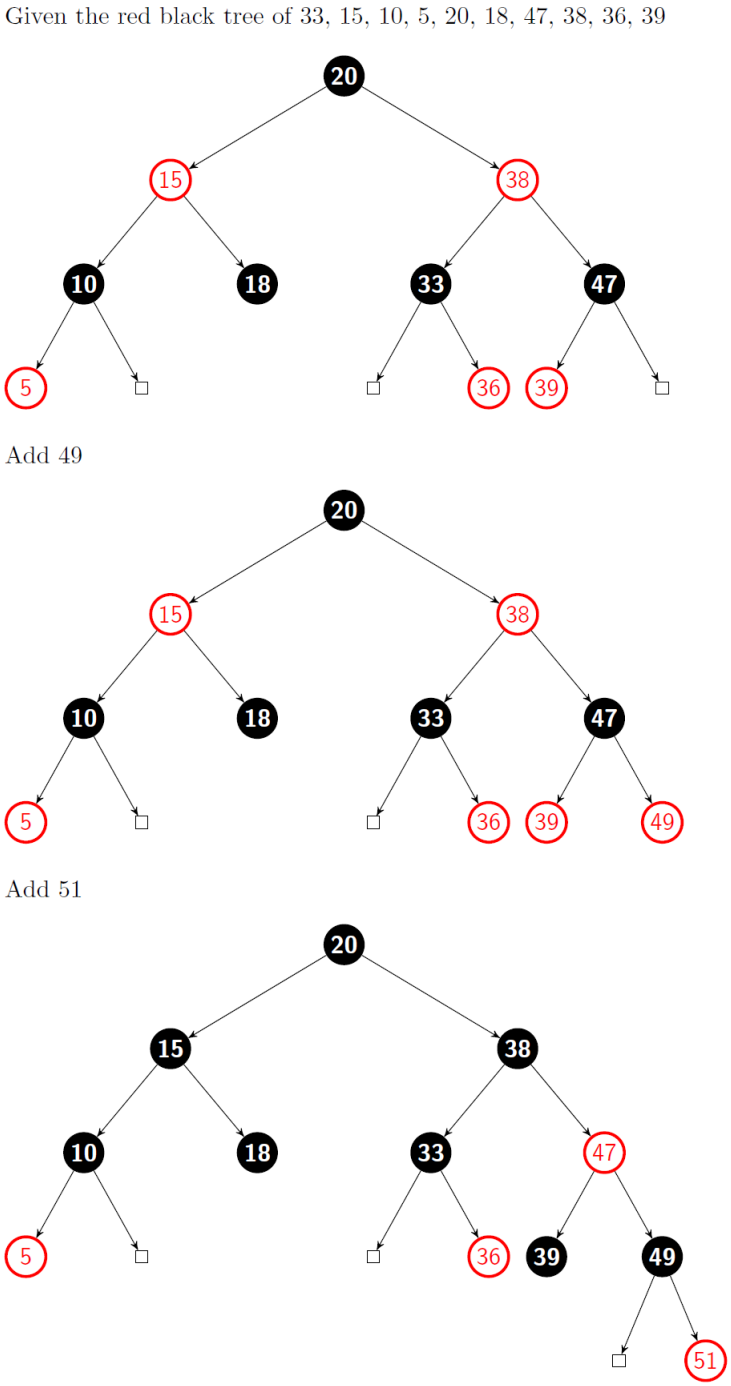


Figure 3.3: The rendered form of the above example

The rendered document is shown in Figure 3.3. As we can see it is very easy to create and show a red black tree. We could do this for any tree or graph as long as we have code for it

and write a method that can display it. As showing the algorithm to create red black trees is not the purpose of this example, we only show the code to display it and to append something to it.

Listing 3.16: RedBlackTree.no

```
/**
 * Clones the current red black tree and inserts a new element into items
 *
 * @param value The value to insert
 * @return The newly created red black tree
 */
RedBlackTree insert(Integer value) {
    if (last == null) {
        last = new RedBlackNode(RedBlackNode.BLACK, value);
        return this;
    }
    RedBlackTree newTree = new RedBlackTree(root());
    newTree.insert(value, newTree.root());

    return newTree;
}
/**
 * Makes sure the TikZ library is used and that everything is prepared to
    typeset a red black tree in LaTeX
 */
void preTex() {
    Document d = nearestAncestor(Document.class);
    d.addPackage("tikz");

    StringBuilder header = new StringBuilder("\\usetikzlibrary{arrows}")
    .append("\\tikzset{treenode/.style = {align=center, inner sep=0pt, text
        centered, font=\\sffamily},\n")
    .append("arn_n/.style = {treenode, circle, white, font=\\sffamily\\
        bfseries, draw=black,\n")
    .append("fill=black, text width=1.5em},% arbre rouge noir, noeud noir\n")
    .append("arn_r/.style = {treenode, circle, red, draw=red,\n")
    .append("text width=1.5em, very thick},% arbre rouge noir, noeud rouge\n")
    .append("arn_x/.style = {treenode, rectangle, draw=black,\n")
    .append("minimum width=0.5em, minimum height=0.5em}% arbre rouge noir, nil
        \n")
    .append("}\n");

    d.addToPreamble(header.toString());

    super.preTex();
}

/**
 * Creates the TeX representation this red black tree
 *
 * @return The TeX representation of this tree
 */
String toTex() {
    if (last == null) {
        return "";
    }

    StringBuilder result = new StringBuilder("\n\\begin{tikzpicture}[->,>=
```

```
        stealth',level/.style={sibling distance = 5cm/#1,");
    result.append("level distance = 1.5cm}]\n");
    result.append(root().toTex());
    result.append("\\end{tikzpicture}");

    return result.toString();
}
```

Listing 3.17: RedBlackNode.no

```
/**
 * Recursively creates the Text representation of the tree.
 *
 * @return The TeX representation of the tree
 */
String toTex() {
    StringBuilder result = new StringBuilder();
    if (isRoot()) {
        result.append("\\node [arn_");
    } else {
        result.append("child { node [arn_");
    }

    if (isBlack()) {
        result.append("n]");
    } else {
        result.append("r]");
    }

    result.append(" {").append(value.toString()).append("}\n");
    // If we don't have two empty leaves
    if (!(!hasLeft() && !hasRight())) {
        if (!hasLeft()){
            result.append("child { node [arn_x] {}}\n");
        } else {
            result.append(left.toTex());
        }
        if (!hasRight()) {
            result.append("child { node [arn_x] {}}\n");
        } else {
            result.append(right.toTex());
        }
    }

    if (!isRoot()) {
        result.append("\n}\n");
    } else {
        result.append(";\n");
    }

    // Remove empty newline, it breaks the tikzpicture
    return result.toString().replaceAll("(\n)\\1+","$1");
}
```

We also show the LaTeX code that is generated for the first of the three trees. This code is not that easy to read, hard to write and thus prune to syntax errors. This code is also newline sensitive, placing empty newlines between the entries makes it so that your document

no longer compiles. The code for the next tree is very similar, but it is hard to reuse this code.

