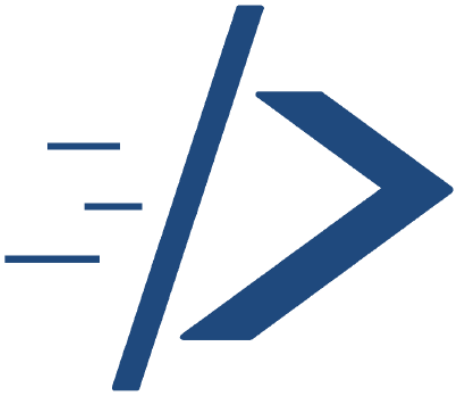
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores



Domain Specific Language generation based on a XML Schema

LUÍS CARLOS DA SILVA DUARTE

Licenciado em Engenharia Informática e de Computadores

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador : Doutor Fernando Miguel Gamboa de Carvalho

Júri:

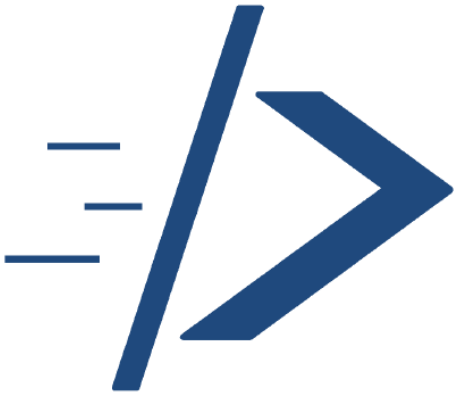
Presidente: [Doutor José Manuel de Campos Lages Garcia Simão]

Vogais: [Doutor António Paulo Teles de Menezes Correia Leitão]
[Doutor Fernando Miguel Gamboa de Carvalho]

Outubro, 2018

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores



Domain Specific Language generation based on a XML Schema

LUÍS CARLOS DA SILVA DUARTE

Licenciado em Engenharia Informática e de Computadores

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador : Doutor Fernando Miguel Gamboa de Carvalho

Júri:

Presidente: [Doutor José Manuel de Campos Lages Garcia Simão]

Vogais: [Doutor António Paulo Teles de Menezes Correia Leitão]
[Doutor Fernando Miguel Gamboa de Carvalho]

Outubro, 2018

Aos meus pais.

Acknowledgments

Ao meu orientador, por todo o apoio que me deu ao longo da realização desta dissertação. A todos os meus amigos que me acompanharam, ajudaram e animaram nesta jornada. E em particular, um grande agradecimento aos meus pais, sem eles nada disto seria possível.

Acronyms and Abreviations

The list of acronyms and abbreviations are as follow.

API *Application Programming Interface* 3
DOM *Document Object Model* 34
DSL *Domain Specific Language* xi
HTML *HyperText Markup Language* xi
IDE *Integrated Development Environment* 61
JAR *Java ARchive* 77
JMH *Java Microbenchmark Harness* 73
JVM *Java Virtual Machine* 29
POM *Project Object Model* 60
SAX *Simple Application Programming Interface for eXtensive Markup Language* . . 34
SQL *Structured Query Language* 2
XHTML *eXtensive HyperText Markup Language* 34
XML *eXtensive Markup Language* xi
XSD *eXtensive Markup Language Schema Definition* xi

Abstract

The use of *markup languages* is recurrent in the world of technology, with *HyperText Markup Language* (HTML) being the most prominent one due to its use in the Web. The requirement of tools that can automatically build well formed documents with good performance is clear. Yet, the most used solution is *template engines*, which neither ensures well-formed documents nor presents good performance, due to the use of external text files.

To tackle the first issue we propose to define HTML templates as first-class functions instead of using text files. To that end, these HTML template functions use a Java *Domain Specific Language* (DSL) to write HTML. Our main goal is to create the required tools that automatically generate that DSL based on its language definition from an *eXtensive Markup Language Schema Definition* (XSD) file. The resulting DSL should enforce the restrictions of the given language which are specified in the XSD file. By removing the use of text files we are also suppressing the file load overhead and reducing `String` manipulation, which in turn increases the overall performance and solves the second issue.

My proposal, named `xmlet`, includes a set of tools that are able to: 1) parse and extract the rules from a XSD file, 2) generate the adequate classes and methods to define the DSL that reflects the language rules, 3) handle the use of the resulting DSL through the implementation of the Visitor pattern. Finally we validated this solution not only for HTML, but also with the Android *eXtensive Markup Language* (XML) for Android layouts and the regular expressions language.

By comparing the developed solution to some state-of-art solutions, including *template engines* and some other solutions with specific innovations, we obtained

very favorable results with the suggested solution being the best performance-wise in all the tests we performed. These results are important, specially considering that apart from being a more efficient solution it also introduces validations of the language usage based on its syntax definition.

Keywords: XML, eXtensive Markup Language, XSD, eXtensive Markup Language Schema Definition, Automatic Code Generation, Fluent Interface, Domain Specific Language.

Resumo

Actualmente a utilização de linguagens de *markup* é recorrente no mundo da tecnologia, sendo o HTML a linguagem mais utilizada graças à sua utilização no mundo da Web. Tendo isso em conta é necessário que existam ferramentas capazes de escrever documentos bem formados de forma eficaz. No entanto, a abordagem mais utilizada, *template engines*, tem dois problemas principais: 1) não garante a geração de documentos bem formados, 2) não garante um bom desempenho, devido à utilização de ficheiros de texto como ficheiros de template.

Para resolver o primeiro problema propomos que um template HTML passe a ser definido como uma *first-class function*. Para isto é necessário criar uma linguagem específica de domínio para que estas funções possam manipular a linguagem HTML. O nosso objectivo principal é criar as ferramentas necessárias para gerar linguagens específicas de domínio com base no ficheiro de definição da linguagem, explícito num ficheiro XSD. A linguagem de domínio gerada deve também garantir que as restrições da respectiva linguagem são verificadas. Removendo os ficheiros textuais que definem templates minimizam-se também os problemas de desempenho introduzidos pelo carregamento de ficheiros de texto e reduzem-se o número de operações sobre `Strings`.

A minha proposta, chamada `xmlet`, inclui ferramentas que possibilitam: 1) a análise e a extração de informação de um ficheiro XSD, 2) a geração de classes e métodos que definem uma linguagem específica de domínio que reflete as regras presentes no ficheiro XSD, 3) a abstração da utilização da linguagem de domínio gerada, com a utilização do padrão Visitor. Para validar esta solução criaram-se linguagens de domínio não só para HTML como também para a linguagem utilizada para definir layouts visuais para Android e para a linguagem das expressões regulares.

Comparando a solução desenvolvida com soluções semelhantes, incluindo *template engines* e algumas soluções com inovações face à abordagem dos *template engines*, obtemos resultados favoráveis. Verificamos que a solução sugerida é a mais eficiente em todos os testes feitos. Estes resultados são importantes, especialmente considerando que apesar de ser a solução mais eficiente introduz também a verificação das restrições da linguagem utilizada tendo em conta a sua definição sintática.

Palavras-chave: XML, eXtensive Markup Language, XSD, eXtensive Markup Language Schema Definition, Geração Automática de Código, Interface Fluente, Linguage Específica de Domínio.

Contents

List of Figures

List of Tables

List of Listings

1

Introduction

The research work that I describe in this dissertation is concerned with the implementation of a Java framework, named `xmlet`, which allows the automatic generation of a Java *fluent interface*[6] recreating a DSL[7] specified by an XML schema. The approach described in this work can be further applied to any other strongly typed environment. As an example of DSL generation we used `xmlet` to automatically generate a Java DSL for HTML5, used by HtmlFlow[4]. A DSL for HTML can be used as a type safe *template engine* that results in an improvement in the numerous existing *template engines*. Furthermore, HtmlFlow outperforms state-of-the-art Java *template engines*, namely Rocker[22], Pebble[20], Trimou[27] or Mustache[18], in some of the more challenging benchmarks such as *template-benchmarks*[3] and *spring-comparing-template-engines*[21].

