

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



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



Preface and acknowledgement

In the first place I thank my promotor, professor Marko van Dooren, for all of the guidance, encouragement and feedback he provided to me during the completion of this thesis. The thesis would not have reached the current state without our weekly meetings.

My thanks also go out to my counselor, Benoit Desouter, for proofreading the final work on such short notice.

I also thank Stijn Seghers for providing feedback on my work and for listening to my rambling about said work.

Lastly, I thank my family and friends for their support during this year. In particular I thank my close family, Marino, Nathalie, Valentin and Amaury, for all the encouragement and support they have given me throughout my studies.

Titouan Vervack, June 2016

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.

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 provides a powerful programming model, and that is easy-to-read and write. The developed language is named Neio.

Neio provides two syntaxes named text mode and code mode. Text mode is what most users write most of the time. For this reason it is based on Markdown as the state-of-the-art analysis proved that it was easy to read, write and learn. Code mode is almost a pure copy of Java. This provides Neio with a big user base that can read and understand code mode. It also allows us to reuse existing Java libraries.

To use Neio a compiler was created, and a few libraries were implemented to demonstrate the strength and versatility of Neio. IDE support for Code mode was also provided.

A lot of future work can still be done on Neio, but we believe that a good basis was provided.

Keywords Markup, Object-oriented, LaTeX, Markdown

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

Titouan Vervack

Supervisor: Prof. dr. ir. Marko van Dooren

Counsellor: Dr. ir. Benoit Desouter

Abstract—Neio is a markup language that offers a modern, object-oriented programming model, and that is easy to read, write and learn. The modern programming model allows for full customisation and control of the document. An object-oriented language was chosen because it can easily represent the structure of a document. The design of Neio was based on the results of a the state-of-the-art analysis concerning document creation was researched. To prototype the language, a compiler for it was developed and several libraries were implemented to demonstrate the flexibility of the language. As a demonstration of the power and usability of the language, the thesis itself was written in it.

Index Terms—Markup, Object-oriented, LaTeX, Markdown

I. INTRODUCTION

What You Is What You Get (WYSIWYG) editors, such as Word and Pages, are the best-known solution for document creation. They immediately show the final document and help the user by providing a GUI for all of the features. This allows them to be used by anyone, even people without technical expertise.

However, these solutions do not offer a good solution, outside of the GUI, to manipulate the document. The features offered in the used programming model are not sufficient to allow for full control of the document.

The GUI makes it easy for the user to forget about the structure of a document. As such, the document is often weakly structured, for example text is put in bold instead of using an appropriate style for the element. This allows for inconsistencies to show up in the document and for small changes to have large impacts on the document. A typical example is that images that were placed earlier move around in an unwanted manner later on.

Markup languages put a greater emphasis on the structure and/or customisation of the document. Markdown [1] for example uses a small fixed syntax [2] for the most common elements used in a document. This makes it is very easy to read and write Markdown documents. However, Markdown lacks customisability.

L^AT_EX [3] on the other hand offers full customisation using the Turing complete programming model of TeX. This programming model is complex and harder to use than modern programming models and also requires a certain technical proficiency to be used.

As such, we developed Neio. It had to be as easy to read, write and learn as Markdown but as powerful as L^AT_EX.

II. THE DESIGN OF NEIO

We decided to use two syntaxes in Neio. The first is the text mode syntax. It is the syntax used by most users to write documents. As such it has to be as simple as possible, which is why it is based on Markdown. In the state-of-the-art analysis, Markdown was found to be very easy to read, write and learn. Documents written in text mode are called Neio documents, an example is shown in Listing. 1.

Listing 1: A simple Neio document.

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

The second syntax is called code mode. It is used to provide the objects and methods that are used in a text document. Code mode is almost a pure copy of the Java syntax. Documents written in code mode are called code files and look almost the same as Java class files.

An important difference compared to Java is that in code mode a method identifier can contain symbols. The symbols that can be used are #, -, *, _, \\$, ^, =, |, \. There are also reserved methods for the ‘newline’ and for text.

Java was chosen because it is one of the most popular programming languages [4] of this age. This provides a big advantage because every Java library can be used in Neio. It is also creates a large user base that can already read and understand code mode.

Lastly, an important difference with Markdown is that the semantics of the text mode syntax are not hard-coded into the language, instead they can be defined in the code files. For example, for slide shows, # can be redefined to create a new slide instead of a chapter.

A text document is actually a chain of method calls that create an object model of the document. This object model can then be visualised later on.

This chain is called the `call chain`. The call chain for the example in Listing 1 is shown in Listing 2.

It is also possible to enter code mode in a text mode by opening a pair of curly braces. This is shown in Listing 4.

Listing 2: The call chain of the example in Listing 1.

```

1 new Document()
2   .newline()
3   .newline()
4   .#("Chapter 1")
5   .newline()
6   .text("This is the first paragraph.");

```

III. PROGRAMMING IN NEIO

A number of libraries illustrate the programming capabilities of Neio. Listing 3 shows how the Table class allows the writer to create a table using the `|`, `-`, and `newline` methods. Figure 1 shows the rendered version.

Listing 3: A table in text mode.

Student club	Rounds	Seconds/Round
HILOK	1030	42
VTK	1028	42
VLK	841	51
VGK	810	53
Hermes and LILA	793	54
HK	771	56
VRG	764	57

Student club	Rounds	Seconds/Round
HILOK	1030	42
VTK	1028	42
VLK	841	51
VGK	810	53
Hermes and LILA	793	54
HK	771	56
VRG	764	57

Fig. 1: The rendered version of the example in Listing 3.

The second example shows how to create sheet music and how to reverse the notes in the `Score`.

Listing 4: A Neio document that creates sheet music.

```

1 [Document]
2 {
3   Score s = new Score().c().d().e().f().g().a().
4     b();
5   return s;
6 }
7 The score is read as {s.print()}.
8 {
9   Score reversed = ss.reverse();
10  return reversed;
11 }
12 The reversed score is read as {reversed.print()}.

```



The score is read as c, d, e, f, g, a, b.

Fig. 2: The rendered version of the score in Listing 4.



The reversed score is read as b, a, g, f, e, d, c.

Fig. 3: The rendered version of the reversed score in Listing 4.

In Listing 4, we also see that code mode can be entered in a sentence. This allows us to easily execute simple expressions such as `s.print()` shown in the example.

It is also worth noting that the entire thesis was written in Neio.

IV. COMPILER

The process a Neio document goes through from Neio source code to \LaTeX code, is shown in Figure 5.

The Neio compiler is created using Chameleon [5], [6] and Jnome [7]. Chameleon is a framework for defining abstract ASTs of software languages. It enables the reuse of generic language constructs and the construction of language-independent development tools. It defines language-independent objects (`Element`, `CrossReference`, ...) as well as paradigm-specific ones (`Type`, `Statement`, `Expression`, ... for Object-Oriented Programming). Jnome is a Chameleon module for Java 7 that provides language specific objects. These libraries allowed us to reuse most language semantics, and to obtain IDE support with only a few lines of code. The architecture is shown in Figure 4.

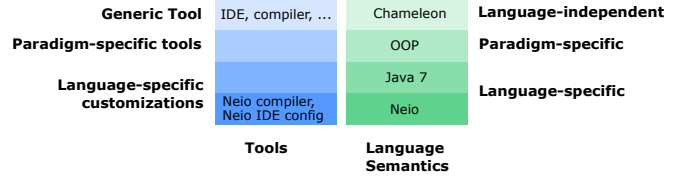


Fig. 4: The Chameleon architecture.

The compiler also copies resources, such as images and configuration files to the output folder.

The final transformation to \LaTeX is done by the Neio standard library. To generate the \LaTeX code the `toTex` method is called on the root of the object model which recursively calls it on all the other objects. The `toTex` method can also call other programs. To produce the scores in Figure 2 and 3, it calls LilyPond [8], which generates a PDF that is then imported in \LaTeX .

REFERENCES

- [1] "Markdown homepage," <https://daringfireball.net/projects/markdown/>, accessed: 07-05-2016.
- [2] "Markdown syntax," <https://daringfireball.net/projects/markdown/syntax>, accessed: 08-05-2016.
- [3] "LaTeX introduction," <https://latex-project.org/intro.html>, accessed: 20-05-2016.

- [4] N. Diakopoulos and S. Cass, “The top programming languages 2015 according to iee spectrum,” <http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2015>, accessed: 30-05-2016.
- [5] “The Chameleon framework,” <https://github.com/markovandooren/chameleon>, accessed: 18-05-2016.
- [6] M. van Dooren, E. Steegmans, and W. Joosen, “An object-oriented framework for aspect-oriented languages,” in *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, ser. AOSD ’12. New York, NY, USA: ACM, 2012, pp. 215–226. [Online]. Available: <http://doi.acm.org/10.1145/2162049.2162075>
- [7] “The Jnome framework,” <https://github.com/markovandooren/jnome>, accessed: 18-05-2016.
- [8] “The LilyPond homepage,” <http://lilypond.org/>, accessed: 30-05-2016.

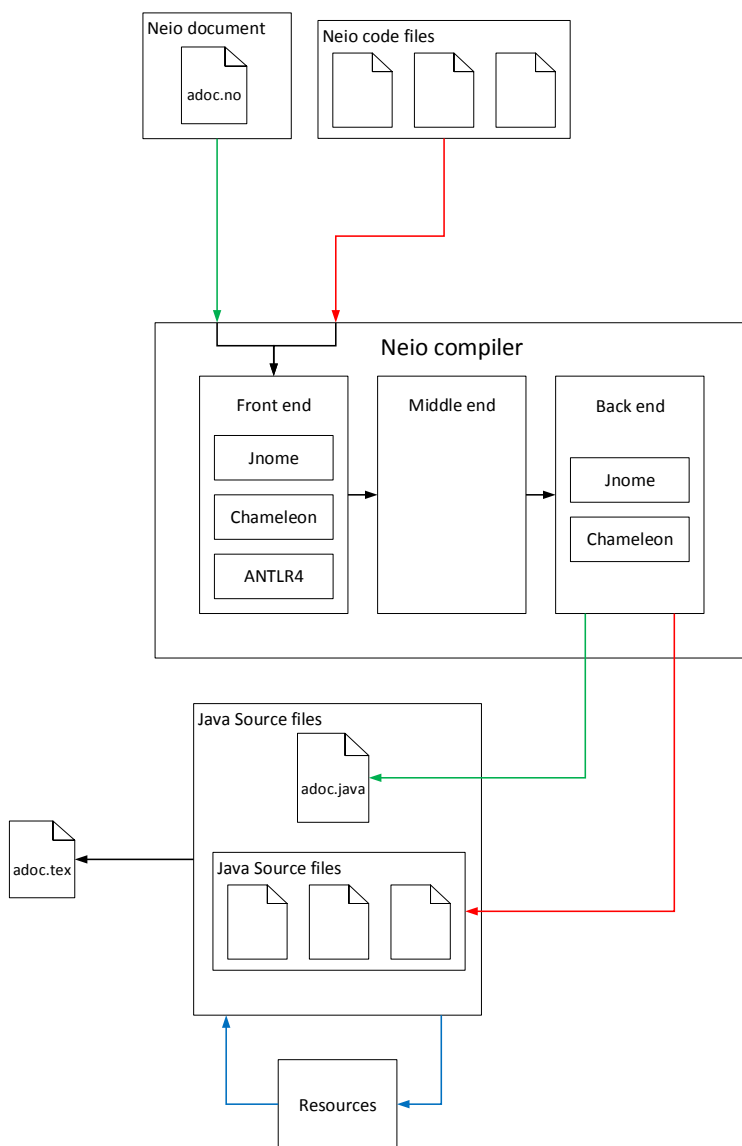


Fig. 5: The process of compiling Neio source code to \LaTeX .

Contents

Preface and acknowledgement	i
Permission for usage	ii
1 Introduction	1
1.1 State-of-the-art	1
1.1.1 Word processors	3
1.1.2 L ^A T _E X	5
1.1.3 Markdown	8
1.1.4 Pandoc	9
1.2 Problem statement	11
1.2.1 Word processors	12
1.2.2 L ^A T _E X	13
1.2.3 Markdown	14
1.2.4 Pandoc	15
1.3 Proposed solution: Neio	16
1.3.1 Target group	17
2 Design of the Neio markup language	18
2.1 Neio document	18
2.2 Code files	19
2.3 Call chain	21
2.4 Method modifications	23
2.4.1 Symbol methods	23
2.4.2 Text	23
2.4.3 Surround methods	24

2.4.4	Nested methods	25
2.4.5	Newlines	27
2.5	Context types	31
2.6	Code blocks	42
2.6.1	Non-scoped code	43
2.6.2	This	44
2.6.3	Scoped code	45
2.6.4	Inline code	46
2.7	Text in code mode	47
2.8	Navigating through the lexical structure	48
2.9	Considerations	50
2.9.1	Static typing	50
2.9.2	Security	50
3	Supported document types and libraries	52
3.1	Document classes	52
3.1.1	Document class definition	52
3.1.2	Document	53
3.1.3	Book	55
3.1.4	Abstract	56
3.2	Libraries	58
3.2.1	BibTeX	58
3.2.2	References	60
3.2.3	L ^A T _E X math and amsmath	61
3.2.4	L ^A T _E X tables	63
3.2.5	Red black trees	66
3.2.6	MetaUML	69
3.2.7	Chemfig	72
3.2.8	Lilypond	74
4	Implementation	77
4.1	Compile process	77

4.2	Used libraries and frameworks	79
4.2.1	ANTLR4	79
4.2.2	Chameleon and Jnome	80
4.3	Translation	83
4.3.1	Reasons for choosing Java	83
4.3.2	Fluent interface	83
4.3.3	Translation to Java	84
4.3.4	Reflection	86
4.3.5	Escaping	88
4.4	Resource usage and creation	89
4.5	Limitations	89
4.5.1	Java	89
4.5.2	Windows	89
4.5.3	Back end	90
5	Future work	91
5.1	Automatisation	91
5.2	Moving content	93
5.3	Use	94
5.4	Packages	94
5.5	Compiler improvement	94
5.6	Tool improvement	95
6	Conclusion and reflection	97

List of Figures

1.1	The detection of an error.	2
1.2	The warning before trying to repair the error.	2
1.3	The inability to repair the error.	3
1.4	An Excel chart in a Word document.	4
1.5	Kerning on	7
1.6	Kerning off	7
1.7	Ligature on	7
1.8	Ligature off	7
1.9	The document rendered as HTML by the official Markdown conversion tool. .	9
1.10	The output of the document to the left.	10
1.11	The output of the document to the left.	11
1.12	Caption below the image.	13
1.13	The image moved, but the caption did not.	13
1.14	The table document rendered as HTML by the official Markdown conversion tool.	15
2.1	The rendered form of the document to the left.	19
2.2	The object model of the previous document.	22
2.3	The rendered version of the document to the left.	25
2.4	The tree representation of the call chain of the previous example.	32
2.5	The first context type: ct0.	33
2.6	The context types ct0 and ct1.	34
2.7	The context types ct0 through ct2.	35
2.8	The context types ct0 through ct5.	36
2.9	The context types ct0 through ct6.	37

2.10	The object model of the previous example.	39
2.11	Correct context lookup for *.	39
2.12	Wrong context lookup for *.	40
2.13	The correct context type representation.	41
2.14	The wrong context type representation.	42
2.15	The rendered form of the example to the left.	47
2.16	The rendered form of the example to the left.	48
2.17	The rendered version of the example to the left.	49
3.1	The rendered form of the example to the left.	62
4.1	The process of compiling Neio source code to LaTeX.	78
4.2	A code file with syntax highlighting, a syntax error and a dependency view.	82

Listings

1.1	document.md	9
1.2	pandocExample.md	10
1.3	behead2.hs	10
1.4	letterTemplate.latex	11
1.5	letter.md	11
2.1	A simple Neio document.	19
2.2	Paragraph.no	19
2.3	Document.no	21
2.4	Text.no	24
2.5	surroundEx.input	25
2.6	Chapter.no	26
2.7	TextContainer.no	26
2.8	Two lists	27
2.9	One list	27
2.10	TextContainer.no	28
2.11	NLHandler.no	28
2.12	ParNLHandler.no	29
2.13	ParNLHandler.no	29
2.14	A table created using a DSL.	30
2.15	An example for context resets.	38
2.16	Automatic return from code block.	44
2.17	Manual return from code block.	44
2.18	template.no	47

2.19	dice.no	48
2.20	Content.no	49
3.1	Content.no	53
3.2	Thesis.no	55
3.3	thesis.no	56
3.4	overview.no	57
3.5	AbstractNLHandler.no	57
3.6	Bibtex.no	59
3.7	Document.no	59
3.8	Chapter.no	61
3.9	Referable.no	61
3.10	An ASCII table created in Neio.	63
3.11	TextContainer.no	64
3.12	TableRow.no	65
3.13	TableNLHandler.no	65
3.14	rbtDocument.no	66
3.15	rbtLatex.tex	68
3.16	StructureNLHandler.no	73
3.17	Atom.no	73
3.18	An example musical score.	74
3.19	Score.no	74
3.20	Score.no	76
3.21	The code of the example.	76
4.1	ClassLexer.g4	79
4.2	ClassParser.g4	79
4.3	project.xml	80
4.4	testInput.no	84
4.5	testInput.java	84
4.6	Content.no	86
4.7	Text.no	88
5.1	Content.no	91

5.2	InlineEq.no	92
-----	-----------------------	----

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 with the exclusion of the title page (which is auto-generated), the extended abstract and a few images. The source code of the book is available on the UGent GitHub [23]. In the same repository you also find the Neio library [22] and source code for the Neio compiler [32].