Listing 3.18: rbtDocument.tex

```
\begin{tikzpicture}[->,>=stealth',level/.style={sibling distance = 5cm/#1,
    level distance = 1.5cm}]
\node [arn_n] {20}
child { node [arn_r] {15}
child { node [arn_n] {10}
child { node [arn_r] {5}
}
child { node [arn_x] {}}
}
child { node [arn_n] {18}
}
}
child { node [arn_r] {38}
child { node [arn_n] {33}
child { node [arn_x] {}}
child { node [arn_r] {36}
}
}
child { node [arn_n] {47}
child { node [arn_r] {39}
}
child { node [arn_x] {}}
}
}
;
\end{tikzpicture}
```

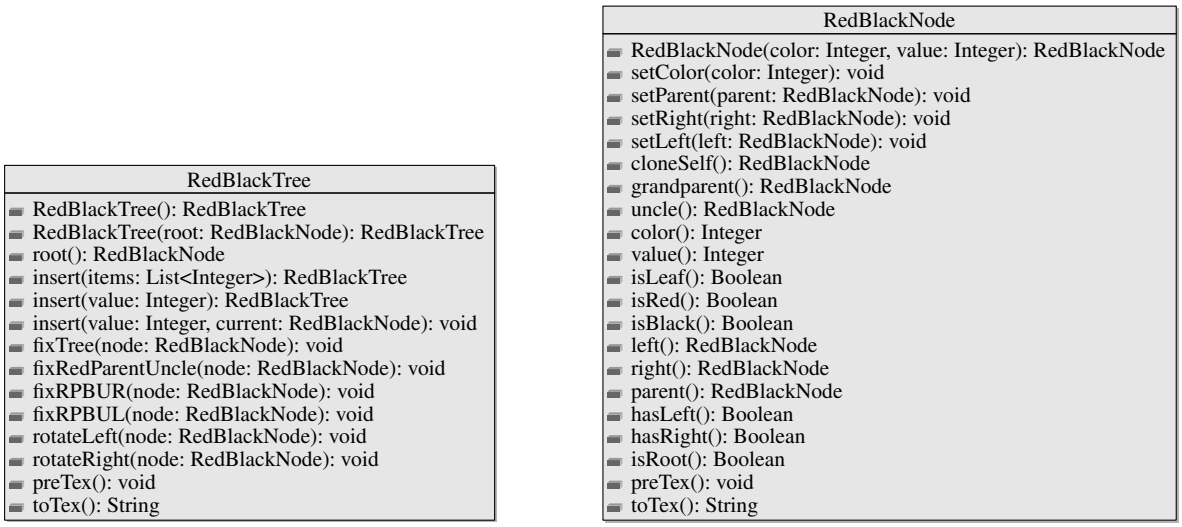
Note that this tree is not optimally implemented, in our case when we append something to the tree, we completely clone it and then add a new node to this new instance. The red black tree in this example also only works for integers. The reason we clone the tree every time we insert a new value is because otherwise all three of the trees would all be shown exactly the same and they would all show the last version of the three.

3.2.6 MetaUML

The next example has already been used a few times in this document. Every UML diagram up to now was created using a UML library implemented in Neio. It is also a little different from the other libraries we have seen thus far as it does not directly translate into LaTeX. Instead we are making use of MetaUML [19]. A high-level overview of how we create UML diagrams is given here, but a lower level overview is given in Chapter 4. We start out with

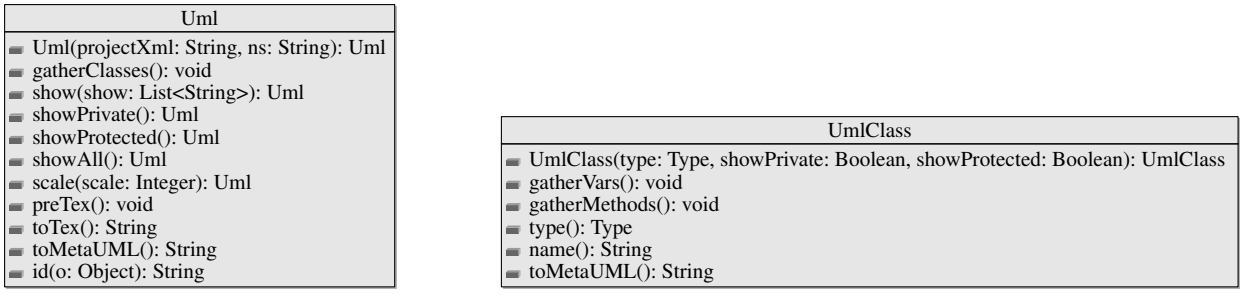
an example.

```
[Document]
{
    new Uml('''./project.xml''', '''neio.stdlib.graph''')
}
```