1.1 Introduction to Domain Specific Languages

High-level programming languages such as Java, C#, JavaScript and others were created with the objective of being abstract, in the sense that they do not compromise with any specific problem. Using these programming languages is usually enough to solve most problems but in some specific situations solving problems using exclusively those languages is counter-productive. A good example of that counter-productivity is thinking about regular expressions. In the Martin Fowler DSL book[8] we can find the regular expression presented in Listing 1.1.

```
1 \d{3}-\d{3}-\d{4}
```

Listing 1.1: Regular Expression Example

Looking at the expression of Listing 1.1 the programmer understands that it matches a `String` similar to `123-321-1234`. Even though the regular expression syntax might be hard to understand at first glance, it may become understandable after a while. It may be easier to use and manipulate by experts than implementing the same set of rules to verify a `String` using control instructions such as `if/else` and `String` operations. It also makes the communication between experts easier when dealing with this concrete problem because there is a standard syntax with well-known rules. Regular expressions are just one of the many examples that show that creating an explicit language to deal with a very specific problem simplifies said problem. Other examples of DSLs are languages such as *Structured Query Language* (SQL)[25], `Apache Ant`[23] or `make`[24].

DSLs can be divided in two types: external or internal. External DSLs are languages created without any affiliation to a concrete programming language. An example of an external DSL is the regular expressions DSL, since it defines its own syntax without any dependency of programming languages. On the other hand an internal DSL is defined within a programming language, such as Java. An example of an internal DSL is jMock[12], which is a Java library that provides tools for test-driven development as shown in Listing 1.2.

```
1 final GreetingTime gt = context.mock(GreetingTime.class);
2 Greeting g = new Greeting();
3 g.setGreetingTime(gt);
4
5 context.checking(new Expectations(){{
6   one(gt).getGreeting();
7   will(returnValue("Good afternoon"));
8 }});
```

Listing 1.2: jMock Use Example

In Listing 1.2 we can see that jMock uses a DSL to create expectations. In the concrete example it obtains the value of a `Greeting`, in line 6 of Listing 1.2, and asserts if the value of the `Greeting` matches the expected value, which is `Good afternoon`, shown in line 7 of Listing 1.2. In this case the semantics of the methods used by jMock aim to simplify the programmer's understanding of the tests that are being performed.

Internal DSLs can also be referred to as embedded DSLs since they are embedded

in the programming language where they are used. Another common term for an internal DSL is *fluent interface* or *fluent* Application Programming Interface *(API)*. The term fluent is inspired by the fluent way that the DSL usage can be read, which is close to a natural language.

Concluding, there are some advantages on internal DSLs over external DSLs, namely: a single compiler, removal of language heterogeneity and in some situations, performance improvements and overall a less complex solution.

From a simple point of view, one of the main goals of this research work is to propose an approach and develop a platform, i.e. `xmlet`, which enables the conversion of a XML based external DSL into an internal DSL. One of the requirements of this approach is that the external DSL's rules must be defined by an XSD document. Thus, on one hand we have a XSD document that defines a set of elements, attributes and rules that together define their own XML language. From a Java environment point of view this XML language is qualified as an external DSL since it is defined in XML, which is a *markup language* that does not depend of the Java programming language. All the information present in the XSD document is used to generate a Java *fluent interface*, which, in this case, is an internal DSL since it uses the Java syntax to define the DSL.

Using this approach we are going to generate a *fluent interface* for the HTML5 language, based on its XSD document. The result of our approach is the automatic code generation of Java classes and interfaces that will reflect all the information present in the XSD document. When we analyze the end result of this work, what we achieve is a Java interface to manipulate a DSL, in this case HTML, which can be used for anything related with HTML manipulation with the upside of having the guarantee that language rules are verified. One of those usages is writing well-formed HTML documents and defining *dynamic views* that will be filled with information received in runtime. An example of a static view is presented in Listing 1.3.

```
1 private static void staticView(StaticHtml view){
2     view.html()
3             .body()
4                 .h1()
5                     .text("This is a static view h1 element.")
6                 .__()
7             .__()
8         .__();
9 }
```

Listing 1.3: Static View Example with HtmlFlow

The static view of Listing 1.3 shows how an internal DSL guarantees the rules of the language by using Java to enforce them. In this concrete example we can see that the logic of element organization of the HTML language is translated to Java methods, ensuring that, for example, the `html` element can only contain either `head` or `body` element as children, as stated by the HTML5 specification.

1.2 Template Engines

Template engines are solutions that use views, more specifically *dynamic views*, to build documents. A view is the output representation of information used to build the user interface of an application. Regarding web applications the view may be defined using the HTML language.

1.2.1 Dynamic Views

In this context, a *dynamic view* is a template with two distinct components, as shown in Listing 1.4, a static component, represented in blue, which defines the structure of the document and a dynamic component, represented in green, which is represented by *placeholders* that are replaced by information received at runtime. A simple example of a *dynamic view* can be an HTML template with the information of a given `Student` object, as shown in Listing 1.4.

```
<html>
    <body>
        <ul>
        {{#student}}
            <li>
                {{name}}
            </li>
            <li>
                {{number}}
            </li>
        {{/student}}
        </ul>
    </body>
</html>
```

Listing 1.4: HTML Template of Student Information in Mustache Idiom

To generate the resulting HTML page from the template of Listing 1.4 we need external input, received at runtime, to resolve the dynamic component of the view. In the previous example, Listing 1.4, the view needs to receive a value for the variable named `{{student}}`. The type that the `student` variable represents should be a type that contains two fields, a `number` and a `name` field. An example of an object with that characteristics is presented in Figure 1.1.

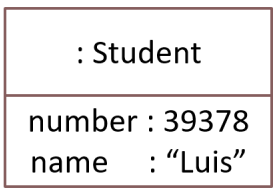


Figure 1.1: Student Object

After the template in Listing 1.4 receives the *context object* presented in Figure 1.1 the resulting HTML document should match the one shown in Listing 1.5.

```
<html>
    <body>
        <ul>
            <li>
                Luis
            </li>
            <li>
                39378
            </li>
        </ul>
    </body>
</html>
```

Listing 1.5: HTML Document with Student Information

Template engines are responsible for generating an HTML document based on a template. *Template engines* are the most common method to manipulate *dynamic views*. *Template engines* are responsible for performing the combination between the *dynamic view*, also named *template*, and a data model object, known as *context object*, which contains all the information required to generate the final document. The example depicted in Figure 1.2 shows the combination of a template document with a *context object*, which is an instance of the `Student` types.

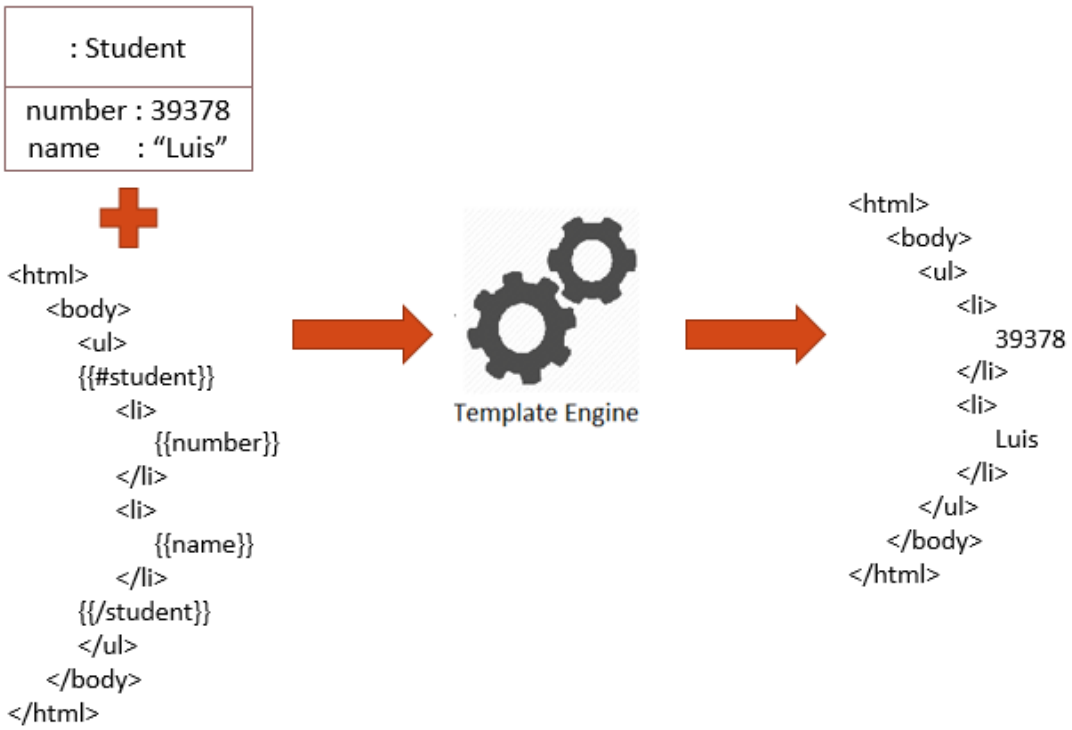


Figure 1.2: The Template Engine Process that Combines a Template Document with a Context Object