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 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.

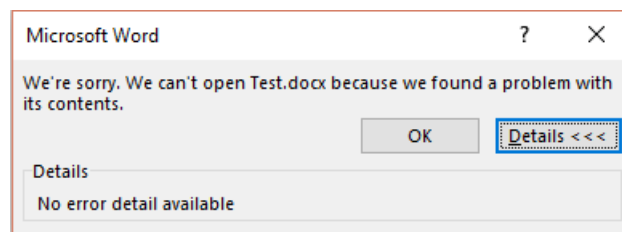


Figure 1.1: The detection of an error.

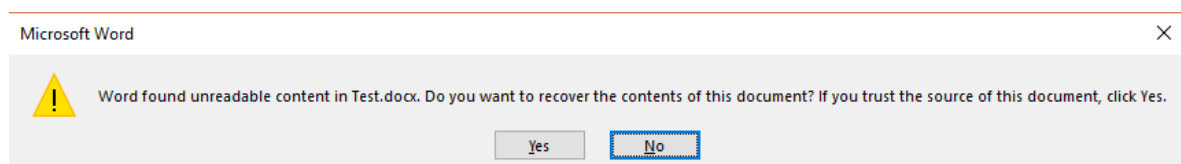


Figure 1.2: The warning before trying to repair the error.

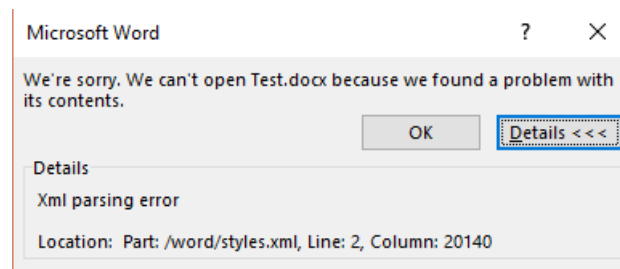


Figure 1.3: The inability to repair the error.

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 [24]. 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.

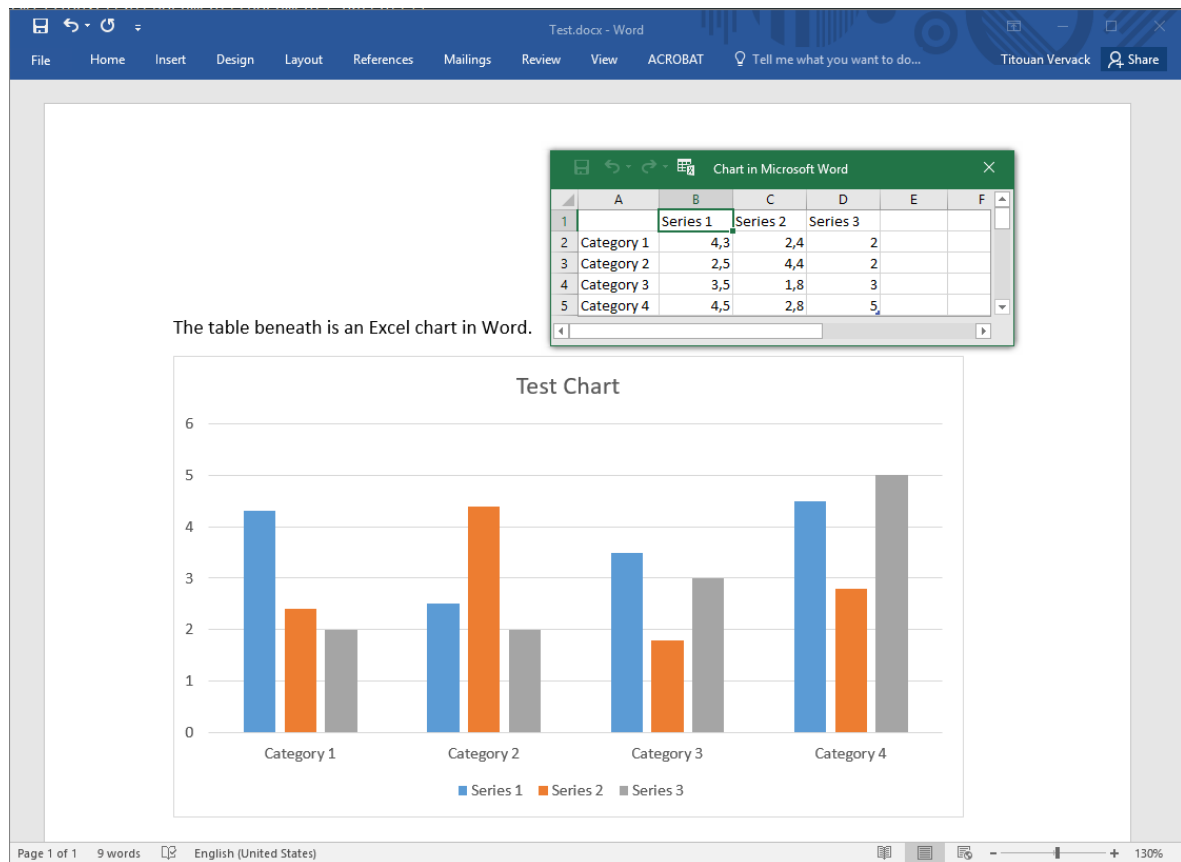


Figure 1.4: An Excel chart in a Word document.

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 L^AT_EX.

1.1.2 L^AT_EX

L^AT_EX is a document preparation system that uses the TeX typesetting system. It provides a set of extra macros that can be used on top of the TeX macros.

L^AT_EX provides rich customisability through the Turing complete programming model provided by TeX. 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 L^AT_EX, 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. L^AT_EX 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:

```

1 \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:

```

1 \newcommand\example{Hello}
2 \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 do that using `\expandafter` as shown below.

```

1 \newcommand{\example}{Hello}
2 \expandafter\renewcommand\expandafter\example\expandafter{\example{} world}
3 \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 [29] 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 [14] 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 [12]. 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 then 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 [24].

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 [18] [31]: a free and open-source software package that powers sites as Wikipedia [35] and Wiktionary [36].
- Stack Exchange [34]: a network of Q&A web sites
- JMathTex [10]: 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 [16] 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 [17] 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

```
1 Markdown
2 =====
3
4 It is:
5 * Simple
6 * Easy to read
7 * Easy to write
```

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 [25] 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

```

1 # Pandoc
2 This pandoc document uses \LaTeX{} and
  markdown in one file!
3
4 \begin{equation*}
5 \sqrt[3]{125}=25
6 \end{equation*}

```

Pandoc

This pandoc document uses \LaTeX and markdown in one file!

$$\sqrt[3]{125} = 25$$

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 [26].

Listing 1.3: behead2.hs

```

1 #!/usr/bin/env runhaskell
2 -- behead2.hs
3 import Text.Pandoc.JSON
4
5 main :: IO ()
6 main = toJSONFilter behead
7   where behead (Header n _ xs) | n >= 2 = Para [Emph xs]
8         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 [33], 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.

```

1 This document executes a shell command.
2
3 ```{pipe="sh"}
4 echo "Hello world! I am " $(whoami)
5 ```

```

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

```

1 \documentclass{letter}
2 \signature{Titouan Vervack}
3 \begin{document}
4 \opening{Dear $addressee$,}
5
6 $body$
7
8 \closing{Sincerely,}
9 \end{letter}
10 \end{document}

```

Listing 1.5: letter.md

```

1 How have you been?
2 I've been longing to see you, but I've
   been very busy...
3
4 My son's getting married next week and
   I would like to invite you!
5 Well, you and your wife of course.
6
7 If you're planning on coming, please
   give me a call.

```

The document can then be build with the following command.

```

1 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.

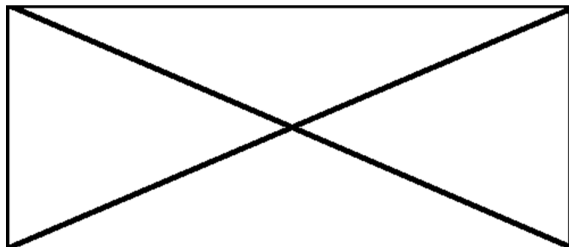


Figure 1 This is the first figure

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.

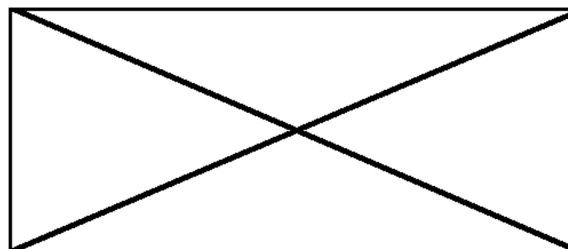


Figure 1.13: The image moved, but the caption did not.

1.2.2 L^AT_EX

Even though L^AT_EX offers customisability through the programming model of TeX, it is not perfect. The programming model is hard to use and read, creating a steep learning curve. It also makes L^AT_EX seem complex, scaring away a lot of potential users. A way to counter this complex look, is LyX [15]. This is a GUI around L^AT_EX offering some of the most common WYSIWYG features, while still utilizing the strength of L^AT_EX. 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, TeX makes it easy to create syntax errors. When these are made TeX 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. For example, if you want to use higher-order programming in TeX, you have to make sure that the function that is passed is not expanded immediately.

Next to this, we also have to watch out for name clashes, because in TeX, and L^AT_EX, most of the macro definitions end up in the root namespace. 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, L^AT_EX 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 [8]).

```
1 Below you can find an HTML table
2 <table border="1">
3   <tr>
4     <td>
5       Element 1
6     </td>
7     <td>
8       Element 2
9     </td>
10  </tr>
11 </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 [28], 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.

```
1 // This is a Neio document
2 [Document]
3
4 /* Below we define a chapter
5  * and a paragraph.
6  */
7 # Chapter 1
8 This is the first paragraph.
```

1 Chapter 1

This is the first paragraph.

Figure 2.1: The rendered form of the document to the left.

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 L^AT_EX, 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 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. To introduce code files, we take a look at the **Paragraph** class that is used to represent the paragraph in Listing 2.1.

Listing 2.2: Paragraph.no

```
1 namespace neio.stdlib;
2
3 import neio.lang.*;
4
5 /**
6  * Represents a paragraph
7  */
8 class Paragraph extends Content;
9
10 // private
```

```

11 Text text;
12
13 /**
14  * Initialises 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.

```

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

```

```

1 new Document()
2   .#("Chapter 1")
3   .text("This is the first paragraph
4         .");

```

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.

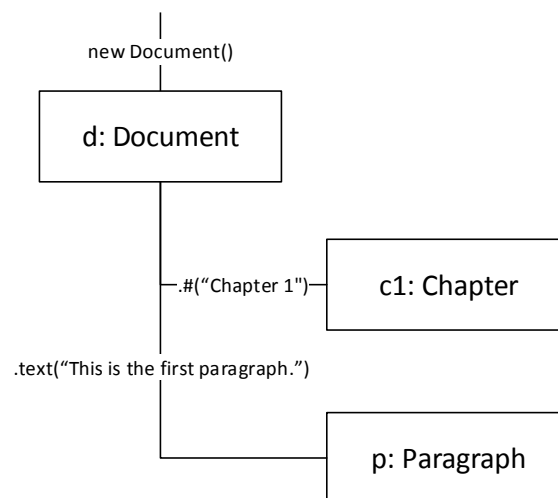


Figure 2.2: The object model of the previous document.

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 document is a **Content**. **Content** is the base class of Neio, it is like **Object** in Java.

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 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 be 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   .#()
```

A symbol method can only be used at the beginning of a line or after another symbol method. The latter is used to chain methods together. We discuss applications of this in Chapter 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. **text** 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.

We use `Text` instead of just a Java `String` 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 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

```
1  /**
2  * Creates a new {@code BoldText} and appends it to this
3  *
4  * @param t The text to make bold
5  * @return The newly created BoldText
6  */
7  public surround Text *(Text t) {
8      return appendText(new BoldText(t));
9  }
10
11 /**
12 * Creates a new {@code ItalicText} and appends it to this
13 *
14 * @param t The text to make italic
15 * @return The newly created ItalicText
16 */
17 public surround Text _(Text t) {
18     return appendText(new ItalicText(t));
19 }
20
21 /**
22 * Creates a new {@code MonospaceText} and appends it to this
23 *
24 * @param t The text to show in a monospace font
25 * @return The newly created MonospaceText
26 */
27 public surround Text `(Text t) {
28     return appendText(new MonospaceText(t));
29 }
```

The text `This is *bold* text` is thus transformed into the following call chain `text("This is ").text(*(new Text("bold"))).text(" text")`. The `text` call, wrapping the `*` call,

is necessary because the `*` method only creates a `BoldText`. It does not append it to the current 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.
```

1 *Chapter 1*

This **Chapter** is written in italic and monospace font.

2 *Chapter 2*

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.

2.4.4 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

```
1  /**
2   * Creates a new Chapter and adds it to this.
3   * If the level of the new Chapter is lower or equal
4   * to the level of this, the new Chapter is added to
5   * the nearest TextContainer above this.
6   * This is a nested call thus only ##+ will match.
7   *
8   * @param title The title of the new Chapter
9   * @param level The level of nesting of the new Chapter
10  */
11  nested Chapter #(Text title, Integer level) {
12      if (level <= this.level) {
13          Chapter c = nearestAncestor(Chapter.class);
14          if (c != null) {
15              return c.hash(title, level);
16          } else {
17              return nearestAncestor(Document.class).hash(title);
18          }
19      }
20      Chapter chapter = new Chapter(title, level);
21      addContent(chapter);
22
23      return chapter;
24  }
```

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

```
1  /**
2   * Creates a new Itemize and adds an ItemizeItem to it
3   * The Itemize is added to this.
4   *
5   * @param text The text to use for the ItemizeItem
6   * @return The created ItemizeItem
7   */
```

```

8 Itemize *(Text text) {
9     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.

2.4.5 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) [38]:

```

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

```

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

```

1 [Document]
2 The first line
3 of the first paragraph.
4
5 The first line
6 of the second paragraph.