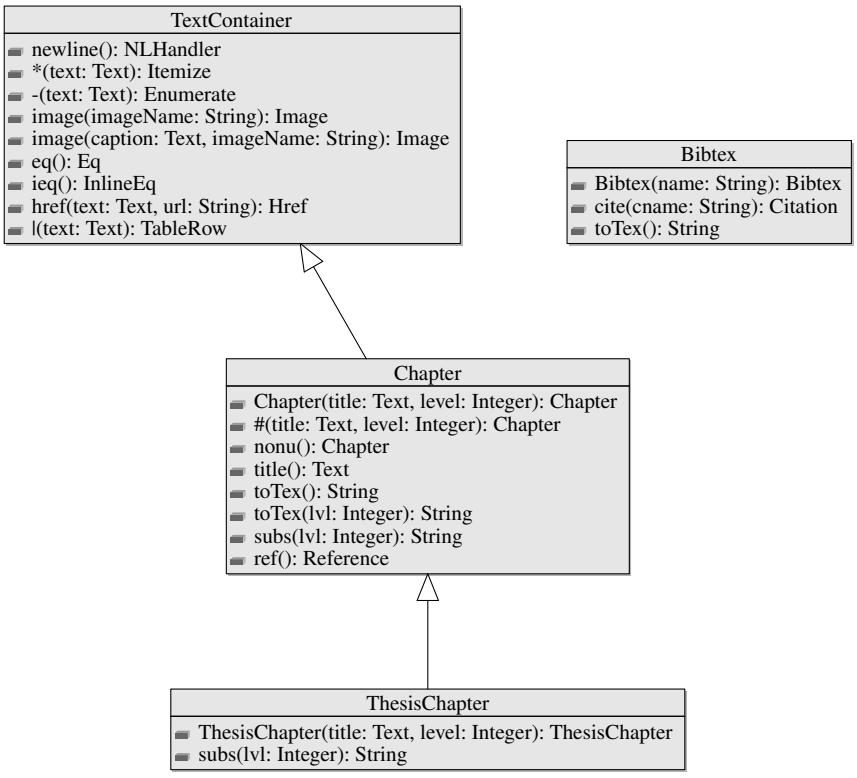


This code reads the `project.xml` file, that tells it what project we want to create UML for. The second parameter is the namespace the uml should be created for. It is also possible to only show a few classes, or to also show private and protected members. This is shown in the example beneath

```
[Document]
{
    List<String> list = new ArrayList<String>();
    list.add('''neio.stdlib.uml.Uml''');
    list.add('''neio.stdlib.uml.UmlClass''');
    new Uml('''./project.xml''', '''neio.stdlib.uml''').scale(100).show(list).
        showPrivate();
}
```



As the code for this library is over 600 lines long, we do not show all of the code. To get an image out of this, we output our object structure to the MetaUML format, instead of LaTeX. And then run it through the `mpost` command line program which generates TeX code for the UML, which we can then include. The problem with the MetaUML tool is that positioning is not done automatically. The way it was implemented in our case is a bit like a tree. We build a tree of levels, the first level contains all the classes that have no super class to be shown as well as the highest level classes. Subclasses are shown on the level beneath their super class. For example, `ThesisChapter` is a subclass of `Chapter`, which is a subclass of `TextContainer`. If we now show these three classes, they are displayed one under the other because we built a tree with three levels. To visualize how the levels are being used, we show this example below but also add the Bibtex class to the UML , as it has no connection to the other classes.



Note that our implementation of UML diagrams only shows what it is asked to show (hence there are no uses arrows) and only draws inheritance arrows for what is shown. This is done to avoid clutter.

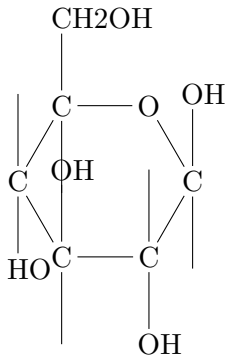
3.2.7 Chemfig

In this example we approach a different domain, the domain of Biology and Chemistry. For it, we created a library and DSL to create structural formulas. An example of the structural formula for glucose is given below.

```
[Document]
{
    new Structure()
}

$ C * 6 _ OH ^ - C _ ^ OH - O - C _ ^ CH2OH - C _ HO ^ - C _ ^ OH -
| Glucose
```

This example produces the following output



Glucose

The first thing we see, is that we create a new `Structure`. This provides a newline handler that allows us to instantiate structure formulas using the `$` method and captions for them using the `|` method. The '$' method shown below, creates an atom and adds it to the structure, the rest of the structural formula is then be build by chaining atoms together.

Listing 3.19: StructureNLHandler.no

```
Atom $(Text text) {
    Atom a = new Atom(text.realText());
    a.setParent(parent());
    parent().addAtom(a);
    return a;
}
```

The `*` method, followed by a number, defines that the formula is a ring, the number defines the number of edges of this ring. The `-` method creates a single link from the previous atom to the next, `=` creates a double link. The `^` method define that the next atom is above the

previous one, while _ puts the next atom below the previous one. The code for one of these is found below.

Listing 3.20: Atom.no

```
Atom =() {
    return createNext("", "=");
}

private Atom createNext(String atomText, String link) {
    Atom a = new Atom(atomText);
    a.setParent(structure);
    next = a.setLink(link);
    return a;
}
```

All of this code is then transformed into a TeX string that is understood by the LaTeX package `chemfig` [5].

3.2.8 Lilypond

Chapter 4

Implementation

Next to designing a language, a compiler to be able to actually use the language had to be build. How we implemented the compiler, as well as some lower level concepts used in Neio classes, is explained in this chapter.

4.1 Used libraries and frameworks

To be able to build the compiler, we made few of a few open-source libraries and frameworks. They are explained in this section. The number of libraries or framework that we used is rather limited, only three of them were used.

4.1.1 ANTLR4

The ANTLR4 [1] library is an open-source parser generator released under the BSD license. It is used to parse all of the Neio files, the Neio documents and the Neio classes. We used ANTLR4 in the following way. First we created two grammars, consisting of a parser and a lexer, for the Neio documents and classes, thus four files in total. These are written using the ANTLR4 DSL. These grammars are then fed to the `antlr4` command line tool, this tool generates Java classes using the grammars you defined. The ANTLR4 Java classes provide

us with an interface that makes use of the Visitor pattern. Using the visitor pattern, we visit every rule and terminal for a given file. Whilst visiting these, we build an Abstract Syntax Tree (AST) of a given file which we can then manipulate further on. The ANTLR4 grammars and visitor classes thus form the front end of our compiler. The objects that are used in this AST, are objects defined in Jnome and Chameleon.

4.1.2 Chameleon and Jnome

Chameleon and Jnome are both projects that were developed by professor van Dooren and released under the MIT license. Chameleon [2] is a framework that allows you to model a programming language, or as we have shown in this Thesis, a markup language. It does so by providing a lot of objects that represent concepts that are commonly used in programming languages. A few examples of such objects are the following: `Expression, VariableDeclaration, Type, MethodInvocation, InfixOperatorInvocation,...`.

