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*—This article presents the work performed in a thesis submitted in order to obtain the academic degree of Master of Science in Computer Science Engineering at Ghent University in June 2016. The goal of the thesis was to design a markup language that is easy-to-read and write, and that uses a modern programming model to allow for full customisation and control of the document. The state-of-the-art concerning document creation was researched and the design of the language was based on the results of this research. 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] is an example of such a language. It has a small fixed syntax [2] for the most common elements used in a document. Because of this it is very easy to read and write Markdown documents. However, it lacks customisability.

LaTeX [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 the modern programming models. It also requires a certain technical proficiency to be used.

As such we set out to design a markup language that is as easy to read, write and learn as Markdown but as powerful as LaTeX. The developed language was named Neio.

II. The design of Neio

We decided to use two files and two syntaxes in Neio. The first is a Neio document. This is the file that most users write. As such it has to be as simple as possible. It uses the text mode syntax which is based on Markdown. Markdown was chosen because the state-of-the-art analysis showed that it is very easy-to-read, write and learn. An example Neio document 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 kind of files is a code file. It is used to provide the objects and methods that are used in a text document. This file is written using code mode syntax which is almost a pure copy of the Java syntax.

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]. This provides a big advantage because every Java library can be used in Neio.

An important difference with Markdown is that the semantics of text mode syntax are not hard-coded into the language, instead they can be defined in the code files. 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.

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

III. Flexibility

A few libraries and DSLs were created to show the flexibility of Neio.

The first one is shown in Listing 3. The table is created using a DSL that consists of the `|`, `-` and `newline` methods. The rendered version is shown in Figure 1.

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.

```
[Document]
{
    Score s = new Score().c().d().e().f().g().a().
        b();
    return s;
}
{
    Score reversed = ss.reverse();
    return reversed
}
```

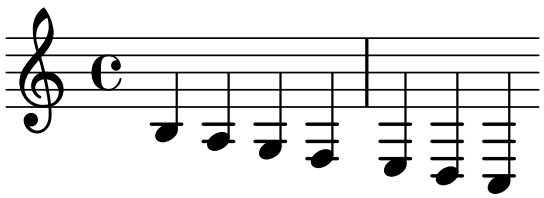

Fig. 2: The rendered version of the score in Listing 4.

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

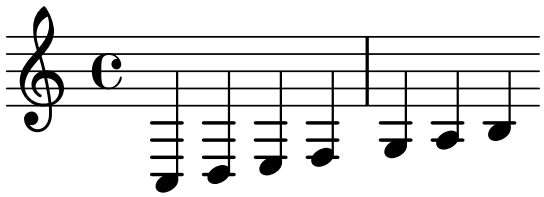

Fig. 3: The rendered version of the reversed score in Listing 4.

IV. Compiler

The process a Neio document goes through from Neio source code to LaTeX code, is shown in Figure 4.

First the Neio document and all of the code files are passed to the compiler. The compiler then reads the neio files using a parser that was generated with ANTLR4 [5] and builds a Concrete Syntax Tree. This tree is visited using the visitor pattern and an Abstract Syntax Tree (AST) of the files is build. The objects used in the AST are provided by Jnome [6] and Chameleon [7] [8].

Chameleon is a framework that can model a programming language, or as shown in this thesis, a markup language. It does so by providing objects for commonly used concepts of programming languages. A few examples of such objects are the following: `Expression`, `VariableDeclaration`, `Type`, `MethodInvocation`,....

Jnome extends the objects available in Chameleon with Java specific object. It is also able to read Java projects and output Java code for an AST consisting of Jnome and Chameleon objects, to Java.

The compiler then transforms the AST to a model that can be output to Java in the middle end. In the back end of the Neio compiler the transformed AST is written out as Java code.

The final transformation to LaTeX is done by code by the Neio standard library. To generate the LaTeX code the `toTex` method is called on the root of the object model. This object will then recursively call `toTex` on all the objects in the object model.

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 ieee spectrum," http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2015, accessed: 30-05-2016.

[5] "ANTLR4," http://www.antlr.org/, accessed: 18-05-2016.

[6] "The Jnome framework," https://github.com/markovandooren/jnome, accessed: 18-05-2016.