```

```

1 new Document()
2   .newline()
3   .text("The first line")
4   .newline()
5   .text("of the first paragraph.")
6   .newline()
7   .newline()
8   .text("The first line")
9   .newline()
10  .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`

```

1 /**
2  * Handles newlines
3  *
4  * @return Returns a new NLHandler
5  */
6 NLHandler newline() {
7     return new NLHandler(this);
8 }

```

`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`

```

1 /**
2  * Creates a Paragraph and adds it to the parent.
3  *
4  * @param text The text for the Paragraph
5  * @return      The newly created Paragraph
6  */
7 Paragraph text(Text text) {
8     new Paragraph(text) par;
9     parent().addContent(par);
10    return par;
11 }
12
13 /**
14  * Returns this
15  *
16  * @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
4   *   TextContainer parent
5   *
6   * @return The NLHandler of the TextContainer that is parent of the existing
7   *   Paragraph
8   */
9  NLHandler newline() {
10     return (TextContainer) (parent().parent()).newline();
11 }

```

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 [7]. 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.

1		Student club		Rounds		Seconds/Round		Dist (km)		Speed km/h	
2	-----										
3		HILOK		1030		42		298,70		24,89	
4		VTK		1028		42		298.12		24.84	
5		VLK		841		51		243.89		20.32	
6		Wetenschappen and VLAK		819		53		237,51		19.79	
7		VGK		810		53		234.90		19.58	
8		Hermes and LILA		793		54		229.97		19.16	
9		HK		771		56		223.59		18.63	
10		VRG		764		57		221.56		18.46	
11		VEK		757		57		219.53		18.29	
12		VPPK		689		63		199.81		16.65	
13		SK		647		67		187.63		15.64	
14		Zeus WPI		567		76		164.43		13.70	
15		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 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.

2.5 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.

A method call in text mode is not always called on the previously returned object. It can be called also be called on an another object which was defined earlier. Context types 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.

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

```
1 new Document()
2   .newline()
3   .#("Chapter 1")
4   .newline()
5   .text("This is the first paragraph
6         .")
7   .newline()
8   .#("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 call chain can be represented as a tree. This is shown below.

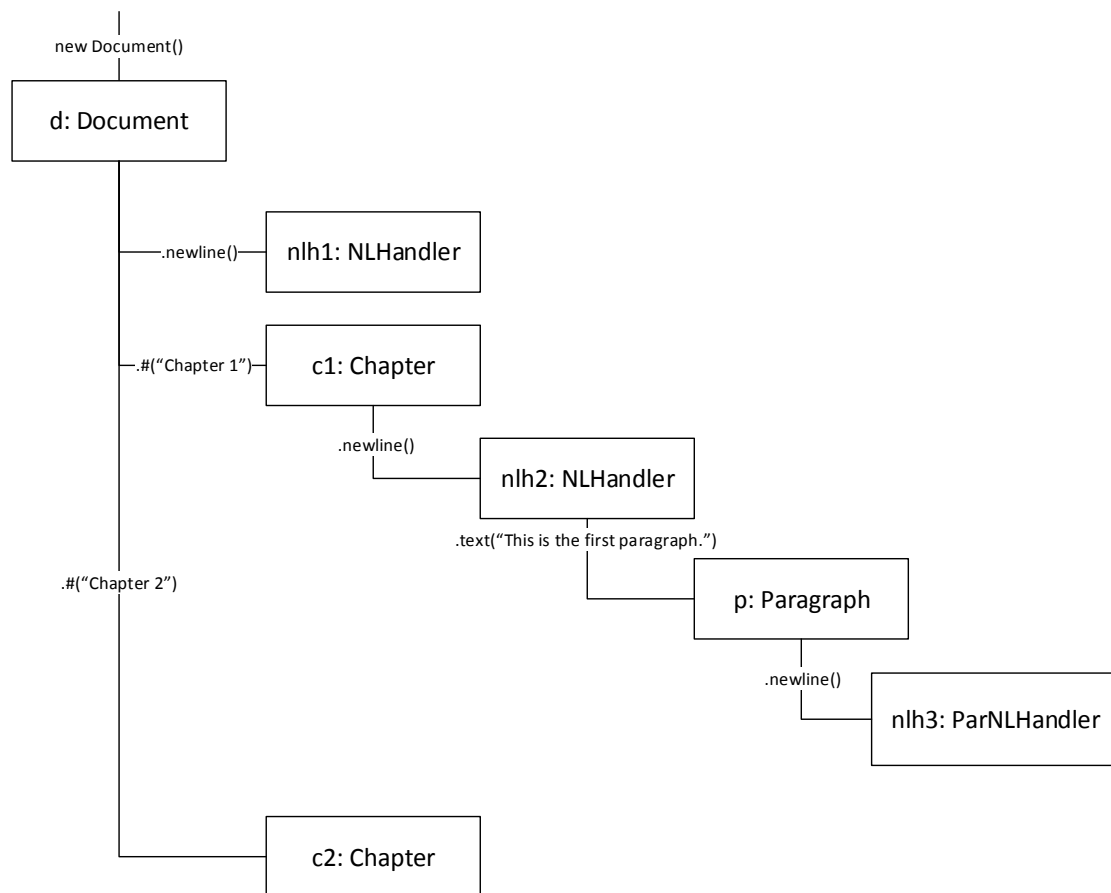


Figure 2.4: The tree representation of the call chain of the previous example.

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

If the previous context type defines the method of your method call, but the actual type does not, then we search for the method in the context.

To understand how context types find the object on which a method is called, we go over the previous example step-by-step .

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.

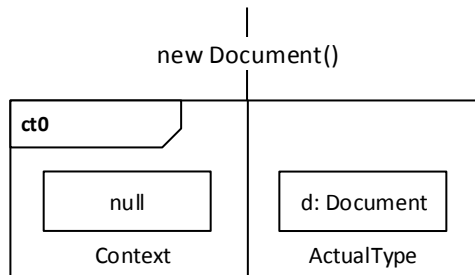
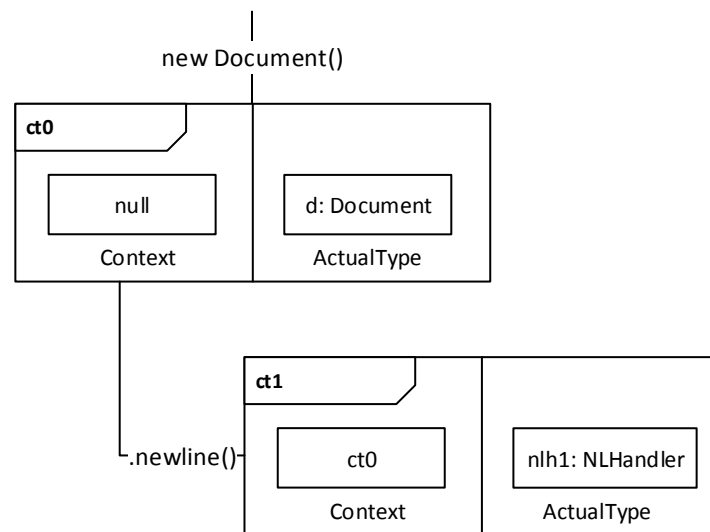


Figure 2.5: The first context type: `ct0`.

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 tree representation of the call chain, with context types, now looks as follows.

Figure 2.6: The context types `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 tree representation is shown below.

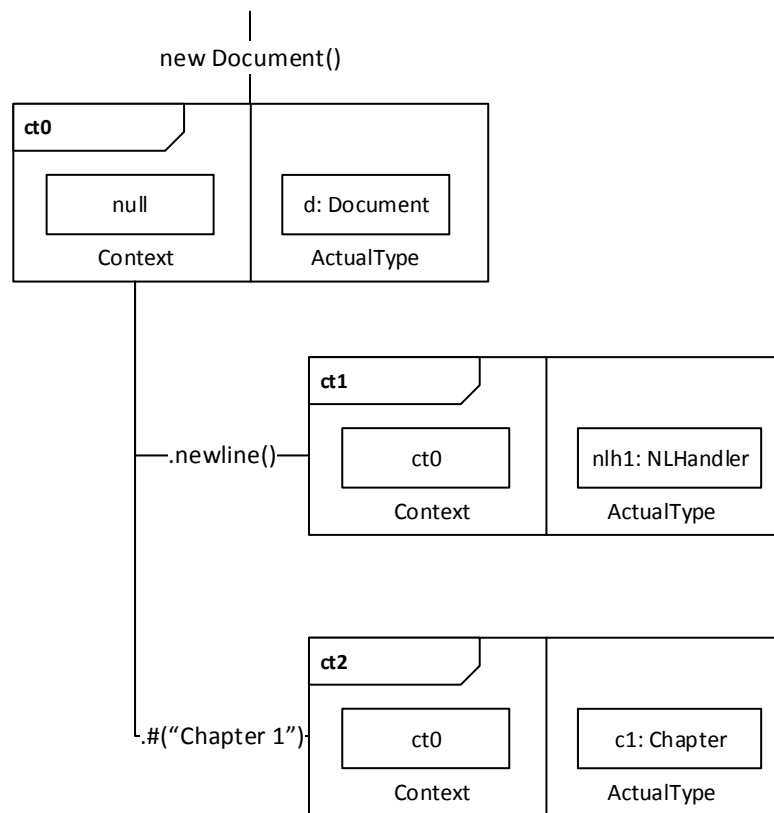


Figure 2.7: The context types ct0 through ct2.

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 tree representation of `ct0` through `ct5` is shown below.

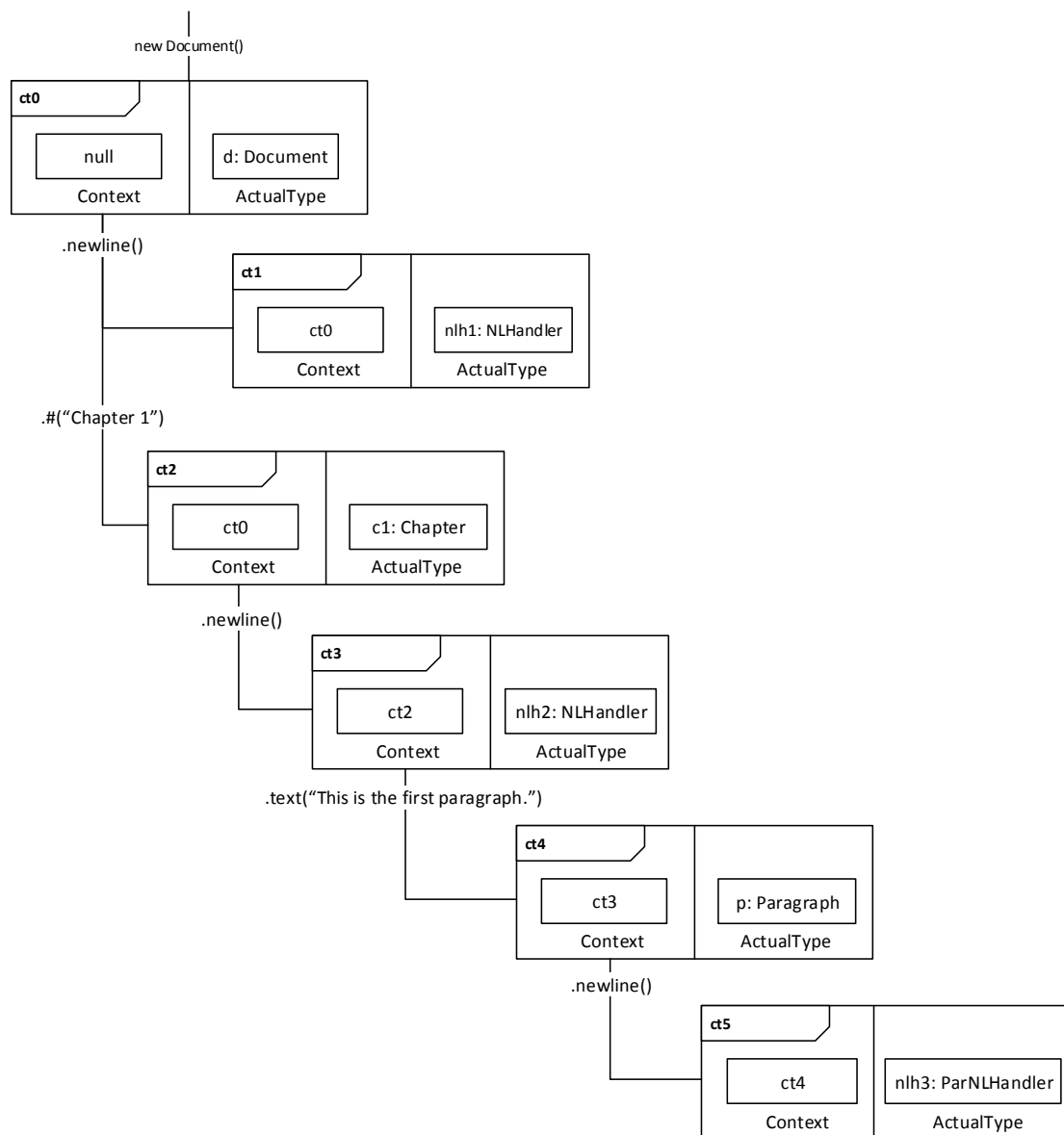


Figure 2.8: The context types ct0 through ct5.

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 `#`, only `Document` has such a definition. The last context type is thus built with `ct0` as context and `Chapter` as a actual type. The final tree representation is shown below.