Jnome [10] uses Chameleon and implements the front end and back end of a compiler specifically for the Java language. It can read Java and build an AST from it (using ANTLR3), thus it is a front end for the Java language. To be able to do this it extends the object model available in Chameleon with Java specific objects. It is also able to write out Java code, given an AST built from Jnome and Chameleon objects. This part is the back end of the compiler.

Jnome can read entire Java projects at once, in our compiler for example, it loads the entire Java library. This is needed because our objects use Java objects, such as `String`s. It reads all of the source files and libraries that are specified in a file called `project.xml`. This file was mentioned earlier in Subsection 3.2.6. This same file is used to read Neio projects in our compiler. The `project.xml` for the Neio compiler is shown below.

Listing 4.1: project.xml

```
<project name="Thesis">
    <language name="neio" />
    <baselibraries>
        <library language="Neio" load="true" />
    </baselibraries>
    <sourcepath>
        <source root="../../examples/0.8/lib" />
    </sourcepath>
```

```
9       <binarypath>
10          <jar file="../build/libs/chameleon.jar" />
11      </binarypath>
12  </project>
```

Having obtained an AST in Subsection 4.1.1, we now manipulate it in the middle end of our compiler. After the manipulations have been completed, we write it out using the Jnome back end.

4.2 Compile flow

Now that all parts of the compiler have been introduced, we show a diagram that depicts the flow a source file from Neio source up to to the final LaTeX file.

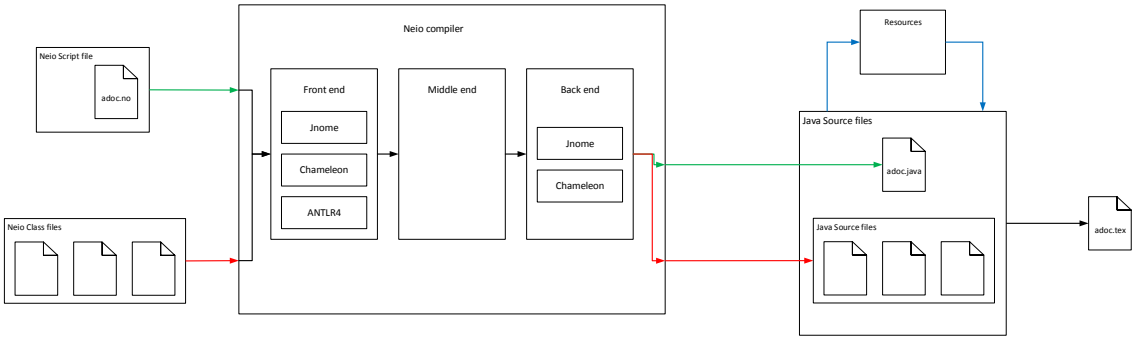


Figure 4.1: Illustration of how a Neio document is compiled. Neio code files translate to Java classes, Neio script files to a Java file with a main method. This then gets output to TeX using the Neio standard library

First of the Java and Neio library (the Neio code files) are read by the front end of the Neio compiler and translated into an AST. Then Neio documents are read by the front end and it is also transformed in an AST. The latter AST is past to the middle end of the compiler and is transformed to an object model that can be output to valid Java. Then the transformed AST and the ASTs for the Neio library are passed to the compiler back end which generates the final Java code. This final Java code is then be compiled and executed again. During the execution, resources might be created, if so, they can be used in the next step. This next

step is the building of the document to a final representation. The way it always happens at is the time is as follows. The `TexFileWriter` is invoked with the root of the AST of the Neio document, this then calls `preTex` and `toTex` on this root. `preTex` allows a document to prepare itself for being printed to LaTeX. A common preparation is to add a package that is uses to the root document. `toTex` does what is says and just creates a LaTeX representation of the object. Once that is completed, it writes out this TeX string to a file named the same as the Neio document but with the TeX extension. Finally the path to this new TeX file is passed to the `TexToPDFBuilder` which compiles the LaTeX code. The code for these two files is shown below.

Listing 4.2: TexFileWriter.no

```
namespace neio.io;

import java.nio.file.*;
import java.util.Arrays;
import neio.lang.Content;

/**
 * Recursively converts Content into a TeX string and prints it to a file
 */
class TexFileWriter implements Writer;

Content content;

/**
 * Initializes the writer with a root content
 *
 * @param content The root Content
 */
TexFileWriter(Content content) {
    this.content = content;
}

/**
 * Creates the TeX string
 *
 * @param name The name of the file to print to (without extension)
 * @return The path to the created file
 */
String write(String name) {
    content.preTex();
    String toWrite = "% Name: " + name + "\n" + content.toTex();
    Path path = Paths.get(name + ".tex");
    Files.write(path, Arrays.asList(toWrite.split("\n")));
    System.out.println("Wrote " + path.toAbsolutePath());

    return path.toAbsolutePath().toString();
}
```

Listing 4.3: TexToPDFBuilder.no

```
namespace neio.io;

import java.io.File;
import java.util.List;
import java.util.ArrayList;

/**
 * @author Titouan Vervack
 */
class TexToPDFBuilder implements Builder;

/**
 * Compiles a TeX file
 *
 * @param name The name of the file to compile (without extension)
 */
void build(String name) {
    if (name != null) {
        List<String> command = new ArrayList<String>();
        command.add("latexmk");
        command.add("-pdf");
        command.add("-dvi-");
        command.add("-bibtex");
        command.add("-lualatex");
        // Make sure the delimiters are correct
        command.add(new File(name).getAbsolutePath());

        ProcessBuilder pb = new ProcessBuilder(command);
        pb.redirectOutput(ProcessBuilder.Redirect.INHERIT);
        pb.redirectError(ProcessBuilder.Redirect.INHERIT);
        pb.start().waitFor();

        String texlessName = name.substring(0, name.length() - 4);
        name = texlessName + ".pdf";
        System.out.println("Wrote " + name);

        command.clear();
        command.add("latexmk");
        command.add("-c");
        command.add(name);

        ProcessBuilder pb2 = new ProcessBuilder(command);
        pb2.redirectOutput(ProcessBuilder.Redirect.INHERIT);
        pb2.redirectError(ProcessBuilder.Redirect.INHERIT);
        pb2.start();
        System.out.println("Cleaned files for " + texlessName);
    }
}
```

4.3 Translation

4.3.1 Reasons for choosing Java

The reason we chose for Java was multi-fold. First and foremost, we already had a front- and back end available for Java thus this saved us a lot of time. A second reason is because Java is platform independent which does not restrict us to a single operating system. Java is also very well known in the current world of computer science, which is why we chose to base our programming model on it. Since the Neio semantics now resemble the Java semantics so well, the reasons for going with Java are enforced.