Since the Web appearance there is a wide consensus around the use of *template engines* to build dynamic HTML documents. From the vast list of existing web *template engines*[30] all of them share the same approach based on a textual template file. The *template engine* scope is also wide, even that they are mostly associated with Web, they are also widely used to generate other types of documents such as emails and reports.

1.2.2 Handicaps

Although there is a wide consensus in the use of *template engines*, this approach still has some handicaps, which we will further analyze.

- Safety and Type Check - There are no validations of the language used in the templates nor the dynamic information. This can result in documents that do not follow the language rules.
- Performance - This aspect can be divided in two, one regarding the text files that are used as templates that have to be loaded and therefore slow the overall performance of the application and the heavy use of `String` operations, which are inherently slow.
- Flexibility - The syntax provided by the *template engines* is sometimes very limited and restricts the operations that can be performed in the template files to few control flow instructions such as `if/else` operations and the `for` operation to loop data.
- Complexity - It introduces one more syntax in the programming environment. For example a Java application using the Mustache[18] *template engine* forces the programmer to use three distinct languages: Java, the Mustache syntax in the template file and the HTML language.

1.3 Thesis Statement

This dissertation thesis is that it is possible to reduce the problems that exist within the use of *template engine* solutions. To suppress these handicaps we propose the creation of a process that automatizes the generation of DSLs based on an existing DSL specified by one XSD document. This process is implemented by the `xmlet` platform, which allows the automatic generation of a strongly typed

and *fluent interface* for a DSL based on the rules expressed in the XSD document of that respective language, such as HTML. The resulting DSL addresses the handicaps of the *template engines* in the following way:

- Safety and Type Check - The generated Java DSL will guarantee the implementation of the language rules defined in the XSD file by reflecting those restrictions in the generated *fluent interface*.
- Performance - Text templates files are replaced by pure Java functions, according to my approach a template is a first-class function.
- Flexibility - The syntax used to perform operations on templates is replaced with the Java syntax without any restriction to the use of all its features.
- Complexity - It replaces the heterogeneity of using three programming languages, i.e. Java, HTML and the *template engine* specific idiom, with the use of a single programming language, i.e. Java.

A brief view of the generated *fluent interface* is presented in Listing 1.6, which shows how the previous example in the Mustache idiom, i.e. Listing 1.4, will be recreated using the `xmlet` solution. The specific details on how the code presented in this example works will be provided in Chapter 4.

```
1  String document = DynamicHtml.view(CurrentClass::studentView)
2                                .render(new Student("Luis", 39378));
3
4  static void studentView(DynamicHtml<Student> view, Student student){
5      view.html()
6              .body()
7                .ul()
8                  .li().dynamic(li -> li.text(student.getName())).__()
9                  .li().dynamic(li -> li.text(student.getNumber())).__()
10               .__()
11             .__()
12           .__();
13 }
```

Listing 1.6: Xmlet Template with Student Information

To implement the `xmlet` process we created three distinct components:

- XsdParser - Parses the DSL described in a XSD document in order to extract information needed to generate the internal Java DSL.

- XsdAsm - Uses XsdParser to gather the required information and uses it to generate the internal Java DSL.
- HtmlApi - A concrete Java DSL for the HTML5 language generated by XsdAsm using the HTML5 XSD document.

The use case for this dissertation will be the HTML language but the process is designed to support any domain language that has its definition described in the XSD syntax. This means that any XML language should be supported as long as it has its set of rules properly defined in a XSD file, as well as any other domain language that can describe its rules in the XSD syntax. To show that this solution is viable with other XSD files we created two additional *fluent interfaces*:

- Android Layouts - Based on a preexisting XSD document detailing the language used to create Android visual layouts;
- Regular Expressions - Based on a created XSD document detailing the operations available in the regular expressions syntax.

1.4 Document Organization

This document will be separated in six distinct chapters. The first chapter, this one, introduces the concept that will be explored in this dissertation. The second chapter introduces the motivation for this dissertation. The third chapter presents existent technology that is relevant to this solution. The fourth chapter explains in detail the different components of the suggested solution. The fifth chapter approaches the deployment, testing and compares `xmlet` to other preexisting solutions. The sixth and last chapter of this document contains some final remarks and description of future work.

2

Problem Statement

In the first chapter we presented *template engines* and discussed their theoretical handicaps, in this chapter we will further analyze other limitations that are presented while using them in a practical setting. This analysis aims to show how fragile the usage of this type of solution can be and the problems that are inherited by using it.

2.1 Motivation

Text has evolved with the advance of technology resulting in the creation of *markup languages* [5]. *Markup languages* work by adding annotations to text, the annotations being also known as tags, which allow to add additional information to the text. Each *markup language* has its own tags and each of those tags add a different meaning to the text encapsulated within them. In order to use *markup languages* the users can write the text and add all the tags manually, either by fully writing them or by using some kind of text helpers such as text editors with IntelliSense[1], which can help diminish the errors caused by manually writing the tags. But even with text helpers the resulting document can violate the restrictions of the respective *markup language* because the editors do not actually enforce the language rules since there is not a process similar to a compile process that can either pass of fail. The most that a text editor can do is highlight the errors to the user.

[1] https://www.techopedia.com/definition/24580/intellisense

The most well-known *markup language* is HTML, which is highly used in Web applications. Other uses of the HTML language are in emails, writing reports, etc.

Now we will present different issues resulting from the use of *template engines* to build HTML views. The examples provided in this section use eight different *template engines*: Freemarker[1], Handlebars[13], Mustache[18], Pebble[20], Thymeleaf[26], Trimou[27], Velocity[28] and Rocker[22]. The templates used in these experimental tests were the same used for different benchmarks presented in Chapter 5. The three issues that will be addressed are:

- Issue 1 - Guarantees of well formed documents;
- Issue 2 - Validation of the HTML language rules;
- Issue 3 - Validation of *context objects*.

We will start with the most basic aspect that we expect from a HTML document, it should be well formed. Let us start with a very simple example as shown in Listing 2.1.

```
1 <html>
2   <!-- -->
3 </html>
```

Listing 2.1: Badly Formed HTML Document

Let us imagine that for some typing mistake the red characters are missing, which means that the opening `<html>` tag is not properly written and its closing tag is not present. It would be expected that in the very least the *template engine* would issue an error while reading the file at run time. But all *template engines* used in this experiment have not issued any kind of error. This is problematic, because the error was not caught neither at compile time nor at run time. These kind of errors would only be observable either on a browser or by using any kind of external tool to verify the resulting HTML page. This is the case where an internal DSL such as the one presented in Listing 1.3 suppresses this problem, since the responsibility of creating tags and properly opening and closing them belongs to the DSL library and not to the person who is writing the template.