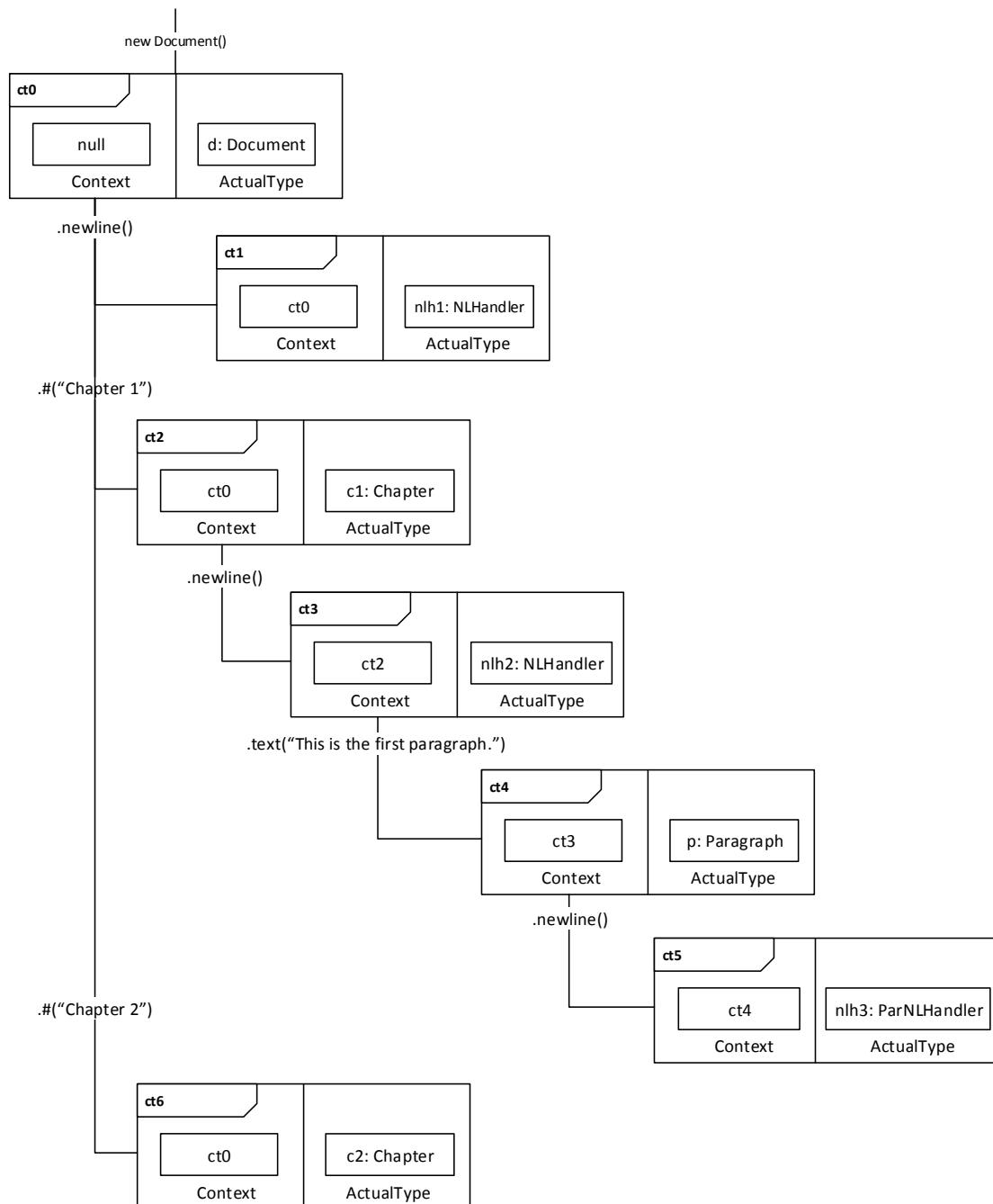


Figure 2.9: The context types ct0 through ct6.

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, a `context reset` happened as part of the lookup

procedure. 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.

```
1 [Document]
2 * Item 1
3 * Item 2
4
5 Paragraph 1
6 * Item 3
```

```
1 new Document()
2   .*( "Item 1" )
3   .*( "Item 2" )
4   .newline()
5   .text("Paragraph 1")
6   .*( "Item 3" )
```

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

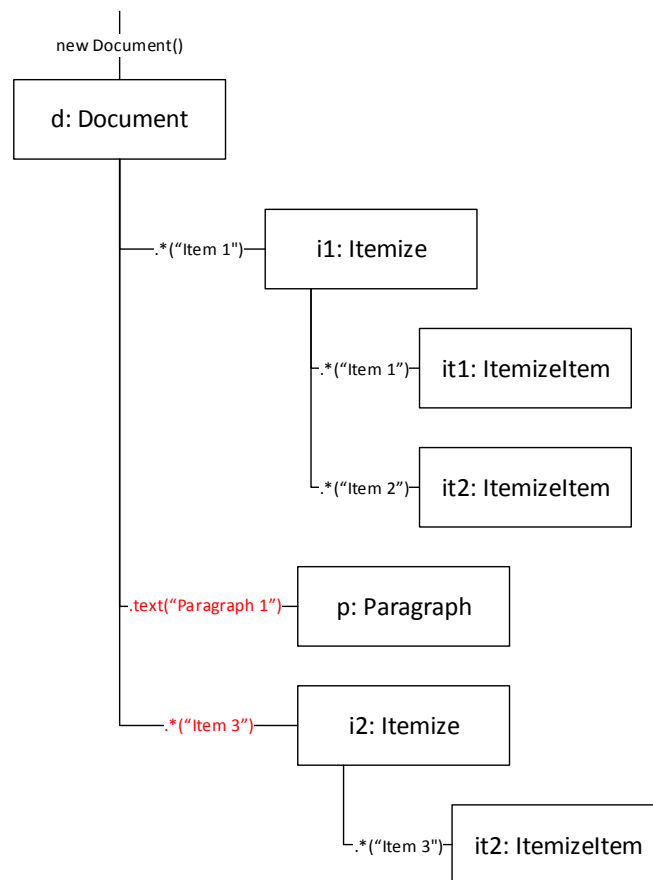
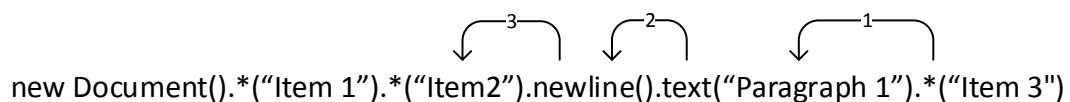


Figure 2.10: The object model of the previous example.

The `*` is defined in `Document` and creates `Itemize` `i1`. The `*` defined in `Itemize` creates `ItemizeItems` and adds them to the `Itemize`. 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.

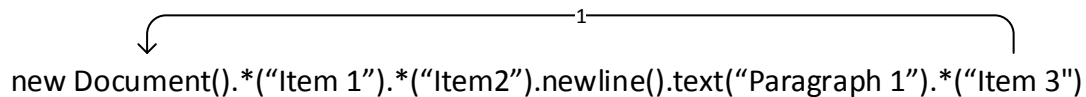


Figure 2.12: Wrong context lookup for `*`.

Finally, context types are only used for text mode, or when `this` is used in code mode.

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.

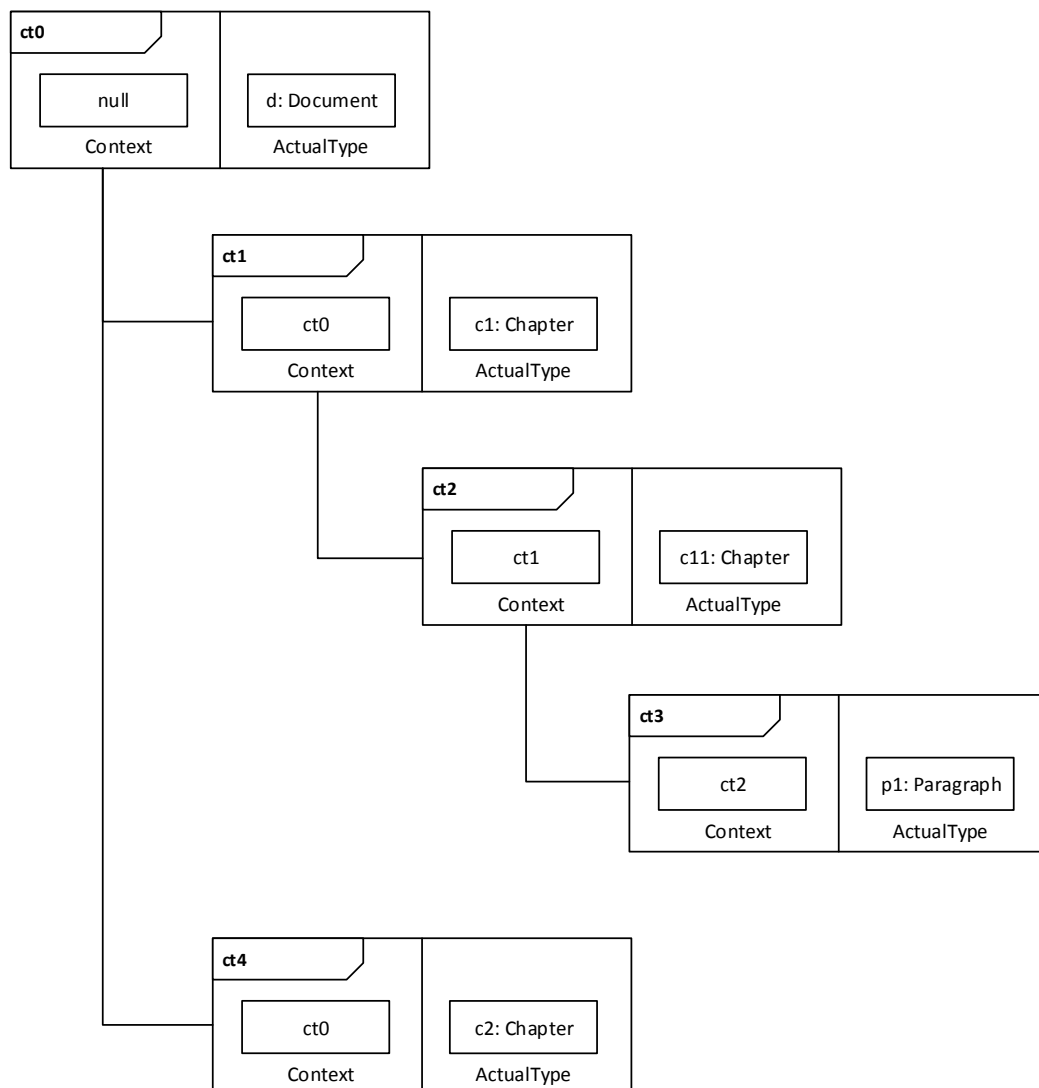


Figure 2.13: The correct context type representation.

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.

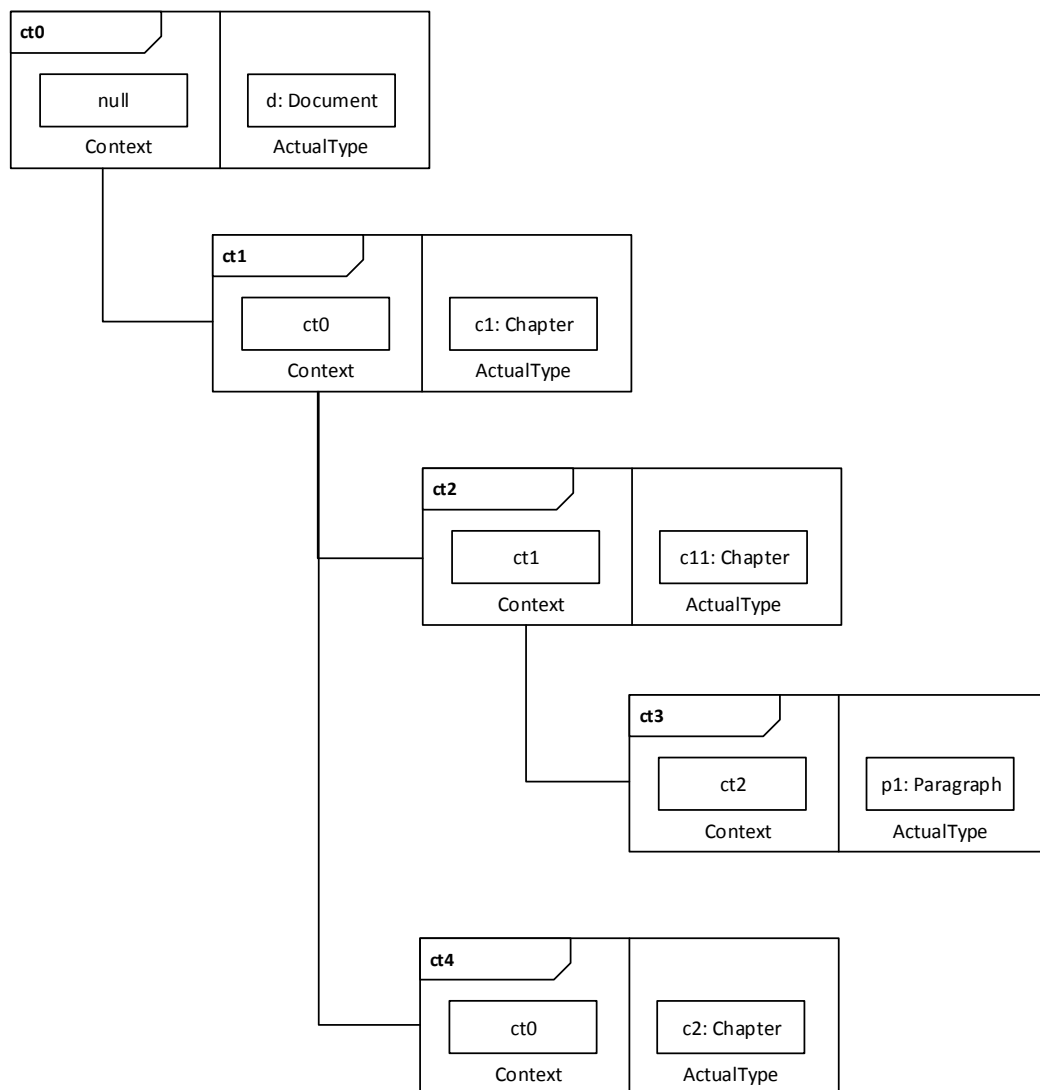


Figure 2.14: The wrong context type representation.

2.6 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 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 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.

2.6.1 Non-scoped code

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

```
1 [Document]
2 # Chapter 1
3 {
4     Chapter chapter2 = new Chapter("
5         Chapter 2")
6 }
```

```
1 new Document()
2     .#("Chapter 1");
3 Chapter chapter2 = new Chapter("Chapter
4     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:

```
1 nl ::= "\r?\n"
2 non-scopedCodeBlock ::= "{" nl (statement ";" nl)* nl* statement nl* ";"? nl*
3     "}" "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 are 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.

```

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

Listing 2.17: Manual return from code block.

```

1 [Document]
2 # Chapter 1
3 {
4     Chapter chapter2 = new Chapter("
5         Chapter 2");
6     return chapter2;
7 }
```

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.

2.6.2 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.

```

1 new Document()
2     .#("Chapter 1");
3 Chapter chapter 2 = new Chapter("Chapter 2")
4 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:

```

1 Document $var0 = new Document();
2 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 `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 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");
```

2.6.3 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 an example of how images are typically included is shown below.

2.6.4 Inline code

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 block to create a template for a letter.

Listing 2.18: template.no

```
1 [Document]
2 {
3     Text addressee = "Thomas Vanhaskel
4         ";
5     Text help = "my math class";
6     Text helpSubject = "math test";
7     Text closings = "Kind regards,";
8     Text name = "Titouan Vervack";
9 }
10 Dear {addressee},
11
12 Thank you for helping me out with {help
13     }.
14 I was able to do great at the {
15     helpSubject} thanks to you.
16 {closings}
17
18 {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`.

2.7 Text in code mode

In the previous sections we created objects such as `Chapter` 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 double quotes (`"`).

A Java `String` is 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 2.6.4, 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

```
1 [Document]
2
3 We roll the dice!
4
5 {
6     Random rng = new Random();
7     Integer limit = 12;
8     Integer random = rng.nextInt() %
9         limit;
10    if(random > (limit / 2)) {
11        "Ouch, we lost {random - (limit
12        / 2)}!"
13    } else {
14        if (random == (limit / 2)) {
15            "Oof, we broke even!"
16        } else {
17            "Great! We won {(limit / 2)
18            - random}!"
19        }
20    }
21 }
```

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 an 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.

2.8 Navigating through the lexical structure

We have discussed all the features of Neio and we know how the object models are built. 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.

```

1 [Document]
2 # Chapter 1
3 ## Chapter 2
4 # Chapter 2
5
6 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

```

1     return list;
2 }
3
4 /**
5  * Recursively checks for a parent of type {@code c}
6  *
7  * @param c Type of the demanded Content
8  * @return The first lexical parent of type {@code c}
9  */
10 <T> T nearestAncestor(Class<T> c) {
11     Content p = parent();
12     while ((p != null) && (!c.isInstance(p))) {
13         p = p.parent();
14     }
15     return list;
16 }
17
18 /**
19  * Returns all direct descendants (the ones right beneath this) of
20  * class {@code k}
21  *
22  * @param k The class of the descendants we are looking for
23  * @return A list of direct descendants of class {@code k}
24  */
25 <T extends Content> List<T> directDescendants(Class<T> k) {
26     List<T> descendants = new ArrayList<T>();
27     for (int i = 0; i < contentSize(); i = i + 1) {
28         Content c = content(i);
29         if (k.isInstance(c)) {
30             descendants.add((T) c);
31         }
32     }
33 }

```

2.9 Considerations

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

2.9.1 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.