4.3.2 Fluent interface

We tried to make use of fluent interfaces as much as possible, in the design of the language, by using method chains. But also in the libraries we implemented in Section 3.2. We chose to this as it imposes less boilerplate code (no constant variable declarations for example) and because it is clearer to read what we are doing. Instead of passing a ton of arguments to one method, we split it up in multiple methods and build up one object.

4.3.3 Outputting

Neio makes use of Context Types, which are not something that exists in other languages as of yet. This means we had to create a custom translation for it. The way we choose to translate it, is by breaking up the method chains created in the Text mode of a Neio document at every method call. A variable is then assigned to the output of this method call.

To know what to call our method on, `ContextType`s were implemented as an extension of `RegularJavaType`s in Jnome. A `RegularJavaType` is what we know as a Java class. This `ContextType` adds a virtual inheritance relation between itself and the actual type of the method, and one between itself and the ContextType of this method. The latter ContextType represents the Context at that point as we saw in Section 2.6.

When a lookup reaches a `ContextType` and asks us what our inherited members are (the members you normally receive from super classes), we return the members of all our inheritance relations. We only have two inheritance relations, the actual type and the ContextType. If what we are looking for was somewhere we could see, we return it and we are able to make use of it. Lookups are provided by Chameleon and explaining how exactly they work is out of the scope of this Thesis.

We also hold a Stack of all the variables that have already been declared. When a lookup for a method, described above, succeeds, we check what type is returned by this method and look for the first variable in our stack of that type, popping everything on the way there. We then know what to call our method on.

Lastly, we also have to translate `this`. The translation is analogue to the previous step, where we searched what to call the method on. To have a starting point, we just replace `this` with the last defined variable and let the `ContextType`s do their job. A translation for the very first example we looked at in this document is shown below.

Listing 4.4: testInput.no

```
[Document]
# Chapter 1
## Chapter 2
# Chapter 2

This document contains {nearestAncestor
    (Document.class).directDescendants(
    Chapter.class).size()} top-level
    chapters.
```

```
new Document()
    .newline()
    .newline()
    .#("Chapter 1")
    .newline()
    .text("This is some text in the
        first Paragraph");
```

Listing 4.5: testInput.java

```
package testInput;

import neio.fib.*;
import neio.io.*;
import neio.lang.*;
import neio.stdlib.*;
import neio.thesis.*;
import neio.stdlib.chem.*;
import neio.stdlib.graph.*;
import neio.stdlib.math.*;
import neio.stdlib.music.*;
import neio.stdlib.uml.*;
import java.util.*;
```

```
14
15 public class testInput {
16     public static void main(String[] args) {
17         Document input = new Document();
18         createDocument(input);
19         finishDocument(input);
20     }
21
22     public static void createDocument(Document input) {
23         NLHandler $var0 = input.newline();
24         Chapter $var1 = input.hash(new neio.lang.Text("Chapter 1 "));
25         NLHandler $var2 = $var1.newline();
26         Chapter $var3 = $var1.hash(new neio.lang.Text("Chapter 2"), 2);
27         NLHandler $var4 = $var3.newline();
28         Chapter $var5 = input.hash(new neio.lang.Text("Chapter 2"));
29         NLHandler $var6 = $var5.newline();
30         NLHandler $var7 = $var6.newline();
31         Paragraph $var8 = $var7.text(new neio.lang.Text("This document
               contains ").text(new neio.lang.Text((""") + ($var5.nearestAncestor(
               Document.class).directDescendants(Chapter.class).size()))).text(
               new neio.lang.Text(" top-level chapters.")));
32     }
33
34     public static void finishDocument(Document input) {
35         java.lang.String $var0 = new TexFileWriter(input).write("testInput");
36         new TexToPDFBuilder().build($var0);
37     }
38
39 }
```

Note that the middle end has created three methods. The first one is the main method, it allows the document to be executed as a whole. It initialises the document class and then calls `createDocument` which actually creates an object model that represents the document. Finally `finishDocument` compiles the document to TeX and create a PDF of of it.

The reason we split up this process is so that we can call the creation of the document separately. This allows us to include a document into another document by just calling the `createDocument` method with an appropriate argument.

The middle end also imported every namespace in the Neio library as well as the `java.util` namespace as those might be needed. The name of the Java class that is generated is the same as the name of the Neio document.

It is important to note that we only split up the methods chains created in Text mode. The code written in code mode is kept as is, only occurrences of `this` are replaced.

4.3.4 Reflection

In Subsection 2.7.1 we touched on `addClassMapping` and said that the class mapping was comparable to a factory. This is true and the need for it appeared when we wanted to create drop in replacements for objects that are created in Neio Text mode. For example, if we want to use a special kind of `Chapter` in the `Document` document class, we would have to create a new document class that overrides the `#` method, we also have to override the nested `#` method in `Chapter` because otherwise only the first `Chapter` would be an instance of the special `Chapter`.

This is a problem that is normally solved by using factories. When you know your object can create instances of other objects, in our example a `Document` creates `Chapter`s, that might be changed with more specialized forms of that object, you create a factory. This factory however would have to be generic enough to be able to be used almost anywhere. Even if we are able to do this, the writer of the specialized object would still have to create a new factory. Factories usually are also very similar to each other, yet the code can not be reused.

For this reason we chose to swap out new calls with calls that use reflection to initialize the object. This is a process that is executed in the middle end of the compiler. Only the new calls of objects extending `Content` have been replaced as those should be the only thing a user wants to replace. Every new call gets replaced by the `getInstance` call in the `Content` class. The code needed to understand this process, as well as an example call is given below.