Addressing the second issue, the rules of the used language should be validated. The HTML language specification specifies many restrictions, either on attribute

types or regarding the organization of the element tree. For example, let us think about the `html` element, the specification states that the only elements that can be direct children of the `html` element are `head` and `body` elements. That means that if we try to define a template as shown in Listing 2.2 the *template engines* should inform us that we are violating the language rules.

```
1 <html>
2     <div>
3       <!-- -->
4     </div>
5 </html>
```

Listing 2.2: Invalid Html Element Containing a Div Element

After trying to use the template of Listing 2.2 with the eight *template engines* that we are using, none of them issued any compile time error, nor any run time error. This means that this error would have to be manually detected by the person who is writing the template, and taking the nature of these rules in consideration it would be hard to verify them manually. By using an approach such as `xmlet` we would discover this error at compile time, as many other similar errors. The organization of elements can be validated at compile time as well as the primitive attribute types. Some validations, such as types with complex restrictions, have to be validated at run time, but even then the feedback is immediate and the user receives a detailed error message.

The third issue that we are going to pinpoint is the use of *context objects*. Every *template engine* uses them, since it contains the information that the *template engine* will use to fill out the *placeholders* defined in the textual template file. But what problems arise from their usage?

```
<html>
    <body>
        <ul>
        {{#student}}
            <li>
                {{name}}
            </li>
            <li>
                {{number}}
            </li>
        {{/student}}
        </ul>
    </body>
</html>
```

Listing 2.3: HTML Template with Placeholders

The template of Listing 2.3 receives a `Student` object that contains a `name` and `number` fields. Most *template engines* use a `Map<String, Object>` as the *context object*. In this case, a valid *context object* should look like the `Map` object created in Listing 2.4.

```
Map<String, Object> context = new HashMap<>();
context.put("student", new Student("Luis", 39378));
```

Listing 2.4: Template Engine with a Valid Context Object

The *context object* defined in the previous example, Listing 2.4, is valid. The *context object* is valid since it defines a pair with the key `student` associated with a `Student` object, which contains a `name` and `number` fields. This definition corresponds with the usage performed in the template defined in Listing 2.3. Yet, in Listing 2.5 and Listing 2.6 we show another *context object* definitions which are invalid, since their contents do not match the information expected by the template defined in Listing 2.3.

```
Map<String, Object> context = new HashMap<>();
context.put("teacher", new Student("Luis", 39378));
```

Listing 2.5: Template Engine with a Context Object with a Wrong Key

```
Map<String, Object> context = new HashMap<>();
context.put("student", new Teacher("MEIC", "ADDETC"));
```

Listing 2.6: Template Engine with a Context Object with a Wrong Type

The first *context object*, Listing 2.5, has a wrong key, `teacher`, whereas the template is expecting an object with the `student` key. The second *context object*, Listing 2.6, has the right key but has a different type, which does not match the fields expected by template of Listing 2.3.

With this information in mind how will the eight *template engines* react when receiving these invalid *context objects*? The Rocker *template engine* is the only one which deals with it in a safe way since its template defines the type that will be received. Moreover its template file is used in the generation of a Java class at compile time, which reflects the template information, and its usages are all safe regarding the *context object*, because the Java compiler validates if the object received as *context object* matches the expected type. The remaining seven *template engines* have no static validations. None of them issue any compile time warning.

Regarding runtime safety only Freemarker issues an exception with a similar example to Listing 2.5 and in the second case, Listing 2.6, only Freemarker and Thymeleaf throw an exception. The remaining solutions ignore the fact that something that is expected is not present and delay the error finding process until the generated file is manually validated.

In this case the use of an internal DSL suppresses this problem, as the template is defined by a Java function where the *context object* is an argument validated at compile time.

Another improvement of using an internal DSL over the use of *template engines* is the language homogeneity. For example, even for the simplest templates we have to use at least three distinct syntaxes. In the following example we will use the Pebble *template engine*, one of the less verbose templates. In this example we define a template to write an HTML document that presents the `name` of all the `Student` objects present in a `Collection` as shown in Listing 2.7.

```
1 <html>
2   <body>
3       <ul>
4       {% for student in students %}
5         <li>{{student.name}}</li>
6       {% endfor %}
7     </ul>
8     </body>
9 </html>
```

Listing 2.7: List of Student Names - Template Definition using Pebble

In this template alone we need to use two distinct syntaxes, the HTML language and the Pebble syntax to express that the template will receive a `Collection` that should be iterated and create `li` tags containing the `name` field of the `Student` type. Apart from the template definition we also need the Java code to generate the complete document, as shown in Listing 2.8.

```
1 PebbleEngine engine = new PebbleEngine.Builder().build();
2 template = engine.getTemplate("templateName.html");
3 StringWriter writer = new StringWriter();
4
5 Map<String, Object> context = new HashMap<>();
6 context.put("students", getStudentsList());
7
8 template.evaluate(writer, context);
9
10 String document = writer.toString();
```

Listing 2.8: List of Student Names - Template Building in Java using Pebble

Even though the template in Listing 2.7 is simple the usage of multiple syntaxes introduces more complexity to the problem. If we scale the complexity of the template and the number of different types used in the *context object* mistakes are bound to happen, which would be fine if the *template engines* gave any kind of feedback on errors, but we already shown that most errors are not reported. Let us take a peek of how this same template would be presented in the latest HtmlFlow version with the definition of the template in Listing 2.9 and the template building in Listing 2.10.

```
1 static void studentListTemplate(DynamicHtml<Iterable<Student>> view,
2                                 Iterable<Student> students){
3     view.html()
4             .body()
5                 .ul()
6                     .dynamic(ul ->
7                       students.forEach(student ->
8                         ul.li().text(student.getName()).__()))
9                 .__()
10            .__()
11        .__();
12 }
```

Listing 2.9: List of Student Names - Template Definition using HtmlFlow with xmlet

```
1 String document = DynamicHtml.view(CurrentClass::studentListTemplate)
2                                .render(getStudentsList());
```

Listing 2.10: List of Student Names - Template Building using HtmlFlow with xmlet

With this solution we have a very compact template definition, where the *context object*, i.e. the `Iterable<Student> students`, shown in line 2 of Listing 2.9, is validated by the Java compiler in compile time, which guarantees that any document generated by this solution will be valid since the program would not compile otherwise. This solution internally guarantees that the HTML tags are created properly, having matching opening and ending tags, meaning that every document generated by this solution will be well formed regardless of the defined template.

2.2 Problem Statement

The problem that is being presented revolves around the handicaps of *template engines*, the lack of compilation of the language used within the template, the performance *overhead* and the issues resulting from the increase of complexity, as presented in Section 1.2.2. To tackle those handicaps we suggested the automated generation of a strongly typed *fluent interface*. To show how that *fluent interface* will effectively work we will now present a small example that consists on the `html` element, Listing 2.11, described in XSD of the HTML5 language definition. The presented example is simplified for explanation purposes. In the examples that are presented below we will use a common set of types that serve as a basis to every *fluent interface* generated by `xmlet`, these classes will be presented in Section 4.2.1.

```
1  <xsd:attributeGroup name="globalAttributes">
2      <xsd:attribute name="accesskey" type="xsd:string" />
3      <!-- Other global attributes -->
4  </xsd:attributeGroup>
5
6  <xsd:element name="html">
7      <xsd:complexType>
8          <xsd:choice>
9              <xsd:element ref="body"/>
10             <xsd:element ref="head"/>
11         </xsd:choice>
12         <xsd:attributeGroup ref="globalAttributes" />
13         <xsd:attribute name="manifest" type="xsd:anyURI" />
14     </xsd:complexType>
15 </xsd:element>
```

Listing 2.11: <hmtl> Element Description in XSD

With this example there are a multitude of classes and members that need to be created:

- `Html` Class - A class that represents the `html` XSD element defined in line 6 of Listing 2.11. The resulting class is presented in Listing 2.12, deriving from `AbstractElement`.

- `body()` and `head()` Methods - These methods are present in the `Html` class, lines 12 and 14 of Listing 2.12 respectively. These methods use the `addChild(Element e)` method to add instances of the `Body` type, shown in Listing 2.13, and `Head` type, shown in Listing 2.14, to the `Html` class children list. These methods belong in the `Html` class since they are defined as possible children of the `html` XSD element, with the usage of the `<xsd:choice>` element in line 8 of Listing 2.11.