2.9.2 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 more detail about what document types can be handled by Neio and how to specifically build these kind of documents. We also show the flexibility of Neio by discussing some libraries that have been developed.

3.1 Document classes

Before we discuss the developed document classes it is important to note that there is still a lot more work has to be done on them. If we just have a look at the user guides of Memoir [19] or KOMA-Script [13], two popular document classes in L^AT_EX, 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.

3.1.1 Document class definition

The document class to be used in a Neio document is defined by placing `[DocumentClass]` at the top of the file. With this line you are specifying what the first object in a Neio document

has to be. This object is the root of the call chain.

However, a document class definition is actually an expression that returns an object of this type. For example, when you write `[Document]`, what you get as root of the call chain might be a `Document`. It might also be an instance of the `Thesis` document class, discussed in the next section, as `Thesis` is a subclass of `Document`.

The reason that this could be also be a subclass is because documents can also be included in another document. To include a document we can pass the main document to the included document. The included document then uses the main document as root of its call chain.

Because of this it is good practice to choose the most generic document class that is available for your document.

3.1.2 Document

The simplest of documents can be created using the `Document` document class. It provides the syntax to create anything that Markdown can create, and on top of that it allows you to create tables, \LaTeX math, UML,... .

Like every Neio document, it can also use the code blocks defined in Section 2.6 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.

Listing 3.1: Content.no

```

1  /**
2   * Sets {@code c} right of this Content and sets the parent of
3   * {@code c} if we have a parent or sets this parent if {@code c}
4   * has a parent. One of the Contents should have a parent!
5   *
6   * @param c The Content to our right
7   * @return The Content to our right
8   */
9  <T extends Content> T leftOf(T c) {
10     if (c == null) {
11         return c;
12     }
13     Minipage mp = prepareParent(this, c);
14     Integer siblings = ((Minipage) (mp.content(0))).siblings();

```

```

15     Minipage mp2 = new Minipage(siblings);
16     mp2.addContent(c);
17     mp.addContent(mp2);
18
19     return c;
20 }
21
22 /**
23  * Sets {@code c} right of this Content and sets the parent of
24  * {@code c} if we have a parent or sets this parent if {@code c}
25  * has a parent. One of the Contents should have a parent!
26  *
27  * @param c The Content to our left
28  * @return The Content to our left
29  */
30 <T extends Content> T rightOf(T c) {
31     c.leftOf(this);
32     return c;
33 }
34
35
36 /**
37  * Prepares the parent of {@code left} or {@code right} to have
38  * horizontally aligned children (by using Minipage).
39  *
40  * @param left The left element whose parent has to be prepared
41  * @param right The right element whose parent has to be prepared
42  * @return The root Minipage
43  */
44 private Minipage prepareParent(Content left, Content right) {
45     Content child = null;
46     if (left.parent() == null) {
47         child = right;
48     } else {
49         child = left;
50     }
51
52     if (Minipage.class.isInstance(child.parent())) {
53         List<Minipage> descendants = child.parent().directDescendants(Minipage
54             .class);
55         for (int i = 0; i < descendants.size(); i = i + 1) {
56             Minipage m = descendants.get(i);
57             m.incrementSiblings();
58         }
59         return (Minipage) (child.parent().parent());
60     }
61
62     Minipage rootPage = new Minipage(0);
63     child.replaceWith(rootPage);
64
65     Minipage mpl = new Minipage(1);
66     mpl.addContent(left);
67     rootPage.addContent(mpl);
68
69     return rootPage;
70 }

```

This code takes the parent of one of the two `Contents`, the base, and links the other `Content` to the base. The implementation creates a root `LATEX Minipage` [21] if none of these object

is already next to another object. 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 $(1/\text{children})$ of the width.

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

The `Document` document class can thus be used to write simple documents, such as small reports. Another document class that is complexer and requires some configuration, is shown below.

3.1.3 Book

To write a book, such as this thesis, we need a more complex document class. To write this thesis we created a new document class that represents the `LATEX` template for a thesis.

In this case the template is implemented in the `Thesis` class given below. To reuse as much functionality as possible, it extends the `Document` class. It adds its own packages and required elements, such as a title page, and it creates its own `LATEX` preamble. The code for this is shown below, the preamble is not shown entirely as it is very long.

Listing 3.2: Thesis.no

```
1  /**
2   * Initialises the document class, adding the required packages and adding a
3   * ToC
4   */
5  Thesis() {
6      addPackage("graphicx")
7      .add("fancyhdr")
8      .add("amsmath")
9      .add("float")
10     .add("listings")
11     .add("tocloft")
12     .add("color")
13     .add("pdfpages")
14     .add("hyperref");
15
16     List<String> options = new ArrayList<String>();
17     options.add("english");
```



```

17     addPackage(options, "babel");
18
19     addClassMapping(Chapter.class, ThesisChapter.class);
20     addContent(new Pdf("TitlePage"));
21     addContent(new BlankPage());
22     addContent(new Pdf("TitlePage"));
23     addContent(new FrontMatter());
24 }
25
26 /**
27  * Builds the preamble for this document class
28  *
29  * @return The preamble for this document class
30  */
31 String preamble() {
32     StringBuilder preamble = new StringBuilder("\\documentclass[11pt,a4paper,
33         onside,notitlepage]{book}\\n")
34     .append("\\setlength{\\topmargin}{2cm}\\n")
35     .append("\\setlength{\\headheight}{14.49998pt}\\n")
36     .append("\\setlength{\\headsep}{1cm}\\n")
37     .append("\\setlength{\\oddsidemargin}{2.5cm}\\n")
38     .append("\\setlength{\\evensidemargin}{2.5cm}\\n")
39     .append("\\setlength{\\textwidth}{16cm}\\n")
40     .append("\\setlength{\\textheight}{23.3cm}\\n")
41     .append("\\setlength{\\footskip}{1.5cm}\\n")
42     .append(packages().toTex())
43     .append(packages().toTex());
44 }

```

In the above code we also tell the document class to not create **Chapters**, but instead create **ThesisChapters**. A **ThesisChapter** redefines a chapter such that a level one chapter is translated to a \LaTeX chapter instead of a \LaTeX section. This is done through the following line.

```

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

```

The document class also provides a method that initialises the main matter, which contains a Table of Contents (ToC) using the following line. This is a method that has to be called in the Neio document when the main matter begins. It is a separate method to allow us to easily add content, such as an abstract, before the main matter.

3.1.4 Abstract

The abstract used in this thesis is written in a separate Neio document (**overview.no**). It uses the **Abstract** document class and is included in this document by the following line.

Listing 3.3: thesis.no

```
1 overview.createDocument(parent().addContent(new Abstract()));
```

The first part of the Neio document for the abstract is shown below.

Listing 3.4: overview.no

```
1 [Abstract]
2 | The design and implementation of a userfriendly object-oriented markup
  | language
3 | Titouan VERVACK
4 | Science in Computer Science Engineering
5 * Prof. dr. ir. Marko van Dooren
6 - Dr. ir. Benoit Desouter
7 | Engineering and Architecture
8 | Ghent University
9 | Applied mathematics, computer science and statistics
10 | Prof. dr. Willy Govaerts
11
12 # Abstract
13 A lot of solutions exist for document creation.
```

As shown in the code above, the parameters for the **Abstract** are passed through symbol methods. To do this, **Abstract** overrides the **newline** method such that it creates an **AbstractNLHandler**. The **AbstractNLHandler** defines the **|**, **-** and ***** methods. It also overrides the **newline** method to return the control of newlines back to the enclosing **Content**. The code for the **-** and **|** methods is shown below.

Listing 3.5: AbstractNLHandler.no

```
1 /**
2  * Adds a counsellor to the Abstract
3  *
4  * @param text The counsellor to add
5  * @return The Abstract that created this NLHandler
6  */
7 Abstract -(Text text) {
8     parent().counsellor(text.realText());
9     return parent();
10 }
11
12 /**
13  * Fills in a new parameter depending on what has been filled in already
14  *
15  * @param text The value of the parameter
16  * @return The Abstract that created this NLHandler
17  */
18 Abstract |(Text text) {
19     Abstract a = parent();
```

```
20     if (a.title() == null) {
21         a.title(text.realText());
22     } else if (a.author() == null) {
23         a.author(text.realText());
24     } else if (a.program() == null) {
25         a.program(text.realText());
26     } else if (a.faculty() == null) {
27         a.faculty(text.realText());
28     } else if (a.university() == null) {
29         a.university(text.realText());
30     } else if (a.department() == null) {
31         a.department(text.realText());
32     } else if (a.chairman() == null) {
33         a.chairman(text.realText());
34     }
35
36     return parent();
37 }
```

The implementation of `|` is not perfect and the way to handle such scenarios should be changed in the future.

We could also have set the parameters using a code block and calling setters defined in `Abstract`. However, this seemed like a more elegant way.

3.2 Libraries

To demonstrate what the strength and flexibility of Neio 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 it would require a lot more effort to reach a state of completion. For example, if we have a look at the manual of a popular library such as TikZ[27], we see that more time is needed to write such a library than there is available for this thesis.

3.2.1 BibTeX

The library created for BibTeX is one of the easiest ones created using Neio. The implementation is actually a binding to L^AT_EX and does not allow for a lot of customisation as we do not have an object model of the BibTeX file.

The first reason we chose to implement BibTeX this way, is to illustrate that creating a binding for a L^AT_EX package is not difficult. The second reason is that we would have had to create an entire BibTeX parser to create a complete object model of the BibTeX files. As the BibTeX format is quite complicated we felt that our time was better spent elsewhere.

The important code for our implementation of BibTeX can be found below.

Listing 3.6: Bibtex.no

```

1  /**
2   * Cites something defined in this BibTeX
3   *
4   * @param cname The keyword of the citation
5   * @return A Citation object representing the citation of {@code cname}
6   */
7  Citation cite(String cname) {
8      return new Citation(cname);
9  }
10
11 /**
12  * Creates a TeX representation of this BibTeX, without citations
13  *
14  * @return The TeX representation of this
15  */
16  String toTex() {
17      StringBuilder result = new StringBuilder("\n\\bibliographystyle{plain}\n")
18          ;
19      result.append("\\bibliography{").append(name).append("}\n");
20      return result.toString();
21  }

```

The `toTex()` method creates the L^AT_EX representation of a `Content`. This method is further discussed in Chapter 4.

A BibTeX file can now be added through the `addBibtex` method and we can cite an entry by using the `cite` method. These methods are both implemented in `Document` and are shown below.

Listing 3.7: Document.no

```

1  /**
2   * Sets a BibTeX file for this Document
3   *
4   * @param The name of the BibTeX file
5   */
6  void addBibtex(String name) {
7      if (name != null) {
8          this.bibtex = new Bibtex(name);

```

```

9      }
10   }
11
12   /**
13    * A proxy method for {@code BibTeX}'s {@code cite}
14    * Returns a Cite that is linked to {@code key}
15    *
16    * @param key The key to access the citation
17    * @return The Cite corresponding to {@code key}
18    */
19   Citation cite(String key) {
20       if ((bibtex != null) && (key != null)) {
21           return bibtex.cite(key);
22       } else {
23           return new Citation("");
24       }
25   }

```

The `Citation` class is a subclass of `Text` as we call it through an inline code block. This is shown below.

```

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

```

`key` is a Java String as it is used to do a lookup on the BibTeX entries, it is thus not allowed to be marked up.

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 defined in `Referable`. An example of referring to a chapter is shown below.

```

1 [Document]
2 # Chapter 1
3 {Chapter chap = (Chapter) parent()}
4
5 In {chap.ref()} we explain references.

```

The example makes use of the `parent` method, defined in `NLHandler`, to put the newly defined `Chapter` in a method.

A `NLHandler` takes the object that created it as a parameter in its constructor so that it can refer to it later on. The `newline` method of a `Chapter` is defined in `TextContainer` (the super class of `Chapter`). The `parent` method in the example thus returns a `TextContainer`,

hence the cast to `Chapter`.

To reference an object, \LaTeX needs a unique label. We use the hashcode of an object for this as it is unique and easy to retrieve at any time. The `ref` method defined in `Chapter` and the `Referable` interface are shown below.

Listing 3.8: `Chapter.no`

```

1  /**
2   * Creates a reference to this Chapter
3   * and adds a correct prefix to it
4   * such as Chapter, Section or
5   * Subsection.
6   *
7   * @return A Reference to this Chapter
8   */
9  Reference ref() {
10     String prefix = "";
11     if (level < 2) {
12         prefix = "Chapter";
13     } else if (level == 2) {
14         prefix = "Section";
15     } else {
16         prefix = "Subsection";
17     }
18     return new Reference(prefix, "" +
        hashCode());
19 }
```

Listing 3.9: `Referable.no`

```

1  namespace neio.lang;
2
3  /**
4   * Declares that a class can be
5   * referred to
6   */
7  interface Referable;
8
9  /**
10   * References an object
11   *
12   * @return A reference to this object
13   */
14  Reference ref();
```

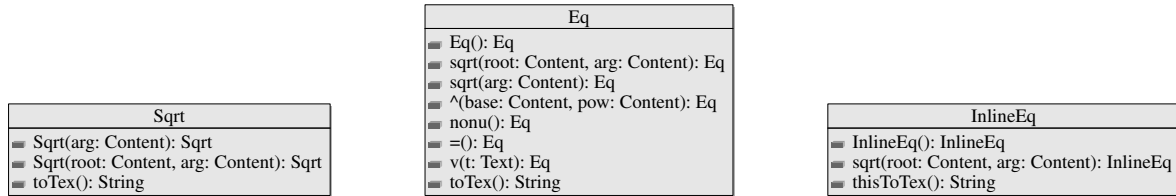
As we need a variable to reference something, this means that the object we want to refer to has to be available when we want to refer to it. This means that we can not refer to things that are defined later on in the project, which \LaTeX can do.

\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. To do this in Neio, we would have to hold off creating the reference until the document is completed.

3.2.3 \LaTeX math and amsmath

\LaTeX is very good at typesetting mathematical formulas, as such we could not forget about it.

Two ways to make use of the \LaTeX math and the `amsmath` package have been created. The UML below shows the most important classes in the Neio math library.



An inline math formula in \LaTeX is created by surrounding a formula with a pair of `'`s. In Neio we can create inline math using an inline code block. We can also wrap the code block with a `$` surround method, that does nothing, to make it look like inline math in \LaTeX . However, this does not offer any new functionality and it is thus not done in this example.

The only method that has been implemented for this class is `sqrt`, so we show how we can create an inline square root.

```

1 {
2     Integer base = 2;
3     Integer arg = 10;
4 }
5 The square root of {arg} is {ieq().sqrt("{base}", "{arg}")}.

```

The `ieq` method, defined in `Document`, creates an `InlineEq`. On this object we call the `sqrt` method. However, the `sqrt` method has `Text` arguments. To transform the `Integer` variables to `Text`, we enter text mode and open an inline code block that returns the variable. A solution to this problem is proposed in Chapter 5.3.