Listing 4.6: Content.no

```
1  // Determines as what class new Content instances will be instantiated
2  private final Map<Class, Class> _classMapping;
3  /**
4   * Returns the Class as which a Class should be instantiated
5   *
6   * @param klass The key of which we want to get a mapping
7   * @return The mapping found for key {@code klass}
8   */
9  final Class classMapping(Class klass) {
10     return _classMapping.get(klass);
11 }
12
13 /**
14  * Overrides default behaviour and instantiates new instances of {@code
        oldClass} as instances of {@code newClass}
15  *
16  * @param oldClass The old class object
```

```
17  * @param newClass The new class object
18  */
19 public final <T> void addClassMapping(Class<T> oldClass, Class<? extends T>
       newClass) {
20     _classMapping.put(oldClass, newClass);
21 }
22
23 /**
24  * Used instead of a new call. This allows to create more specific objects
        than originally
25  * specified. e.g. this allows to create SpecialChapter's instead of Chapter's
26  * <p>
27  * It will create an instance of {@code klass} in case the class mapping has
        not been overridden.
28  * If the mapping was overridden, an instance of the overriding class will be
        created.
29  *
30  * @param klass      The class to instantiate
31  * @param paramTypes The types of the parameters
32  * @param params     The parameters to instantiate {@code klass}
33  * @return The instantiated object
34  */
35 public final <T> T getInstance(Class<T> klass, Class[] paramTypes, Object[]
       params) {
36     // Have I been overridden?
37     Class result = classMapping(klass);
38     Content current = parent();
39     // Has any of the parents been overridden?
40     while ((current != null) && (result == null)) {
41         result = current.classMapping(klass);
42         current = current.parent();
43     }
44
45     // No parents have been overridden
46     if (result == null) {
47         return instantiate(klass, paramTypes, params);
48     }
49     // A parent was overridden, use that type
50     else {
51         return (T) instantiate(result, paramTypes, params);
52     }
53 }
54
55 /**
56  * Does the actual instantiation of an object using Reflection
57  *
58  * @param klass      The class to instantiate
59  * @param paramTypes The types of the parameters
60  * @param params     The parameters to instantiate {@code klass}
61  * @return The instantiated object
62  */
63 private <T> T instantiate(Class<T> klass, Class[] paramTypes, Object[] params)
        {
64     return klass.getConstructor(paramTypes).newInstance(params);
65 }
```

```
1 Chapter chapter = getInstance(Chapter.class, new Class[]{Text.class, Integer.
      class}, new Object[]{title, 1});
```

If you already understood the code from the comments, this paragraph can be skipped. Every

instance of content has a mapping from `Class` to `Class`, by default this mapping is empty. When we want to use a specialized form of an object we call `addClassMapping`, as we did in Subsection 2.7.1. Take the following example.

```
1 [Document]
2 # Chapter 1
3 {
4     nearestAncestor(Document.class).addClassMapping(Enumerate.class,
          FibEnumerate.class);
5 }
```

We added a mapping from `Enumerate.class` to `FibEnumerate.class` in the root `Document`. When we now call `getInstance(Enumerate.class, ...), the root of the call is Chapter, not Document`. But fear not, class mappings are looked up recursively. In the `getInstance` method we first check if we have a mapping for the class we are trying to instantiate. If we do the value of the mapping is used, if not we recursively check the mappings of our parents. If no mapping has been found after this, we just instantiate the class that has been given as argument to `getInstance`.

4.3.5 Escaping

A Neio document is parsed and converted a number of times. First, it is parsed by the Neio compiler, then it is transformed into Java code that is then converted into LaTeX code (in most cases). This means that we have to handle escaping of characters in three different languages. Escaping in Neio is easy, just add a \ before the character you want to escape and it is escaped. Of course \`[ntr]` hold on to their special meaning. However \`[bfu]` have not been included in Neio as they are not yet needed. In the future it might prove useful to include these special characters to.

Moving on, we can not just translate these characters straight to Java as we have no special meaning for \`[bfu]`. For this reason the middle end transforms every backslash into two backslashes, and transforms a double backslash (used to denote an escaped single backslash in Neio) to four backslashes in Java as this is how you represent an escaped backslash in a Java String. We then do some further translation in the Text class to assure that we are able to create valid latex. LaTeX only has ten special characters thus replacing is quite easy. The

code to do the replacing is shown below.

Listing 4.7: Text.no

```
1
2  /**
3   * Creates the TeX representation of this Text
4   * Printing out {@code realText} or the TeX representation of {@code text}
5   * Replaces special characters to create valid TeX
6   *
7   * @return The TeX representation of this Text
8   */
9  String thisToTex() {
10     String me = "";
11     if (text == null) {
12         // Create valid TeX text
13         me = realText.replaceAll(BS + BS, ESC + "textbackslash{}");
14         me = me.replaceAll(BS + "([^#$%&~\\^_{}" + BS + "])", "$1");
15         me = me.replaceAll(ESC, BS);
16         me = me.replaceAll(BS + "~", BS + "textasciitilde{}");
17         me = me.replaceAll(BS + "\\^", BS + "textasciicircum{}");
18     } else {
19         me = text.toTex();
20     }
21
22     return me;
```

4.4 Resource usage and creation

4.5 Limitations

Whilst development of the compiler a few limitations were discovered, they are discussed in this section.

4.5.1 Windows

In Section 4.4 we saw that command line commands are sometimes executed. Executing commands using the Java Process class, however came with some not immediately visible issues. When developing on Linux, executing shell commands from Java worked perfectly fine and gave no problems whatsoever. However, on Windows some of the processes would hang, some would always hang, some would hang only once in a while. The reason for this is that

Windows offers only a limited buffer size for the input and output streams of a process, the program deadlocks if the input is not written or read. After closer inspection of the Javadoc [8] this became apparent but since the API for the Process class is so simple, it was a surprise nevertheless. The fact that we have to worry about different platforms on a cross-platform language as Java is also something that is not too common.

4.5.2 Back end

About every back end has some limitations on how we can translate our Neio document. This is because Neio can use symbols as methods. In Java, methods can only contain letters, numbers, $ or _. This means we have to translate the symbol method to valid Java identifiers. The way we do this is very straightforward. We use the following mapping and just replace any occurrence of a key in a method or method invocation with the value.

```
"#" -> "hash"
"*" -> "star"
"=" -> "equalSign"
"^" -> "caret"
"-" -> "dash"
"_" -> "underscore"
"`" -> "backquote"
"$" -> "dollar"
"|" -> "pipe"
```

With other back ends we would do the same, some languages allow you to overwrite operators but doing this might lead to unexpected side effects, thus just translating it this way seemed like the safest way to go.