[7] "The Chameleon framework," https://github.com/markovandooren/chameleon, accessed: 18-05-2016.

[8] 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

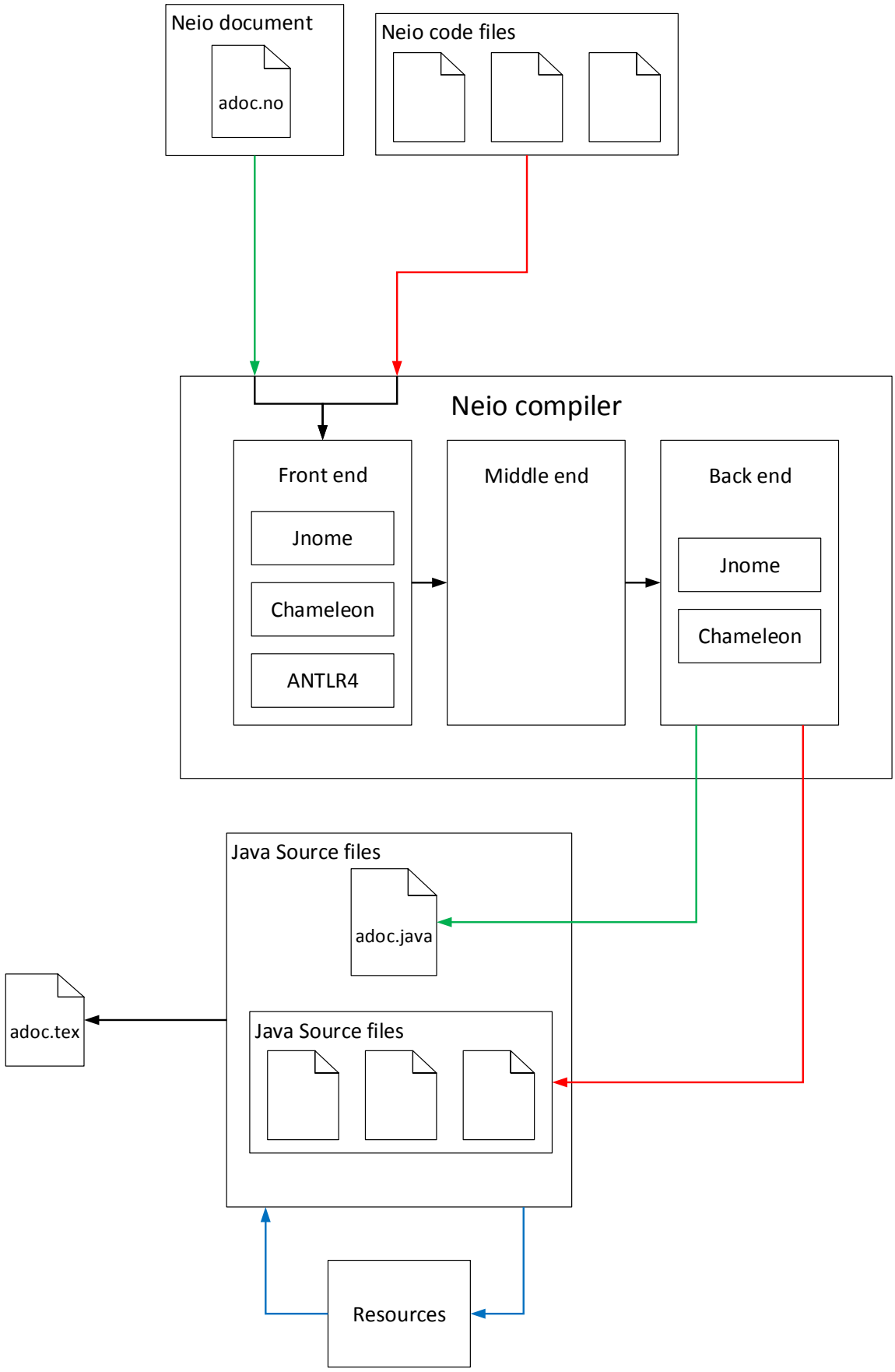

Fig. 4: The process of compiling Neio source code to LaTeX