It is also possible to explicitly create an equation, in \LaTeX this is done using the `equation` environment. An example of how to create this in Neio is shown below.

```

1 {{
2     eq().nonu().sqrt("{base}", "{arg}")
3     ;
4     eq().^("{base}", "{arg}").=().v("{
5         Math.pow(base, arg)");
6 }}

```

$$2^{10} = 1024.0 \quad (1)$$

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

The code above creates two `Equations`. We disabled numbering on the first one through the `nonu` method and called `sqrt` on it.

In the second equation we use the `^` method to create an exponentiation and use the `=` method to create an equation. 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.

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 . In this section we explain how the table shown in Subsection 2.4.5 was built.

To create tables we created a DSL that is readable in text mode and that allows you to easily edit values in the table later on. The latter is important because customising table elements inline, as is done in \LaTeX , makes a table hard to read.

The table is created as an ASCII table, with pipes, spaces, dashes and values for the elements of the table. The example shown in Subsection 2.4.5 is repeated below.

Listing 3.10: An ASCII table created in Neio.

	Student club	Rounds	Seconds/Round	Dist (km)	Speed km/h
1	-----	-----	-----	-----	-----
2					
3	HILOK	1030	42	298,70	24,89
4	VTK	1028	42	298.12	24.84
5	VLK	841	51	243.89	20.32
6	Wetenschappen and VLAK	819	53	237,51	19.79
7	VGK	810	53	234.90	19.58
8	Hermes and LILA	793	54	229.97	19.16
9	HK	771	56	223.59	18.63
10	VRG	764	57	221.56	18.46
11	VEK	757	57	219.53	18.29
12	VPPK	689	63	199.81	16.65
13	SK	647	67	187.63	15.64
14	Zeus WPI	567	76	164.43	13.70
15	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

Note that the table above is created using the syntax in Listing 3.10 and is not an image that was included in the document. In fact, all of the examples shown from here on out, until the end of this chapter, are created using the DSLs demonstrated in said examples.

First of all, we note that it does not matter how many spaces are placed inside the elements of the table. The DSL for this table uses of the symbol method `|`, a nested method `-` and a new `NLHandler`.

The first `|` method is defined in `TextContainer`, so that it can be used in `Chapters` as well as `Documents` (they are both subclasses of `TextContainer`). It is shown below

Listing 3.11: `TextContainer.no`

```

1  /**
2   * Creates a new Table and adds a new TableRow to it.
3   * The Table is added to this.
4   *
5   * @param text The text to use in the first element of the TableRow
6   * @return The created TableRow
7   */
8  TableRow |(Text text) {
9      new TableRow(text) row;
10     new Table(row) table;
11     addContent(table);

```

```

12 |
13 |     return row;
14 | }

```

The `TableRow` that is returned from the `|` method is a row that is still under construction. By continuously calling `|` we are able to construct the entire row. This is the reason why we mentioned that symbol methods could be called after another symbol method in Subsection 2.4.1.

The `|` method that builds the `TableRow` is defined in `TableRow` and shown below. We also show the `|` method that has no arguments, it defines the end of a row.

Listing 3.12: `TableRow.no`

```

1 TableRow |(Text text) {
2     elements.add(new TableElement(text));
3     return this;
4 }
5
6 TableRow |() {
7     fte = new FinalTableElement();
8     return this;
9 }

```

The `newline` method defined in `TableRow` returns a `TableNLHandler`.

This handler defines the `-` nested method and the `|` symbol method. The former creates a horizontal line in the document and disregards the nesting level. This level could however be used to determine the width of the table for example. The latter is used to start the construction of a new `TableRow`.

The code for the `-` and `|` methods defined in `TableNLHandler` is shown below.

Listing 3.13: `TableNLHandler.no`

```

1 /**
2  * Creates a horizontal line and appends it to our parent
3  *
4  * @param level The level of this nested operator, it is ignored
5  * @return The appended Hline
6  */
7 nested Hline -(Integer level) {
8     return parent().appendHline();
9 }
10

```

```

11 /**
12  * Creates a new TableRow and appends it to our parent
13  *
14  * @param text The text to be used in the first element of the new TableRow
15  * @return The new TableRow under construction
16  */
17 TableRow |(Text text) {
18     return parent().appendRow(new TableRow(text));
19 }

```

Finally we can write an empty newline to return control of newlines to the enclosing `Content`

The libraries that are discussed in the next sections are domain specific and show how versatile Neio can be.

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 for example `TikZ`. We could also create it in a third party program and include an image of the instance in our document, this solution also works for Markdown.

However, since an algorithm is used, and we have a programming model, could not we just build the tree and then display it? Yes we can! All 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 below.

Listing 3.14: `rbtDocument.no`

```

1 [Document]
2 {
3     List<Integer> tree = new ArrayList<Integer>();
4     tree.add(33);
5     tree.add(15);
6     tree.add(10);
7     tree.add(5);
8     tree.add(20);
9     tree.add(18);
10    tree.add(47);
11    tree.add(38);
12    tree.add(36);
13    tree.add(39);
14
15    String numbers = '''' + tree.get(0);
16    for (int i = 1; i < tree.size(); i = i + 1) {

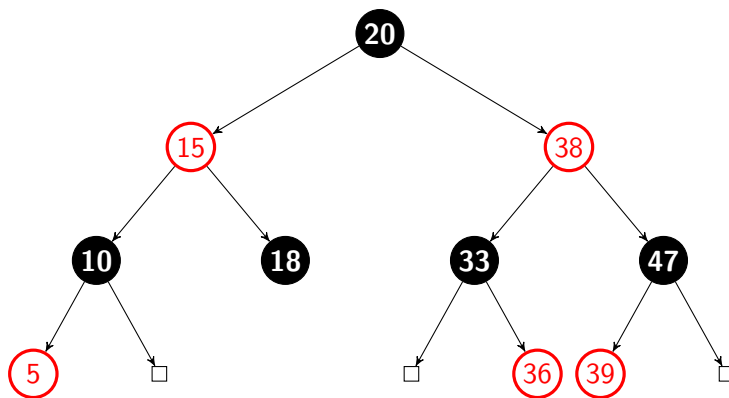
```

```

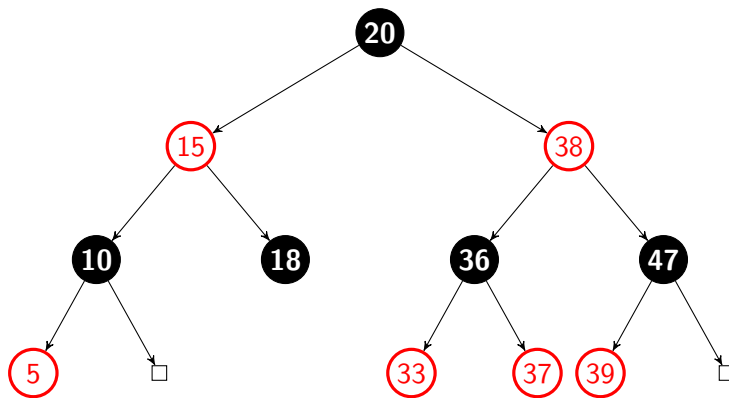
17     numbers = numbers + ',', '' + tree.get(i);
18   }
19 }
20
21 Given the red black tree of {numbers}
22 {
23     RedBlackTree rbt = new RedBlackTree().insert(tree);
24     return rbt;
25 }
26
27 Add 37
28 {
29     RedBlackTree rbt2 = rbt.insert(37);
30     return rbt2;
31 }
32
33 Add 51
34 {
35     RedBlackTree rbt3 = rbt2.insert(51);
36     return rbt3;
37 }

```

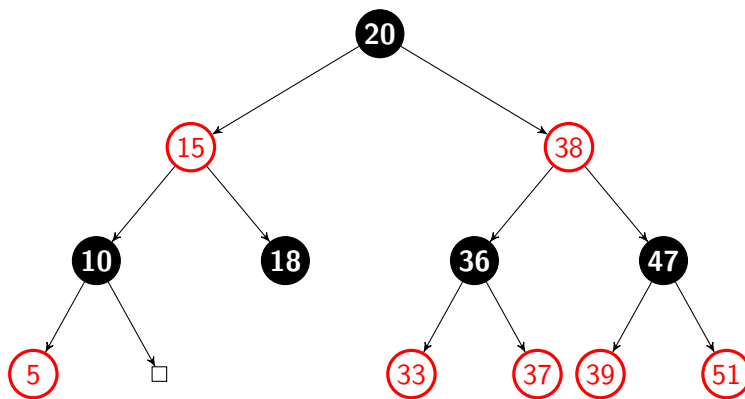
Given the red black tree of 33, 15, 10, 5, 20, 18, 47, 38, 36, 39



Add 37



Add 51



This code allows us to very easily create and show red black trees. Of course, we are not limited to red black trees. This can be done for any tree or graph as long as we have code for it and write a method that can display it.

To compare our method to typesetting every instance of the tree in \LaTeX , we show the \LaTeX code for the first instance of the tree. This code is not that easy to read, hard to write and prone to syntax errors. For example, this code is newline sensitive, placing empty newlines between the entries throws a syntax error at compilation. The code for the next instance of the tree is very similar, but it is hard to reuse this code.

Listing 3.15: rbtLatex.tex

```

1 \begin{tikzpicture}[->,>=stealth',level/.style={sibling distance = 5cm/#1,
   level distance = 1.5cm}]
2 \node [arn_n] {20}
3 child { node [arn_r] {15}
4 child { node [arn_n] {10}
5 child { node [arn_r] {5}
6 }
7 child { node [arn_x] {} }
8 }
9 child { node [arn_n] {18}
10 }
11 }
12 child { node [arn_r] {38}
13 child { node [arn_n] {33}
14 child { node [arn_x] {} }
15 child { node [arn_r] {36}
16 }
17 }
18 child { node [arn_n] {47}
19 child { node [arn_r] {39}
20 }
21 child { node [arn_x] {} }
22 }
23 }
24 ;

```

```
25 \end{tikzpicture}
```

Note that this tree is not implemented optimally, when we append a new value to the tree in our implementation, we completely clone the tree and add a new node to this new tree. 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 the three trees would be represented by the same object. They would thus all show the latest version of the tree. This problem is discussed in Chapter 5.

3.2.6 MetaUML

The next example has already been used in Subsection 3.2.3. Every UML diagram in this document is created using the UML library discussed in this section.

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 [20]. How this works exactly is shown in Chapter 4. For now it is enough to know that the object model build by this library is output as a string that is understood by MetaUML. MetaUML then creates a MetaPost file [39] which is included in the document.

In this section a high-level overview of how we create UML diagrams is given. We start out with an example.

```
1 [Document]
2 {
3     new Uml(''./project.xml'', ''neio.stdlib.graph'')
4 }
```

RedBlackTree
<pre> ■ RedBlackTree(): RedBlackTree ■ RedBlackTree(root: RedBlackNode): RedBlackTree ■ root(): RedBlackNode ■ insert(items: List<Integer>): RedBlackTree ■ insert(value: Integer): RedBlackTree ■ insert(value: Integer, current: RedBlackNode): void ■ fixTree(node: RedBlackNode): void ■ fixRedParentUncle(node: RedBlackNode): void ■ fixRPBUR(node: RedBlackNode): void ■ fixRPBUL(node: RedBlackNode): void ■ rotateLeft(node: RedBlackNode): void ■ rotateRight(node: RedBlackNode): void ■ preTex(): void ■ toTex(): String </pre>

RedBlackNode
<pre> ■ RedBlackNode(color: Integer, value: Integer): RedBlackNode ■ setColor(color: Integer): void ■ setParent(parent: RedBlackNode): void ■ setRight(right: RedBlackNode): void ■ setLeft(left: RedBlackNode): void ■ cloneSelf(): RedBlackNode ■ grandparent(): RedBlackNode ■ uncle(): RedBlackNode ■ color(): Integer ■ value(): Integer ■ isLeaf(): Boolean ■ isRed(): Boolean ■ isBlack(): Boolean ■ left(): RedBlackNode ■ right(): RedBlackNode ■ parent(): RedBlackNode ■ hasLeft(): Boolean ■ hasRight(): Boolean ■ isRoot(): Boolean ■ preTex(): void ■ toTex(): String </pre>

This code reads the `project.xml` file, that tells it what project we want to create an UML diagram for. The second parameter is the namespace the UML diagram should be created for. This means that the UML diagram in this example is created from the **actual** classes that make up this document.

When the document is compiled, the `neio` code files are read and

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.

```

1 [Document]
2 {
3     List<String> list = new ArrayList<String>();
4     list.add(''neio.stdlib.uml.Uml'');
5     list.add(''neio.stdlib.uml.UmlClass'');
6     new Uml(''./project.xml'', ''neio.stdlib.uml'').scale(80).show(list).
       showAll();
7 }

```

Uml
<ul style="list-style-type: none"> ■ Uml(projectXml: String, ns: String): Uml ■ gatherClasses(): void ■ show(show: List<String>): Uml ■ showPrivate(): Uml ■ showProtected(): Uml ■ showAll(): Uml ■ scale(scale: Integer): Uml ■ preTex(): void ■ toTex(): String ■ toMetaUML(): String ■ id(o: Object): String

UmlClass
<ul style="list-style-type: none"> ■ UmlClass(type: Type, showPrivate: Boolean, showProtected: Boolean): UmlClass ■ gatherVars(): void ■ gatherMethods(): void ■ type(): Type ■ name(): String ■ toMetaUML(): String

In this example the UML diagram shows two of the **actual** classes that were used to create this UML diagram. The code to create these UML diagrams thus reads and displays itself. How this works exactly is explained in Chapter 4.

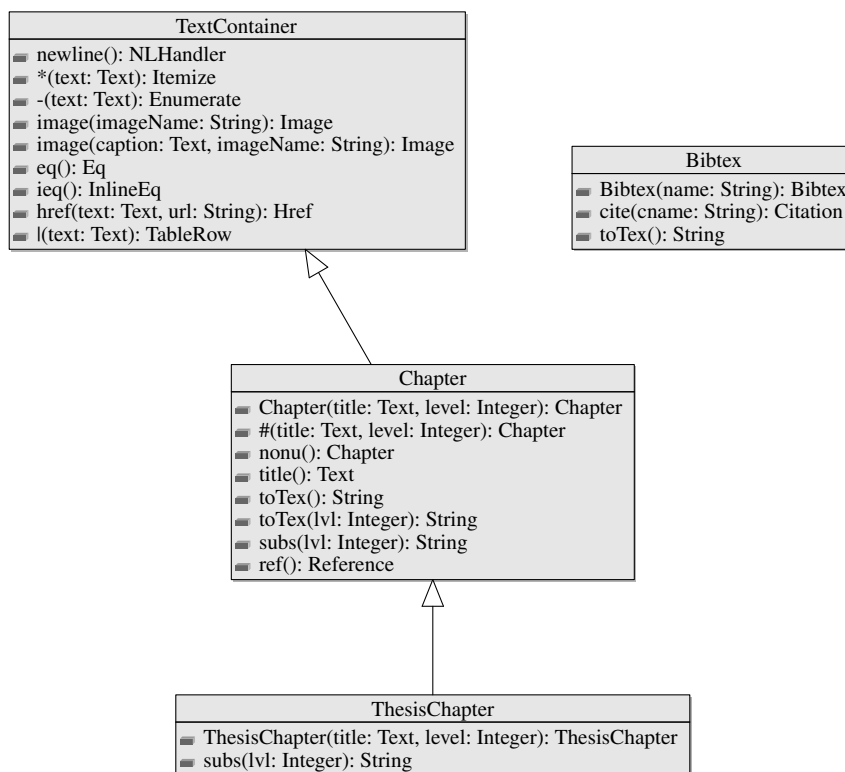
As the code for this library is over 600 lines long, we do not show all of the code.