Chapter 5

Future work

As mentioned before in Section 2.10 there are still a few things that could be done in the future. There are however a lot more things that could be done and we discuss them in the rest of this chapter.

5.1 Automatisation

Automatic StringBuilder

Since we are constantly translating to TeX and other formats, we use `StringBuilder` very often. An example is almost every `toTex` method, we show the one defined in `Content` as an example.

Listing 5.1: Content.no

```
1  /**
2   * Returns a String representing this Content and its children as TeX.
3   * The string is build by recursively calling {@code toTex} on all of children
4   * in this Content.
5   *
6   * @return The TeX string
7   */
8  String toTex() {
9      StringBuilder tex = new StringBuilder();
10     for (int i = 0; i < content.size(); i = i + 1) {
11         tex.append("\n").append(content.get(i).toTex());
```

```
12      }
13
14      return tex.toString();
15  }
```

In fact we use it so many that in the future it might be better to just replace `String`s by `StringBuilder`s at all times. This would make it easier for people to write packages. The performance gain, or loss of this would have to be investigated though.

Automatic return of the object itself

Another pattern that is often encountered is that we often return the same object even when we have not changed it. This is to be able to make use of fluent interfaces, as said before. An example from the `InlineEq` class is shown below.

Listing 5.2: InlineEq.no

```
1  /**
2   * Creates a square root and adds it to this equation, returning this to allow
        further chaining
3   *
4   * @param root The base of the square root
5   * @param arg  The value to take the square root of
6   * @return This equation
7   */
8  InlineEq sqrt(Content root, Content arg) {
9      content().add(new Sqrt(root, arg));
10     return this;
11 }
```

We actually already have a solution for this, constructors. Constructors do not tell us in their signature what they are going to return, but we do know what they are going to return. No explicit return is needed either. Implementing this for other methods would allow us to lower the amount of boilerplate code that has to be written.

Automatic Text conversion

The final automation is needed to counter what we saw with the math library. There we had to open an inline code block just to open Text mode because that creates a `Text`. A

way to prevent this would be to automatically convert `Content`, or maybe even `Object` in a generic way. In Java this is done through the `toString` method while in python the `repr` and `str` methods are used for this. However if we were to implement it as naively as this, we would lose any kind of static typing when a Text is expected as an argument. This does not seem like a good thing, especially when you consider that not every object might have a real textual representation. This means that automatically transforming an object to text might in a lot of cases not even create a sensible result.

A better way might be to define an interface, `Representable`, that has a `repr` method. Any class that implements this class, and the native types such as String and Integer, could than be transformed into a Text using this `repr` method.

5.2 Moving content

When we assign content to a variable, and apply small changes to it later on, it is possible that the content moves or that both the original and the new content are changed. This is because the language does not enforce every placed content to be static and in turn this is the responsibility of the library developer and the user of those libraries. We showed that we had to make sure that we cloned our objects when making changes to them before in Subsection 3.2.5 and Subsection 3.2.8. It would be a very nice feature if Neio could enforce this behaviour by itself and take care of the implementation of clone by itself. A way that this might be implemented is by copying objects that are going to be changed through reflection.

5.3 Use

5.4 Packages

Package support

No package system was provided in this thesis, but in the future however it would be beneficial to do so. It would allow users without a lot of technical expertise to import new functionality easily. The way packaging is handled right now is straight through LaTeX packages and Java packages. We allow you to add LaTeX packages through the `addPackage` method in `Document` and we allow you to import Java classes through the regular `import namespace` statements. To actually include a Neio library in the document, right now the source code for it has to be put into a folder, next to the rest of the standard Neio library. In the future we should be able to download (manually or automatically) packaged libraries and have them be stored somewhere else.

Implementation of packages

As we said before, non of the document classes or libraries that were created in this thesis are complete, and we also did not create that many of them, in comparison to what Java and LaTeX offer that is. In the future a lot of work should be put into writing complete libraries and porting libraries from other available languages such as LaTeX. This can however be done by the community for a big part, as is what happens for Java and LaTeX.

5.5 Compiler improvement

Efficiency

The compiler is fast enough for small documents but for a large document like this thesis, the compile time starts to ramp up. The thesis takes about 30 seconds to a minute to compile,

depending on the compute power of your computer. This is on top of the `latexmk` that has to run afterwards and that also takes quite a while. However no attention was paid to the efficiency of the compiler in this thesis, as thus it is likely that there are a lot of optimizations that can be performed.

Other front- and back ends

Another compiler improvement that could be created in the future, is that we could create different front and back ends. We do not have to output to Java, if someone were to write a back end for python for example, we could make use of that back end. The same thing is possible for the front end. The syntax of Neio documents and code files could be changed, as long as the fundamental design decisions of the language are kept the same, such as `ContextType`s and nested methods, another syntax could be designed. As long as a front end is developed that can parse a Neio document with a different syntax that can still parse the document into the same AST as is used at this moment, it is possible to change the currently used syntax..

5.6 Tool improvement

To effectively be able to use Neio, we need some improvements to the tools available right now. Chameleon provides support for the Eclipse IDE and in the future this could be used to improve our tooling.

Syntax highlighting

One of the base requirements to be able to work with a language is syntax highlighting. This is one of the features that Chameleon provides through its Eclipse integration. However it was not made for a language such as Neio where we represent a document as a series of method chains. At this moment the Eclipse integration treats the entire method chain as a whole instead of every method call separately. This would cause problems as everything between

codeblocks would be colored the same and we would not be able to have different colouring for special methods, such as the symbol methods.

It should however work out of the box for Code mode and the code files.

Auto completion

A second feature that Chameleon offers and that is very useful when writing in a certain language, is auto completion. This should again work well in Code mode and code files, however it would be a bit harder in Text mode. This is because it would have to know when to auto complete and when it should just let us write. It might be best to hide the auto complete behind a hotkey while in Text mode. This still allows the user to get a list of methods he could use at a certain time, but would remove the annoyance of having a pop-up for auto completion come up at every word that is written.

IDE with preview

The last thing that would be a very nice to have for a markup language like this, would be an IDE that previews the document. This is not something that is supported by Chameleon and would thus require quite a bit more work. The compiler in its current state is also not ready for this, as said before it gets quite slow on large documents. As it would have to be constantly running in the background, it would use a lot of energy (which is problematic for laptops) and would be too slow to actually produce results in time.