- `attrManifest(String manifest)` Method - A method present in `Html` class, line 8 of Listing 2.12, which uses the `addAttr(Attribute a)` method to add an instance of the `AttrManifest` type, shown in Listing 2.15, to the `Html` attribute list. This method is present in the `Html` class because the `html` XSD element defines an XSD attribute named `manifest` with the type `xsd:anyURI`, which is mapped to the `AttrManifest` type, shown in Listing 2.15.

```
1  class Html extends AbstractElement implements GlobalAttributes {
2      public Html() { }
3
4      public void accept(Visitor visitor){
5           visitor.visit(this);
6      }
7
8      public Html attrManifest(String attrManifest) {
9          return this.addAttr(new AttrManifest(attrManifest));
10     }
11
12     public Body body() { return this.addChild(new Body()); }
13
14     public Head head() { return this.addChild(new Head()); }
15 }
```

Listing 2.12: Html Class Corresponding to the XSD Element Named html

- `Body` and `Head` Classes - Classes created based on the `body` and `head` XSD elements. The classes are shown in Listing 2.13 and Listing 2.14 respectively. These classes will be generated using the same process used to generate the `Html` class, with the differences between the classes depending on the contents of their respective XSD elements.

```
1  public class Body extends AbstractElement {
2      //Similar to Html, based on the contents of
3      //<xsd:element name="body">
4  }
```

Listing 2.13: Body Class Corresponding to the XSD Element Named body

```
1  public class Head extends AbstractElement {
2      //Similar to Html, based on the contents of
3      //<xsd:element name="head">
4  }
```

Listing 2.14: Head Class Corresponding to the XSD Element Named head

- `AttrManifest` Class - A class that represents the `manifest` XSD attribute, defined in line 13 of Listing 2.11. The `AttrManifest` class is shown in Listing 2.15, deriving from `BaseAttribute`.

```
1 public class AttrManifest extends BaseAttribute<String> {
2     public AttrManifest(String attrValue) {
3         super(attrValue);
4     }
5 }
```

Listing 2.15: AttrManifest Class Corresponding to the XSD Attribute Named manifest

- `GlobalAttributes` Interface - An interface representing the `globalAttributes` XSD attribute group, defined in line 1 of Listing 2.11. This interface has default methods for each attribute it contains, e.g. the `accesskey` attribute defined in line 2 of Listing 2.11 is used to generate the `attrAccesskey` method shown in line 2 of Listing 2.16. The default methods objective is to add a certain attribute to the attributes list of the type that implements the interface. This interface is implemented by all the generated classes that are based on a XSD element that contains a reference to the attribute group that this interface represents, e.g. the `Html` class implements the `GlobalAttributes` interface because the `html` XSD element contains a reference to the `globalAttributes` XSD attribute group, line 12 of Listing 2.11.

```
1 public interface GlobalAttributes extends Element {
2     default Html attrAccesskey(String accesskeyValue) {
3         this.addAttr(new AttrAccesskey(accesskeyValue));
4         return this;
5     }
6
7     // Similar methods for the remaining attributes
8     // present in the globalAttributes attributeGroup.
9 }
```

Listing 2.16: GlobalAttributes Interface Corresponding to the XSD Attribute Group Named globalAttributes

By analyzing this little example we can observe how `xmlet` implements one of its most important features that was lacking in the *template engine* solutions, the user is only allowed to generate a tree of elements that follows the rules specified by the XSD file of the given language, e.g. the user can only add `Head` and `Body` instances as children to the `Html` class and the same goes for attributes as well, to add attributes to an `Html` instance the user can only use methods that add

an instance of the `AttrManifest` class or the default methods provided by the `GlobalAttributes` interface. This solution effectively uses the Java compiler to enforce most of the specific language restrictions. The other handicaps are also solved, since the template can now be defined within the Java language eradicating the requirement of textual files that still need to be loaded into memory and resolved by the *template engine*. The complexity and flexibility issues are also tackled by moving all the parts of the problem to the Java language, removing language heterogeneity and allowing the programmer to use the Java syntax to create the templates.

2.3 Approach

The approach to achieve a solution was to divide the problem into three distinct aspects, as previously stated in Section 1.3.

The XsdParser project is an utility project that is required in order to parse all the external DSL rules present in the XSD document into structured Java classes.

The XsdAsm project is the most important aspect of `xmlet`, since it is the aspect that will deal with the generation of all the *bytecodes* that make up the classes of the Java *fluent interface*. This project should translate as many rules of the parsed language definition, its XSD file, into the Java language in order to make the resulting *fluent interface* as similar as possible to the language definition.

The HtmlApi is the main use case for `xmlet`. It is a concrete client of the XsdAsm project, it will use the HTML5 language definition file in order to request a strongly typed *fluent interface*, named HtmlApi. This use case is meant to be used by the HtmlFlow library, which will use HtmlApi to manipulate the HTML language to write well formed documents.

3

State of Art

In this chapter we are going to introduce the technologies used in the development of this work, such as the XSD language in order to provide a better understanding of the next chapters, and also introduce the latest solutions that moved on from the usual *template engine* approach and in different ways tried to innovate in order to introduce safety and reliability to the process of generating HTML documents.

3.1 XSD Language

The XSD language is a description of a type of XML document. The XSD syntax allows the definition of a set of rules, elements and attributes that together define an external DSL. This specific language defined in a XSD document aims to solve a specific issue, with its rules serving as a contract between applications regarding the information contained in the XML files that represent information of that specific language. The XSD main purpose is to validate XML documents, if the XML document follows the rules specified in the XSD document then the XML file is considered valid otherwise it is not. To describe the rules and restrictions for a given XML document the XSD language relies on two main types of data: elements and attributes. Elements are the most complex data type, they can contain other elements as children and can also have attributes. Attributes on the other hand are just pairs of information, defined by their name and their value.

The value of a given attribute can be restricted by multiple constraints existing on the XSD syntax. There are multiple elements and attributes present in the XSD language, which are specified in the XSD Schema rules[29]. In this dissertation we will use the set of rules and restrictions of the provided XSD documents to build a *fluent interface* that will enforce the rules and restrictions specified by the given file.

3.2 The Evolution of Template Engines

We have already presented the idea behind *template engines* in Section 1.2 and their handicaps in Section 1.2.2, but here we are going to present some recent innovations that some *template engines* introduced in order to solve or minimize some of the problems listed previously. We are going to compare the features each solution introduces and create a general landscape of the preexisting solutions similar to the use case that `xmlet` will use.

3.2.1 HtmlFlow 1

The HtmlFlow[4] library was the first to be approached in the developing process of `xmlet`. The HtmlFlow motivation is to provide a library that allowed its users to write well formed type-safe HTML documents. The HtmlFlow version that existed prior to this project, which will be named HtmlFlow 1, only supported a subset of the HTML language, whilst implementing some of the rules of the HTML language. This solution was a step in the right direction, it removed the requirement to have textual files to define templates by moving the template definition to the Java language. It also provided a very important aspect, it performed language validations at compile time, which is great since it guarantees that those problems will be solved at compile time instead of run-time. The main downside of this solution was that it only supported a subset of the HTML language, since recreating all the HTML language rules manually would be very time consuming and error prone. This problem led to the requirement of creating an automated process to translate the language rules to the Java language. By using this version of HtmlFlow we observe code that is very similar to the current version, HtmlFlow 3, which uses `xmlet`, as shown in Listing 3.1. The most notable issue while using HtmlFlow 1 is the lack of the whole HTML syntax and poor navigation on the generated element tree.

```
1  HtmlView<?> taskView = new HtmlView<>();
2  taskView
3      .head()
4      .title("Task Details")
5      .linkCss("https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/
           bootstrap.min.css");
6  taskView
7      .body().classAttr("container")
8      .heading(1, "Task Details")
9      .hr()
10     .div()
11     .text("Title: ").text("ISEL MPD project")
12     .br()
13     .text("Description: ").text("A Java library for serializing objects
           in HTML.")
14     .br()
15     .text("Priority: ").text("HIGH");
```

Listing 3.1: HtmlFlow Version 1 Code Example

3.2.2 J2html

J2html[10] is a Java library used to write HTML. This solution does not verify the specification rules of the HTML language either at compile time or at runtime, which is a major downside. But on the other hand it removes the requirement of having text files to define templates by defining the templates within the Java language. It also provides support for the use of most of the HTML language, which is probably the reason why it has more garnered more attention than HtmlFlow 1. This library also shows that the issue we are trying to solve with the HtmlFlow is relevant since this library is quite popular, currently having 442 stars on the project Github page[1]. In Listing 3.2 we show the required code using J2html to generate the example template defined in Listing 1.4. Regarding its use, it is simple to use since it has a similar syntax to HTML, which makes it easily understandable. The way it uses parameters to pass children elements also helps keep track of depth in the element tree.