The problem with the MetaUML tool is that positioning is not done automatically. The way we implemented positioning is 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 and also add the Bibtex class to the UML diagram, to show what happens to classes that have 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. These arrows are only shown for direct super classes. This is done to avoid clutter.

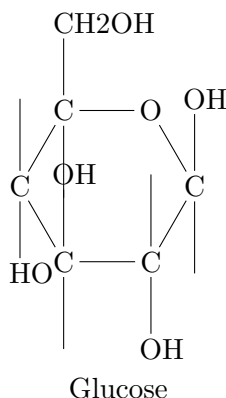
3.2.7 Chemfig

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

```

1 [Document]
2 {
3     new Structure()
4 }
5
6 $ C * 6 _ OH ^ - C _ ^ OH - O - C _ ^ CH2OH - C _ HO ^ - C _ ^ OH -
7 | Glucose
  
```

This example produces the following output



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 build by chaining atoms together.

Listing 3.16: StructureNLHandler.no

```

1 Atom $(Text text) {
2     Atom a = new Atom(text.realText());
3     a.setParent(parent());
4     parent().addAtom(a);
5     return a;
6 }
```

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. Finally, the `^` method defines 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 3.17: Atom.no

```

1 Atom =() {
2     return createNext("", "=");
3 }
4
5 private Atom createNext(String atomText, String link) {
6     Atom a = new Atom(atomText);
7     a.setParent(structure);
8     next = a.setLink(link);
9     return a;
10 }
```

The `toTex` method returns a \LaTeX representation of the structure that is understood by the \LaTeX package `chemfig` [5].

3.2.8 Lilypond

For the last example we create sheet music. An example musical score is shown below.

Listing 3.18: An example musical score.

```

1 [Document]
2 {
3     Score s = new Score().a().b().c(1).d(1).e(1).f(1)
4         .g(1).f(1).e(1).d(1).c(1).b().a()
5         .g().f().e().d().c().b(- 1).a(- 1);
6     return s;
7 }

```



```

10     return new Score(this, new Note(note, octave, this));
11 }
12 /**
13  * Adds the c node to a new Score
14  *
15  * @param octave The octave of this c
16  * @return A new Score containing a new c
17  */
18 Score c(Integer octave) {
19     return add(C, octave);
20 }

```

We can now query the notes of this score, for example `{s.get(8)}` gives us the eighth note. Doing this for our example tells us that the eighth node is e.

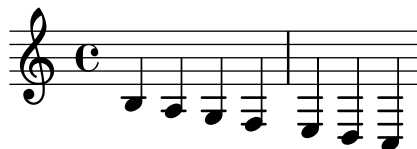
It is also possible to reverse a score or shift it up or down using the `reverse` and `shift` methods respectively. The sequence of notes can also be printed using the `print` method. These methods are demonstrated in the example below.



This score is read as c, d, e, f, g, a, b.



We can shift every note by one, doing so yields the following notes: d, e, f, g, a, b, c.



Reversing the score results in the following sequence of nodes: b, a, g, f, e, d, c

The code for this example as well as the implementation of the `reverse` and `shift` methods is given below.

Listing 3.20: Score.no

```

1  /**
2  * Copies this Score and shifts every
   node up or down by {@code shift}
3  *
4  * @param shift The amount to shift the
   nodes by
5  * @return A new Score where every node
   is shifted by {@code shift}
6  */
7  Score shift(Integer shift) {
8      Score s = new Score(this);
9      List<Note> n = s.notes();
10     for (int i = 0; i < n.size(); i = i
11         + 1) {
12         Note note = n.get(i);
13         note.shift(shift);
14     }
15     return s;
16 }
17
18 /**
19 * Creates a copy of this Score with
   the notes in reverse order
20 *
21 * @return A new Scope with the notes
   of this Score in reverse order
22 */
23 Score reverse() {
24     Score s = new Score(this);
25     Collections.reverse(s.notes());
26     return s;
27 }

```

Listing 3.21: The code of the example.

```

1  {
2      Score ss = new Score().c().d().e().
   f().g().a().b();
3      return ss;
4  }
5
6  This score is read as {ss.print()}.
7  {
8      Score sss = ss.shift(1);
9      return sss;
10 }
11
12 We can shift every note by one, doing
   so yields the following notes: {sss
   .print()}.
13 {
14     Score s4 = ss.reverse();
15     return s4;
16 }
17
18 Reversing the score results in the
   following sequence of nodes: {s4.
   print()}

```

To typeset these music sheets, the object model is written out in a format that is understood by LilyPond [30]. This is written to a file and the file is passed to the `lilypond` command line tool. This tool generates a PDF that is then inserted as an image in the document.

Chapter 4

Implementation

To use Neio, we also had to build a compiler. How we implemented it, as well as some lower level concepts used in Neio code files, is explained in this chapter.

4.1 Compile process

We start out by showing a diagram that depicts the process a source file goes through Neio source code up to the final \LaTeX file.

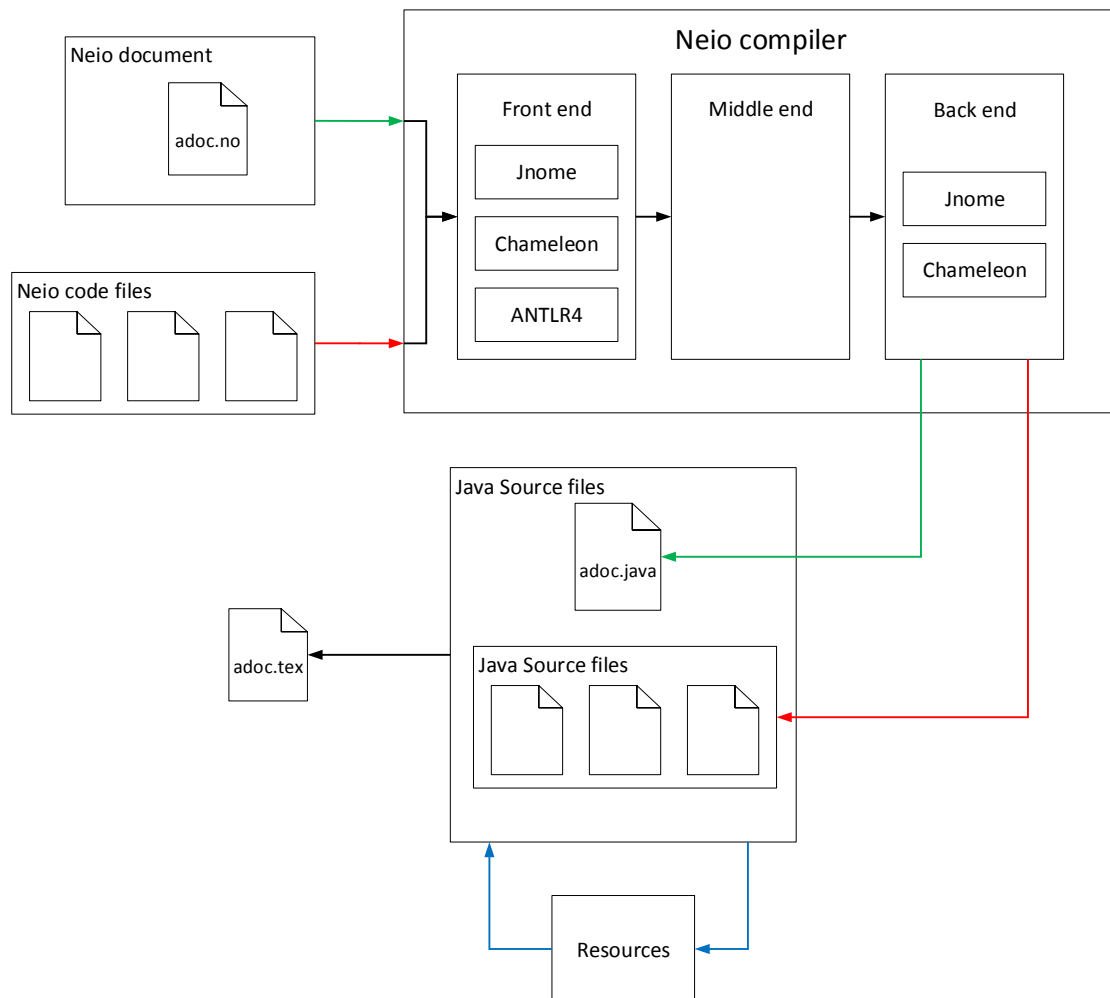


Figure 4.1: The process of compiling Neio source code to LaTeX.

First of, the Java and Neio library (the Neio code files) are read by the front end of the Neio compiler and translated into a Concrete Syntax Tree (CST). This CST is created by the parser and lexer that were generated by ANTLR4. The CST is then visited using the visitor pattern and an Abstract Syntax Tree (AST) is build.

The AST contains objects provided by Jnome and Chameleon. It is then passed to the middle end of the compiler and transformed to an object model that can be output to valid Java code. The transformed AST and the ASTs for the Neio library are passed to the compiler back end which generates the final Java code.

This generated Java code is then compiled and executed again. During the execution, resources might be created, if so, they can be used in the next step.

The last step uses the Neio library to create a \LaTeX file of the created object model.

4.2 Used libraries and frameworks

In the previous section we mentioned three libraries/frameworks that were used to successfully compile the document. They are discussed in more detail in this chapter.

4.2.1 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 code files.

We used ANTLR4 in the following way. First we created two grammars, consisting of a lexer and a parser, for the Neio documents and classes, thus four files in total. They are written using the ANTLR4 DSL. An example of the parser rule that describes the class definition and the lexer tokens that it uses are shown below.

Listing 4.1: ClassLexer.g4

```

1 ABSTRACT : 'abstract';
2 CLASS : 'class';
3 INTERFACE : 'interface';
4 EXTENDS : 'extends';
5 IMPLEMENTS : 'implements';
6 SCOLON : ',';
7 fragment LETTER : [a-zA-Z];
8 fragment CHAR : LETTER | DIGIT |
   UNDERSCORE | DOLLAR;
9 Identifier : CHAR+;
```

Listing 4.2: ClassParser.g4

```

1 classDef : ABSTRACT? header Identifier
   inheritance* SCOLON;
2 header : CLASS | INTERFACE;
3 inheritance : ( EXTENDS type
   | IMPLEMENTS type);
4
```

These are then fed to the `antlr4` command line tool. This tool generates Java classes using the grammars you defined. These classes are compiled and then used to visit the Concrete Syntax Tree of a file using the visitor pattern.

Whilst visiting every rule and terminal, we build an Abstract Syntax Tree (AST) for the given file. This tree can then be manipulated in later stages.

The objects that are used in the AST are objects defined in Jnome and Chameleon. These object, in combination with the ANTLR4 grammars and visitor classes, thus form the front end of our compiler.

4.2.2 Chameleon and Jnome

Chameleon and Jnome are both projects that were developed by professor van Dooren and released under the MIT license.

Chameleon [2] [37] is a framework for defining abstract ASTs of software languages. It enables the reuse of generic language constructs and the construction of language-independent development tools. It does so by providing objects for commonly used concepts in software languages. A few examples of such objects are the following: `Expression`, `VariableDeclaration`, `Type`, `MethodInvocation`, `InfixOperatorInvocation`,

Jnome [11] is a Chameleon module for Java 7. It extends the objects available in Chameleon with Java-specific objects. It is a front end for the Java language because it can read Java code and build an AST from it (using ANTLR3). It is also able to output Java code for a given an AST consisting of Jnome and Chameleon objects. This part is the back end of the Neio 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 `Strings`. It reads all of the source files and libraries that are specified in a file called `project.xml`. This was mentioned earlier in Subsection 3.2.6. The same file is used to read Neio projects in our compiler. The `project.xml` for the Neio compiler is shown below.

Listing 4.3: `project.xml`

```
1 <project name="Thesis">
2   <language name="neio" />
3   <baselibraries>
4     <library language="Neio" load="true" />
```

```
5     </baselibraries>
6     <sourcepath>
7         <source root="../../examples/0.8/lib" />
8     </sourcepath>
9     <binarypath>
10         <jar file="../../build/libs/chameleon.jar" />
11         <jar file="../../build/libs/jnome.jar" />
12         <jar file="../../build/libs/neio.jar" />
13     </binarypath>
14 </project>
```

In Subsection 3.2.6 the following is happening. First, the Neio compiler reads the Java library as well as the Neio standard library, Neio compiler, Chameleon and Jnome. Which libraries to read exactly is shown in `project.xml` as is shown in Listing 4.3. Then the Java code is generated and executed.

During this execution, the `Uml` class uses the code in the Neio compiler to repeat the process of reading these libraries. It can now create UML representation of anything it read in the previous step.

The libraries can be read using the following two lines.

```
1 NeioProjectBuilder projectBuilder = new NeioProjectBuilder();
2 JavaView view = (JavaView) (projectBuilder.build(projectXml));
```

The `projectBuilder` does some initialisation and loads the `project.xml`. The `view` contains all of the namespaces, classes,... that were read.

Chameleon also provides IDE support for the code mode (syntax errors and highlighting, project outline, dependency analysis,...) as is shown below.

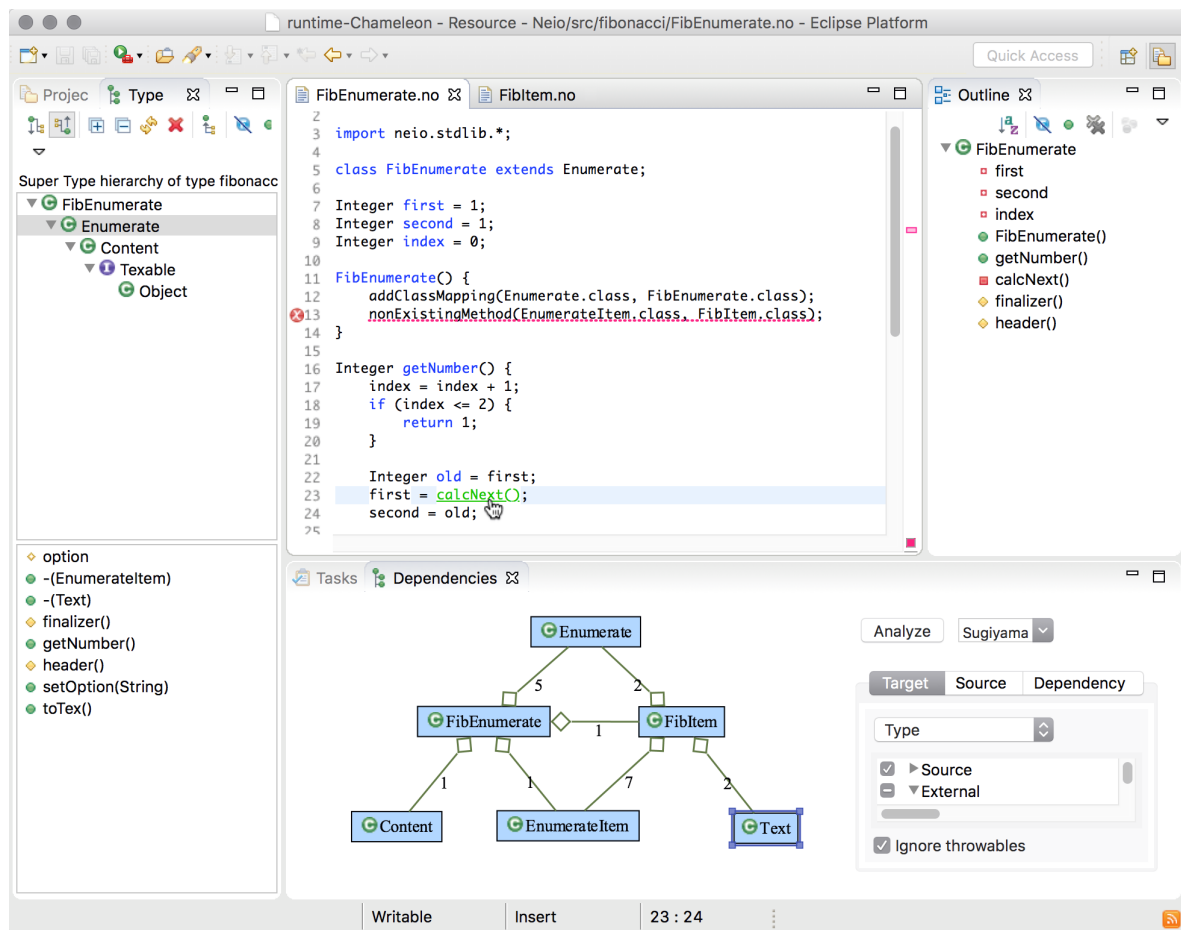


Figure 4.2: A code file with syntax highlighting, a syntax error and a dependency view.