Chapter 6

Conclusion and reflection

Summary

In this thesis we researched the current state-of-the-art concerning document creation (Chapter 1). From this research we concluded that a good combination of user-friendliness and customisability was hard to find. To achieve such a combination we created a new markup language, called Neio, that uses a modern, object-oriented programming model.

We decided on the features (Chapter 2) that Neio needed to achieve its goals and used them to create the examples in Chapter 3. These examples were created to show the simplicity and flexibility of Neio.

To be able to prototype Neio, we also created a compiler and a standard library for it (Chapter 4).

Finally, we discussed the work that can and should still be done on Neio in the future in Chapter 5.

Reflection

We used the language for four months (the rest of the time was spent developing it), wrote a Thesis in it and wrote a complete compiler for it. After all of this we found that it might

have been better to just use almost pure Java syntax for the code mode. We changed a few things, such as adding syntactic sugar for new calls and calling packages, namespaces. The latter was done because in our eyes, namespace is a more general term, that better explains what this keyword does.

But by choosing different keywords, we ended up with name conflicts. At some point in the development we wanted to make use of a class in a package called `namespace`. Of course this did not work out well as the parser did not know how to parse a package name containing a keyword. By changing the syntax as much as we did, it also became impossible to run Javadoc on our code files without making major changes to them.

Disallowing a substantial part of Java (anonymous classes, shorthand operators such as `++` and `+=`,...) also hurts the adoption rate of the language as you almost always to invest some time converting a Java file into a Neio file.

We think that it would have been better to have just used Java syntax, with the modifications defined in Section 2.4, for the code mode. In the future we might then also add the proposition made in Subsection 5.1.

Conclusion

In the end, we showed that is possible to create a healthy mix between user friendliness and a powerful programming model. We showed that the language can be as easy to read, write and learn as Markdown. While it can also be as powerful and diverse as LaTeX whilst remaining quite legible. A lot of work can still be done, and should still be done to allow the widespread use of Neio, as is shown in Chapter 5, but we think that good basis has been provided. The completion of this book and the diverse examples in Chapter 3 should be enough proof of this.

Bibliography

[1] Antlr4. http://www.antlr.org/. Accessed: 18-05-2016.

[2] The chameleon framework. https://github.com/markovandooren/chameleon. Accessed: 18-05-2016.

[3] Stephen Checkoway, Hovav Shacham, and Eric Rescorla. Are text-only data formats safe? or, use this latex class file to pwn your computer. In *LEET*, 2010.

[4] Steve Checkoway, Hovav Shacham, and Eric Rescorla. Dont take latex files from strangers. 2011.

[5] Chemfig latex package. https://www.ctan.org/pkg/chemfig. Accessed: 26-05-2016.

[6] Fluent interfaces. http://www.martinfowler.com/bliki/FluentInterface.html. Accessed: 27-05-2016.

[7] Github flavored markdown. https://help.github.com/enterprise/11.10.340/user/articles/github-flavored-markdown/. Accessed: 27-05-2016.

[8] Javadoc for process. https://docs.oracle.com/javase/8/docs/api/java/lang/Process.html. Accessed: 09-05-2016.

[9] Jmathtex: Tex in java. http://jmathtex.sourceforge.net/. Accessed: 08-05-2016.

[10] The jnome framework. https://github.com/markovandooren/jnome. Accessed: 18-05-2016.

[11] Donald E. Knuth and Michael F. Plass. Breaking Paragraphs into Lines. *Software - Practice and Experience*, 11:1119–1184, 1981.

[12] Koma-script guide. `http://texdoc.net/texmf-dist/doc/latex/koma-script/scrguien.pdf`. Accessed: 18-05-2016.

[13] Latex introduction. `https://latex-project.org/intro.html`. Accessed: 20-05-2016.

[14] Lyx homepage. `https://www.lyx.org/`. Accessed: 25-05-2016.

[15] Markdown homepage. `https://daringfireball.net/projects/markdown/`. Accessed: 07-05-2016.

[16] Markdown syntax. `https://daringfireball.net/projects/markdown/syntax`. Accessed: 08-05-2016.

[17] Mediawiki latex extension. `https://www.mediawiki.org/wiki/Extension:Math`. Accessed: 08-05-2016.

[18] Memoir user guide. `http://tug.ctan.org/tex-archive/macros/latex/contrib/memoir/memman.pdf`. Accessed: 18-05-2016.

[19] Metauml. `https://github.com/ogheorghies/MetaUML`. Accessed: 18-05-2016.

[20] Latex minipage. `http://www.sascha-frank.com/latex-minipage.html`. Accessed: 09-05-2016.

[21] Neio source code of this thesis. `https://github.ugent.be/tivervac/neio/blob/master/examples/0.8/input/thesis/thesis/thesis.no`. Accessed: 27-05-2016.

[22] Office object insertion. `https://support.office.com/en-us/article/Insert-an-object-in-Word-or-Outlook-8fc1ea53-0e01-4603-a4cf-98c49b6ea3f5`. Accessed: 27-05-2016.

[23] Pandoc. `http://pandoc.org/`. Accessed: 08-05-2016.

[24] Pandoc filter example. `http://pandoc.org/scripting.html#json-filters`. Accessed: 25-05-2016.

[25] Till Tantau. *The TikZ and PGF Packages*.

[26] The colemak keyboard layout. `http://colemak.com/`. Accessed: 27-05-2016.

[27] The latex beamer package. `https://www.ctan.org/pkg/beamer`. Accessed: 27-05-2016.

[28] The mediawiki homepage. `https://www.mediawiki.org/`. Accessed: 27-05-2016.

[29] The panpipe pandoc filter. `http://chriswarbo.net/git/panpipe/index.html`. Accessed: 27-05-2016.

[30] The stackexchange homepage. `https://www.stackexchange.com/`. Accessed: 27-05-2016.

[31] The wikipedia homepage. `https://www.wikipedia.org/`. Accessed: 27-05-2016.

[32] The wiktionary homepage. `https://www.wiktionary.org/`. Accessed: 27-05-2016.

[33] W3c ebnf specification. `https://www.w3.org/TR/REC-xml/#sec-notation`. Accessed: 27-05-2016.