[1] https://github.com/tipsy/j2html

```
Student student = new Student(39378, "Luis Duarte");

String document =
    html(
        body(
            ul(
                li(student.getName()),
                li(student.getNumber())
            )
        )
    ).render();
```

Listing 3.2: J2html Code Example

3.2.3 Rocker

The Rocker[22] library is very different from the two libraries presented before. Its approach is at is core very similar to the classic *template engine* solution since it still uses a textual file to define the template. But contrary to the classic *template engines* the template file is not used at run-time. This solution uses the textual template file to automatically generate a Java class to replicate that specific template in the Java language. This means that instead of resorting to loading the template defined in a text file at run time it uses the automatically generated class to generate the final document, by combining the static information present in the class with the received input. This is very important, by two distinct reasons. The first reason is that this solution can validate the type of the *context objects* used to create the template at compile time. The second reason is that this solution has very good performance due to all the static parts of the template being `hardcoded` into the Java class that defines the specific template. This was by far the best competitor with `xmlet` regarding performance. The biggest downside of this solution is that it does not verify the HTML language rules or even well formed XML documents. Regarding its use, Rocker is a bit more complex. It has three distinct aspects: the template, the generated Java class and the Java code needed to render it. In Listing 3.3 we show the Rocker template which is required to replicate the template of Listing 1.4.

```
1 @import com.mitchellbosecke.benchmark.model.Student
2 @args (Student student)
3 <html>
4     <body>
5         <ul>
6             <li>
7                 @student.getName()
8             </li>
9             <li>
10                @student.getNumber()
11            </li>
12        </ul>
13    </body>
14 </html>
```

Listing 3.3: Rocker Template Example

After defining the template and compiling the project, Rocker generates a Java class based on this template, i.e. the template defined in Listing 3.3. The generated class is shown in Listing 3.4. The class presented here is simplified, but it still gives a good overview of what Rocker does. It defines a *context object* of type `Student`, in line 4 and receives a value for it in line 11 of Listing 3.4. The template itself is separated into `Strings`, e.g. in the current example we have two placeholders the `@student.getName()` and `@student.getNumber()`, so Rocker stores three `Strings` in static variables: the `String` before the first placeholder, i.e. field `PLAIN_TEXT_0_0` in line 34 of Listing 3.4, the `String` in between placeholders, i.e. field `PLAIN_TEXT_1_0` in line 36 of Listing 3.4, and lastly the `String` after the second placeholder, i.e. field `PLAIN_TEXT_2_0` in line 38 of Listing 3.4. The `__doRender` method, line 26 of Listing 3.4, joins the different template `Strings` with the placeholders.

```
1  public class studentTemplate extends DefaultRockerModel {
2
3    private Student student;
4
5    public studentTemplate student(Student student) {
6      this.student = student;
7      return this;
8    }
9
10   static public studentTemplate template(Student student) {
11     return new studentTemplate().student(student);
12   }
13
14   static public class Template extends DefaultRockerTemplate {
15
16     protected final Student student;
17
18     public Template(studentTemplate model) {
19       super(model);
20       this.student = model.student();
21     }
22
23     @Override
24     protected void __doRender() throws IOException,RenderingException{
25       __internal.writeValue(PLAIN_TEXT_0_0);
26       __internal.renderValue(student.getName(), false);
27       __internal.writeValue(PLAIN_TEXT_1_0);
28       __internal.renderValue(student.getNumber(), false);
29       __internal.writeValue(PLAIN_TEXT_2_0);
30     }
31   }
32
33   private static class PlainText {
34     static private final String PLAIN_TEXT_0_0 =
35     "\n<html>\n    <body>\n        <ul>\n            <li>\n          ";
36     static private final String PLAIN_TEXT_1_0 =
37     "\n            </li>\n            <li>\n                ";
38     static private final String PLAIN_TEXT_2_0 =
39     "\n            </li>\n        </ul>\n    </body>\n</html>";
40   }
41 }
```

Listing 3.4: Rocker Java Class Example

Lastly, the code required to define the *context object* and render the template is the shown in Listing 3.5. Here the code is quite simple, the only requirement is to pass a valid *context object*, which is validated by the Java compiler since the *context object* type is defined in the textual template file, line 1 of Listing 3.3, which Rocker uses to as parameter to the `template` method in generated the class, shown in Listing 3.4.

```
1 String document =
2     templates.studentTemplate
3         .template(new Student(39378, "Luis Duarte"))
4         .render()
5         .toString();
```

Listing 3.5: Rocker Use Example

3.2.4 KotlinX Html

Kotlin[14] is a programming language that runs on the *Java Virtual Machine* (JVM). The language main objective is to create an inter-operative language between Java, Android and browser applications. Its syntax is not compatible with the standard Java syntax but both languages are inter-operable. The main reasons to use this language is that it heavily reduces the amount of textual information needed to create code by using type inference and other techniques.

Kotlin is relevant to this project since one of his children projects, KotlinX Html, defines a DSL for the HTML language. The solution KotlinX Html provides is quite similar to what the `xmlet` will provide in its use case.

- Elements - The generated Kotlin DSL will guarantee that each element only contains the elements and attributes allowed as stated in the HTML5 XSD document. This is achieved by using type inference and the language compiler.
- Attributes - The possible values for restricted attributes values are not verified.
- Template - The template is embedded within the Kotlin language, removing the textual template files.
- Flexibility - Allows the usage of the Kotlin syntax to define templates, which is richer than the regular *template engine* syntax.

- Complexity - Removes language heterogeneity, the programmer only programs in Kotlin.

KotlinX[15] HTML DSL is the most similar solution to `xmlet`. The only difference is that `xmlet` takes advantage of the attributes restrictions present in the XSD document in order to increase the verifications that are performed on the HTML documents that are generated by the generated *fluent interfaces*. Both solutions also use the Visitor pattern in order to abstract themselves from the concrete usage of the DSL. The main difference between KotlinX Html and `xmlet` is performance, the Kotlin DSL is slow compared to other *template engines* solutions. Having a library as popular as the Kotlin HTML DSL, currently with 518 stars on its Github page[2], providing a solution so similar to `xmlet` also shows that the approach makes sense and tackles a real world problem. To use it, we must start by creating an HTML document, and then we can start to add elements to it. An example is shown in Listing 3.6, defining the same template defined in Listing 1.4. Using it feels pretty straightforward, its quite similar to the code we get while using J2html, with the advantage of guaranteeing the implementation of the HTML syntax rules.

```
1  val student = Student(39378, "Luis Duarte")
2
3  val document = createHTMLDocument()
4      .html {
5          body {
6              ul {
7                  li { student.name }
8                  li { student.number }
9              }
10         }
11     }.serialize(false)
```

Listing 3.6: Kotlin Template Example

[2] https://github.com/Kotlin/kotlinx.html

3.2.5 HtmlFlow 3

After developing `xmlet` and adapting the HtmlFlow library to use it some characteristics changed. This version of HtmlFlow that uses `xmlet` will be referred as HtmlFlow 3 from now on. The safety aspects of HtmlFlow 1 are kept since the general idea for the solution is kept with the usage of the HtmlApi generated by `xmlet`. Regarding the negative aspects of HtmlFlow 1, three of them were solved:

- Small language subset - Solved by using the automatically generated HtmlApi, which defines the whole HTML language within the Java language.
- Attribute value validation - The HtmlApi validates every attribute value based on the restrictions defined for that respective attribute in the HTML XSD document.
- Maintainability - Since it uses an automatically generated DSL if any change occurs in the HTML language specification the only change needed is to generate a new DSL based on the new language rules defined in the XSD file.