This support is enabled through the few lines of code shown below.

```

1 public class Bootstrapper extends EclipseBootstrapper {
2     public Language createLanguage() throws ProjectException {
3         Neio result = new NeioLanguageFactory().create();
4         result.setPlugin(EclipseEditorExtension.class, new JavaEditorExtension
5             ());
6         result.setPlugin(DependencyOptionsFactory.class, new
7             JavaDependencyOptionsFactory());
8         return result;
9     }
10 }

```

Next to this code, we also have to add a few configuration files: `plugin.xml`, `build.properties` and `META-INF/MANIFEST.MF`. We also have to add an `xml/` directory which is used for syntax highlighting. These files can be reused for another language as well. This only requires us to fill in the name of the plugin and the extension of the language in these files.

Of course the compiler also has to fill in position information, but this is needed to create comprehensive error messages anyway.

Having obtained an AST in Subsection 4.2.1, we now manipulate it in the middle end of our compiler. The transformed AST is then output to Java using the Jnome back end.

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 popular in the computer science world [6], which is why we chose to base our code mode on it. This popularity is very important as it gives us a large user base that can read and understand code mode, but we are also able to reuse Java libraries. Because of said popularity, a lot of libraries already exist, which dramatically improves the possibilities of Neio. For example, it is possible to use a library that tests your network connection, and directly include the results in a Neio document. In TeX this is also possible, but because it is so hard to use the programming model and because there are only so little TeX developers, almost no libraries of this kind are available.

4.3.2 Fluent interface

We tried to make use of fluent interfaces as much as possible. In the design of the language we did this by using method chains, but we also tried to use it in the libraries we implemented in Section 3.2. We chose to do this as it imposes less boilerplate code (less variable declarations) and because it is clearer to read what we are building. Instead of passing a ton of arguments to one method, we split it up in multiple methods and build a single object.

4.3.3 Translation to Java

Neio makes use of context types, which is a new concept. This means we had to create a custom translation for it. The way we choose to translate this, 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` was implemented as a subclass of the `RegularJavaType` class in `Jnome`. A `RegularJavaType` is what we know as a Java class.

We also created a `NeioMethodInvocation` which creates a `ContextType` when its type is queried. The constructor of `ContextType` takes two `Types` as arguments. The first one is the actual return type of the method, the second one is the context of the the method. The `ContextType` then adds the two `Types` as super classes. so that it can access there methods.

When a lookup is done on a `ContextType` it first checks the actual type and then the context. Lookups are provided by Chameleon and explaining how exactly they work is out of the scope of this Thesis.

The lookup on a `NeioMethodInvocation` now returns the correct type, but we still have to find which variable corresponds to this type. To implement this we add every variable that is declared to a stack. To find the correct variable we iteratively pop the stack until a variable with the correct type is found.

Lastly, we also have to translate `this` as we saw in Subsection 2.6.2. All we have to do for this, is replace `this` with the last defined variable and go through the above process again.

The complete translation to Java is shown below.

Listing 4.4: testInput.no

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

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

Listing 4.5: testInput.java

```

1 package testInput;
2
3 import neio.fib.*;
4 import neio.io.*;
5 import neio.lang.*;
6 import neio.stdlib.*;
7 import neio.thesis.*;
8 import neio.stdlib.chem.*;
9 import neio.stdlib.graph.*;
10 import neio.stdlib.math.*;
11 import neio.stdlib.music.*;
12 import neio.stdlib.uml.*;
13 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         NLHandler $var1 = $var0.newline();
25         Chapter $var2 = input.hash(new Text("Chapter 1"));
26         NLHandler $var3 = $var2.newline();
27         Paragraph $var4 = $var3.text(new Text("This is the first paragraph.))
28         ;
29     }
30
31     public static void finishDocument(Document input) {
32         java.lang.String $var0 = new TexFileWriter(input).write("testInput");
33         new TexToPDFBuilder().build($var0);
34     }
35 }

```

The complete translation is given in Listing 4.5. 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 the object model that represents the document. Finally `finishDocument` compiles the document to L^AT_EX and compiles that to a PDF.

The reason we split up this process is so that we can call the creation of the document separately which allows us to include documents. To do this we call the `createDocument` method with an appropriate argument, we saw this in action in Subsection 3.1.4.

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

same as the name of the Neio document.

4.3.4 Reflection

In Subsection 3.1.3 we used `addClassMapping` and to tell the document class to create `ThesisChapters` instead of `Chapters`. This method is used to tell a `Content` to create specialized versions of a certain `Content`. It thus solves the same problems that factories solve. The reason we did not use actual factories is because we did not want the user to write them. Factories are usually straightforward and similar to each other, yet not reusable.

Every `Content` has a class mapping that is initialised as an empty map. To use a specialized version of a class, you add a mapping from the generic class to the specialized class.

The middle end of the compiler swaps out every new call to a `Content` in a `Content` by a call to `getInstance`. This method will check if there is a mapping for the generic class in this `Content`, or any of its parents. If there is, then an instance of specialized class is created. If not, an instance of the generic class is generated. The instances are created through reflection. The code for `getInstance` and the class mappings is shown 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     return descendants;
4 }
5
6 /**
7  * Returns the Class as which a Class should be instantiated
8  *
9  * @param klass The key of which we want to get a mapping
10 * @return The mapping found for key {@code klass}
11 */
12 final Class classMapping(Class klass) {
13     return _classMapping.get(klass);
14 }
15
16 /**
17  * Overrides default behaviour and instantiates new instances of {@code
18     oldClass} as instances of {@code newClass}
19  *
20  * @param oldClass The old class object
21  * @param newClass The new class object
22  */
23 public final <T> void addClassMapping(Class<T> oldClass, Class<? extends T>
24     newClass) {

```

```

23     _classMapping.put(oldClass, newClass);
24 }
25
26 /**
27  * Used instead of a new call. This allows to create more specific objects
28  * than originally
29  * specified. e.g. this allows to create SpecialChapter's instead of Chapter's
30  * <p>
31  * It will create an instance of {@code klass} in case the class mapping has
32  * not been overridden.
33  * If the mapping was overridden, an instance of the overriding class will be
34  * created.
35  *
36  * @param klass      The class to instantiate
37  * @param paramTypes The types of the parameters
38  * @param params      The parameters to instantiate {@code klass}
39  * @return The instantiated object
40  */
41 public final <T> T getInstance(Class<T> klass, Class[] paramTypes, Object[]
42     params) {
43     // Have I been overridden?
44     Class result = classMapping(klass);
45     Content current = parent();
46     // Has any of the parents been overridden?
47     while ((current != null) && (result == null)) {
48         result = current.classMapping(klass);
49         current = current.parent();
50     }
51
52     // No parents have been overridden
53     if (result == null) {
54         return instantiate(klass, paramTypes, params);
55     }
56     // A parent was overridden, use that type
57     else {
58         return (T) instantiate(result, paramTypes, params);
59     }
60 }
61
62 /**
63  * Does the actual instantiation of an object using Reflection
64  *
65  * @param klass      The class to instantiate
66  * @param paramTypes The types of the parameters
67  * @param params      The parameters to instantiate {@code klass}
68  * @return The instantiated object
69  */

```

```

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

```

We explain this again using the example below.

```

1 [Document]
2 # Chapter 1
3 {
4     nearestAncestor(Document.class).addClassMapping(Enumerate.class,
5         FibEnumerate.class);
6 }

```


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 as said before, `getInstance` also checks the parents for class mappings. We first check if a mapping is available in `Chapter`. As there is none, we recursively check the mappings of our parents. The mapping is found in the first parent, `Document`, and thus we create an instance of `FibEnumerate` one of `Enumerate`.

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 text mode 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 `\[bf]` have not been included in Neio as they are not yet needed. In the future it might prove useful to include these special characters too.

We can not just translate these characters straight to Java as we have no special meaning for `\[bf]`. For this reason, the middle end transforms every backslash into two backslashes, and transforms a double backslash (denotes 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 it creates valid \LaTeX . \LaTeX only has ten special characters thus replacing is quite easy. The code to do the replacing is shown below. `ESC` is an escape sequence that is used to prevent use from adjusting our changes, of the first replace, in the second replace.

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;
23 }
```

4.4 Resource usage and creation

As mentioned in Section 4.1,

4.5 Limitations

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

4.5.1 Java

4.5.2 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

[9] 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.

4.5.3 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.

```
1 "#" -> "hash"
2 "*" -> "star"
3 "=" -> "equalSign"
4 "^" -> "caret"
5 "-" -> "dash"
6 "_" -> "underscore"
7 "<" -> "backquote"
8 "$" -> "dollar"
9 "|" -> "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 2.9 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  }  
3  
4  /**  
5   * Returns a String representing this Content and its children as TeX.  
6   * The string is build by recursively calling {@code toTex} on all of children  
7   * in this Content.  
8   *  
9   * @return The TeX string  
10  */  
11 String toTex() {
```

```

12     StringBuilder tex = new StringBuilder();
13     for (int i = 0; i < content.size(); i = i + 1) {
14         tex.append("\n").append(content.get(i).toTex());
15     }

```

In fact we use it so many that in the future it might be better to just replace `Strings` by `StringBuilders` 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 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
3   * further chaining
4   *
5   * @param root The base of the square root
6   * @param arg The value to take the square root of
7   * @return This equation
8   */
9  InlineEq sqrt(Content root, Content arg) {
10     content().add(new Sqrt(root, arg));
11     return this;

```

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 then 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 `ContextTypes` 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 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 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.

In text mode it is annoying that we can not use more than one character as a symbol method. For example, it is not possible to use meaningful symbol such as `->` as a method. It would thus have been better if we had implemented symbol methods in a way that allows for more than one character to be used as an identifier.

As shown in Subsection 3.1.4, some work should also be put into making symbol methods more flexible. It should be possible to easily pass parameters of an object through symbol methods.

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 easy to read, write and learn. This was achieved by extending fluent interfaces and basing text mode on Markdown which is already easy to read, write and learn.

We also showed that it can be as powerful and diverse as \LaTeX whilst remaining quite legible. This was achieved by basing code mode on Java. The popularity of Java [6] assures that there are a lot of libraries that can be used directly, and it gives us a large user base that can read and understand code mode. The diverse examples showed in Chapter 3, and the completion of this book demonstrate this.

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.

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. Don't take LaTeX files from strangers. 2011.
- [5] Chemfig LaTeX package. <https://www.ctan.org/pkg/chemfig>. Accessed: 26-05-2016.
- [6] Nick Diakopoulos and Stephen Cass. The top programming languages 2015 according to iee spectrum. <http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2015>. Accessed: 30-05-2016.
- [7] Fluent interfaces. <http://www.martinfowler.com/bliki/FluentInterface.html>. Accessed: 27-05-2016.
- [8] GitHub Flavored Markdown. <https://help.github.com/enterprise/11.10.340/user/articles/github-flavored-markdown/>. Accessed: 27-05-2016.
- [9] Javadoc for process. <https://docs.oracle.com/javase/8/docs/api/java/lang/Process.html>. Accessed: 09-05-2016.
- [10] JMathTeX: TeX in Java. <http://jmathtex.sourceforge.net/>. Accessed: 08-05-2016.

-
- [11] The Jnome framework. <https://github.com/markovandooren/jnome>. Accessed: 18-05-2016.
 - [12] Donald E. Knuth and Michael F. Plass. Breaking Paragraphs into Lines. *Software - Practice and Experience*, 11:1119–1184, 1981.
 - [13] KOMA-Script guide. <http://texdoc.net/texmf-dist/doc/latex/koma-script/scrguien.pdf>. Accessed: 18-05-2016.
 - [14] LaTeX introduction. <https://latex-project.org/intro.html>. Accessed: 20-05-2016.
 - [15] LyX homepage. <https://www.lyx.org/>. Accessed: 25-05-2016.
 - [16] Markdown homepage. <https://daringfireball.net/projects/markdown/>. Accessed: 07-05-2016.
 - [17] Markdown syntax. <https://daringfireball.net/projects/markdown/syntax>. Accessed: 08-05-2016.
 - [18] MediaWiki LaTeX extension. <https://www.mediawiki.org/wiki/Extension:Math>. Accessed: 08-05-2016.
 - [19] Memoir user guide. <http://tug.ctan.org/tex-archive/macros/latex/contrib/memoir/memman.pdf>. Accessed: 18-05-2016.
 - [20] MetaUML. <https://github.com/ogheorghies/MetaUML>. Accessed: 18-05-2016.
 - [21] LaTeX Minipage. <http://www.sascha-frank.com/latex-minipage.html>. Accessed: 09-05-2016.
 - [22] The neio library. <https://github.ugent.be/tivervac/neio/tree/master/examples/0.8/lib/neio>. Accessed: 27-05-2016.
 - [23] 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.
 - [24] 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.

-
- [25] Pandoc. <http://pandoc.org/>. Accessed: 08-05-2016.
 - [26] Pandoc filter example. <http://pandoc.org/scripting.html#json-filters>. Accessed: 25-05-2016.
 - [27] Till Tantau. *The TikZ and PGF Packages*.
 - [28] The Colemak keyboard layout. <http://colemak.com/>. Accessed: 27-05-2016.
 - [29] The LaTeX beamer package. <https://www.ctan.org/pkg/beamer>. Accessed: 27-05-2016.
 - [30] The LilyPond homepage. <http://lilypond.org/>. Accessed: 30-05-2016.
 - [31] The MediaWiki homepage. <https://www.mediawiki.org/>. Accessed: 27-05-2016.
 - [32] The neio compiler. <https://github.ugent.be/tivervac/neio/tree/master/neio>. Accessed: 27-05-2016.
 - [33] The PanPipe Pandoc filter. <http://chriswarbo.net/git/panpipe/index.html>. Accessed: 27-05-2016.
 - [34] The StackExchange homepage. <https://www.stackexchange.com/>. Accessed: 27-05-2016.
 - [35] The Wikipedia homepage. <https://www.wikipedia.org/>. Accessed: 27-05-2016.
 - [36] The Wiktionary homepage. <https://www.wiktionary.org/>. Accessed: 27-05-2016.
 - [37] Marko van Dooren, Eric Steegmans, and Wouter Joosen. An object-oriented framework for aspect-oriented languages. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development, AOSD '12*, pages 215–226, New York, NY, USA, 2012. ACM.
 - [38] W3C EBNF specification. <https://www.w3.org/TR/REC-xml/#sec-notation>. Accessed: 27-05-2016.
 - [39] What is MetaPost. www.tex.ac.uk/FAQ-MP.html. Accessed: 30-05-2016.