By using `xmlet` the HtmlFlow library was also able to improve its performance. With the mechanics created by the usage of `xmlet` it is now possible to replicate the performance improvements of the Rocker solution. An example of its usage was already shown in Listing 1.6. Its syntax ends up being similar to the other solutions presented in this chapter, with the most notable difference being the fact that it allows to use Java functions to create template, as shown on line 4 of Listing 1.6.

3.2.6 Feature Comparison

To have a better overview on all the previously presented solutions we will now show a table that has a list of most important features and which solutions implements them.

	J2Html	Rocker	KotlinX	HtmlFlow*
Template Within Language	✓	*1	✓	✓ / ✓
Elements Validations	✗	✗	✓	✓ / ✓
Attribute Validations	✗	✗	✗	✗ / ✓
Fully Supports HTML	✗	✓	✓	✗ / ✓
Well-Formed Documents	✓	✗	✓	✓ / ✓
Maintainability	✗	✓	✓	✗ / ✓
Performance	✓	✓	✗	✗ / ✓

Table 3.1: Template Engines Feature Comparison
✗- Feature not present
✓- Feature present
HtmlFlow* - HtmlFlow 1 / HtmlFlow 3
*1 - Template class generated at compile time

As we can see in the Table 3.1, most of these solutions tend to move the template definition from the textual files to the current language syntax, in this case Java. This removes the *overhead* of loading the textual files and parsing them at runtime. Another feature that the different solutions share is that they all create well formed documents, apart from Rocker. The general problem that extends to all the solutions that previously existed is the lack of validations that enforce the HTML language rules. KotlinX Html is the solution that mostly resembles what `xmlet` pretends to implement but is heavily handicapped when it comes to performance, being one of the worst in the benchmarks performed, which will be presented in Chapter 5.

4

Solution

This chapter will present `xmlet`, its different components and how they interact between them. Generating a Java *fluent interface* based on a XSD file includes two distinct tasks:

1. Parsing the information from the XSD file;

2. Generating the *fluent interface* classes based on the resulting information of the previous task.

Those tasks are encompassed by two different projects, XsdParser, presented in Section 4.1, and XsdAsm, presented in Section 4.2. In this case the XsdAsm has a dependency to XsdParser.

The main use case of `xmlet` is the generation of a Java DSL for HTML. To that end, the HtmlApi, presented in Section 4.3.1, specifies how the XsdAsm project can be used, specifically using the HTML5 XSD file, to generate the Java HTML *fluent interface*.

Finally, the HtmlFlow 3 is responsible for establishing the output of a concrete HtmlApi usage. Some additional remarks regarding changes on the HtmlFlow library will be provided in Section 4.3.3.

4.1 XsdParser

XsdParser is a library that parses a XSD file into a list of Java objects. Each different XSD tag has a corresponding Java class and the attributes of a given XSD type are represented as fields of that class. All these classes derive from the same abstract class, `XsdAbstractElement`. All Java representations of the XSD elements follow the schema definition for XSD elements, referred in Section 3.1. For example, the `xsd:annotation` tag only allows `xsd:appinfo` and `xsd:documentation` as children nodes, and can also have an attribute named `id`, therefore XsdParser has the following class as shown in Listing 4.1.

```
1 public class XsdAnnotation extends XsdAbstractElement {
2
3     private String id;
4     private List<XsdAppInfo> appInfoList = new ArrayList<>();
5     private List<XsdDocumentation> documentations = new ArrayList<>();
6
7     // (...)
8 }
```

Listing 4.1: Simplified Version of the Generated XsdAnnotation Class

4.1.1 Parsing Strategy

The first step of this library is handling the XSD file. The Java language has no built in library that parses XSD files, so we needed to look for other options. The main libraries found that address this problem were *Document Object Model* (DOM) and *Simple Application Programming Interface for eXtensive Markup Language* (SAX). After evaluating the pros and cons of those libraries the choice ended up being DOM, since a XSD file is a tree of XML elements. This choice was based mostly on the fact that SAX is an event driven parser and DOM is a tree based parser, which is more adequate for the present issue. DOM is a library that maps HTML, *eXtensive HyperText Markup Language* (XHTML) and XML files into a tree structure composed by multiple elements, also named nodes. This is exactly what XsdParser requires to obtain all the information from the XSD files, which is described in XML.

This means that XsdParser uses DOM to parse the XSD file and obtain its root element, a `xs:schema` node, performing a single read on the XSD file, avoiding multiple reads, which are less efficient (Listing 4.2).

```
1 private Node getSchemaNode(String filePath)
2     throws IOException, SAXException, ParserConfigurationException {
3     DocumentBuilderFactory dbFactory =
4                                 DocumentBuilderFactory.newInstance();
5     DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
6     Document doc = dBuilder.parse(xsdFile);    //Parses the XSD file.
7
8     // Obtains the first node of the document, which
9     // should be the xs:schema node.
10    return doc.getFirstChild();
11 }
```

Listing 4.2: DOM Document Parsing

After obtaining the root node of the XSD file the XsdParser verifies if that node is a `XsdSchema` node as shown in Listing 4.3. If that is the case it proceeds by performing the `parse` function of the `XsdSchema` class.

```
1 Node schemaNode = getSchemaNode(filePath);
2
3 if (isXsdSchema(schemaNode)){
4     XsdSchema.parse(this, schemaNode);
5 }
```

Listing 4.3: XsdParser Parsing the XsdSchema Node, which triggers the parsing of the whole XSD document

The `XsdSchema` element `parse` function, shown in line 13 of Listing 4.4, converts the `Node` attributes into a `Map` object, which `XsdSchema` receives in the constructor. Each class extracts their field information from the received attribute `Map` object in their constructor methods, e.g. `XsdSchema` constructor in line 2 of Listing 4.4. To guarantee that the information parsed by the classes is compliant with the XSD syntax we perform multiple validations. To validate the possible values for any given attribute, e.g. the `finalDefault` attribute from the `xsd:schema` element, we use `Enum` classes. Any parsed value that is meant to be assigned to one of this `Enum` variables has its content verified to assert if the received value belongs to the possible values for that attribute. In lines 5 and 6 of Listing 4.4 we can see this behaviour, we first obtain the `finalDefault` attribute value from the `Map` object and then we invoke `belongsToEnum` passing an instance of the `FinalDefaultEnum` type and the parsed value. The `belongsToEnum` method will assert if the received value is present in the possible values for the received `Enum` class and if present it returns the `Enum` instance that represents the received value, otherwise it will throw an exception.

```
1  public class XsdSchema extends XsdAnnotatedElements {
2     private XsdSchema(XsdParser parser, Map<String, String> attributes){
3        super(parser, attributes);
4
5        String finalDef = attributes.getOrDefault("finalDefault", "");
6        this.finalDefault = belongsToEnum(FinalDefaultEnum.instance,
             finalDef);
7
8        this.xmlns = attributes.getOrDefault(XMLNS, xmlns);
9
10       // Similar code used for the remaining attributes.
11    }
12
13    public static ReferenceBase parse(XsdParser parser, Node node) {
14       NamedNodeMap nodeAttributes = node.getAttributes();
15       Map<String, String> attributes = convertNodeMap(nodeAttributes);
16
17       return xsdParseSkeleton(node, new XsdSchema(parser, attributes));
18    }
19 }
```

Listing 4.4: XsdSchema Extracting Information from the received Node

The parsing of `XsdSchema` continues by parsing its children nodes. To parse children elements of any given `XsdAbstractElement` type we have the `xsdParseSkeleton` function present in the `XsdAbstractElement` class. This function will iterate in all the children of a given node, line 5 of Listing 4.7, invoke the respective `parse` function of each children, line 17/18 of Listing 4.7, and then notify the parent element, using the Visitor pattern[9], line 20 of Listing 4.7.

In XsdParser the Visitor pattern is used to ensure that each concrete element defines different behaviours for different types of children. This provides good flexibility for implementing certain XSD syntax restrictions, e.g. the element `complexContent` can only receive `extension` and `restriction` elements as children.

```
1 <xsd:complexContent>
2     <xsd:restriction base="xsd:string"/>
3     <xsd:attribute name="dummy"/>
4 </xsd:complexContent>
```

Listing 4.5: ComplexContent element with Restriction and Attribute children

```
1  class XsdComplexContentVisitor extends XsdAnnotatedElementsVisitor {
2
3      private final XsdComplexContent owner;
4
5      // ...
6
7      @Override
8      public void visit(XsdRestriction element) {
9          owner.setRestriction(ReferenceBase.createFromXsd(element));
10     }
11
12     @Override
13     public void visit(XsdExtension element) {
14         owner.setExtension(ReferenceBase.createFromXsd(element));
15     }
16 }
```

Listing 4.6: XsdComplexContentVisitor Class

Parsing the XSD of Listing 4.5, will result in the `xsd:complexContent` element being parsed, followed by the parsing of its children, i.e. `xsd:restriction` and `xsd:attribute`. When the `xsd:restriction` element is parsed the resulting instance, i.e. the `childElement` variable in line 17 of Listing 4.7, accepts his parents Visitor instance, in line 20 of Listing 4.7. The `accept` method of the `XsdRestriction` type will invoke the `visit` method which receives the type `XsdRestriction`, which means that, in this case, it will invoke the `visit` method of line 8 of Listing 4.6. In the `XsdComplexContentVisitor` of Listing 4.6 we can see that it only defines behaviour for the two children types it supports, `XsdRestriction` and `XsdExtension`. This means that if a XSD file defines an invalid element for the current context, such as the `xsd:attribute` of Listing 4.5 which is not allowed as children of `xsd:complexContent`, the parser will just ignore that parsed element, since there is no behaviour defined for `XsdAttribute` elements on the `XsdComplexContentVisitor`.

```
ReferenceBase xsdParseSkeleton(Node node, XsdAbstractElement element){
    XsdParser parser = element.getParser();
    Node child = node.getFirstChild();

    while (child != null) { //Iterates in all children from node.
        //Only parses element nodes, ignoring comments and text nodes.
        if (child.getNodeType() == Node.ELEMENT_NODE) {
            String nodeName = child.getNodeName();

            //Searches on a mapper for a parsing functions
            //for the respective type.
            BiFunction<XsdParser, Node, ReferenceBase> parserFunction =
                XsdParser.getParseMappers().get(nodeName);

            //Applies the parsing functions, if any, and notifies
            //the parent objects Visitor to the newly created object.
            if (parserFunction != null){
                XsdAbstractElement childElement =
                  parserFunction.apply(parser, child).getElement();

                childElement.accept(element.getVisitor());
                childElement.validateSchemaRules();
            }
        }

        child = child.getNextSibling(); //Moves on to the next sibling.
    }

    ReferenceBase wrappedElement= ReferenceBase.createFromXsd(element);
    parser.addParsedElement(wrappedElement);
    return wrappedElement;
}
```

Listing 4.7: XsdParseSkeleton Function - Parsing Children From a Node

Having each element parsing their own children means that the only requirement for parsing a XSD document will be parsing its root element, that should always be a `XsdSchema`.

Based on the explanation provided above, we will give a more detailed description about the parsing process made by `XsdParser` using a small concrete example extracted from the HTML XSD file, present in Listing 4.8.

```
1 <xs:schema>
2     <xs:element name="html">
3         <!-- -->
4     </xs:element>
5 </xs:schema>
```

Listing 4.8: Parsing Concrete Example

Step 1 - DOM parsing:

The parsing starts with the DOM library parsing the code, Listing 4.8, which returns the `xs:schema` node, i.e. schemaNode in Listing 4.3. XsdParser verifies if the node is in fact a `xs:schema` node and after verifying that in fact it is, it invokes the `XsdSchema parse` function (line 19 of Listing 4.4).

Step 2 - `XsdSchema` Attribute Parsing:

The `XsdSchema parse` function receives the `Node` object and converts it to a `Map` object (line 21 of Listing 4.4). The map object is then passed to the `XsdSchema` constructor, line 2 of Listing 4.4, which will extract the information from the `Map` object to the class fields.

Step 3 - `XsdSchema` Children:

Parses the children of the `XsdSchema`. The `xsdParseSkeleton` function, Listing 4.7, is called (line 17 of Listing 4.4) and starts to iterate the `xs:schema` node children, which, in this case, is a node list containing a single element, the `xs:element` node.

Step 4 - `XsdElement` Attribute Parsing:

The parsing of the `xs:element` node is similar to `xs:schema`, it extracts the attribute information from its respective node in its constructor.

Step 5 - `XsdSchema` Visitor Notification:

After parsing the `xs:element` node the previously created `XsdSchema` object is notified using the Visitor pattern. This notification informs the `XsdSchema` object that it contains the newly created `XsdElement` object. The `XsdSchema` should then act accordingly based on the type of the object received as its children, similar to the behaviour shown in `XsdComplexContentVisitor` of Listing 4.6.

4.1.2 Reference solving

After the parsing process described previously, there is still an issue to solve regarding the existing references in the XSD schema definition. In XSD files the usage of the `ref` attribute is frequent to avoid repetition of XML code. This generates two main problems when handling reference solving, the first one being the existence of elements with `ref` attributes referring non existent elements and the other being the replacement of the reference object by the referenced object when present. In order to effectively help resolve the referencing problem some wrapper classes were added. These wrapper classes contain the wrapped element and serve as a classifier for the wrapped element. The existing wrapper classes are as follow:

- `UnsolvedElement` - Wrapper class to each element that has a `ref` attribute.
- `ConcreteElement` - Wrapper class to each element that is present in the file.
- `NamedConcreteElement` - Wrapper class to each element that is present in the file and has a `name` attribute present.
- `ReferenceBase` - A common interface between `UnsolvedReference` and `ConcreteElement`.

Having these wrappers on the elements allow for a detailed filtering, which is helpful in the reference solving process. A concrete example of how this process works is in Listing 4.9.

```
1  <xsd:schema>
2      <!-- NamedConcreteType wrapping a XsdGroup -->
3      <xsd:group id="replacement" name="flowContent">
4          <!-- (...) -->
5      </xsd:group>
6
7      <!-- ConcreteElement wrapping a XsdChoice -->
8      <xsd:choice>
9          <!-- UnsolvedReference wrapping a XsdGroup -->
10         <xsd:group id="toBeReplaced" ref="flowContent"/>
11     </xsd:choice>
12 </xsd:schema>
```

Listing 4.9: Reference Solving Example

In this short example we have a `XsdChoice` element, line 8 of Listing 4.9, that contains a `XsdGroup` element with a `ref` attribute, line 10 of Listing 4.9. At this point the `XsdChoice` is contained in a `ConcreteElement` object and the `XsdGroup` is contained in a `UnsolvedReference` object. When replacing the `UnsolvedReference` objects the `XsdGroup` with the `ref` attribute, of line 10 of Listing 4.9, is going to be replaced by a copy of the already parsed `XsdGroup` with the `name` attribute, of line 3 of Listing 4.9, which is contained into a `NamedConcreteElement` object. This is achieved by accessing the parent of the element, in this case accessing the parent of the `XsdGroup` with the `ref` attribute of line 10 of Listing 4.9, which is the `XsdChoice` element. After accessing the `XsdChoice` object we can replace the `XsdGroup` of line 10 with the `XsdGroup` of line 3 of Listing 4.9.

To resume this process, there are three steps:

- Step 1 - Obtain all the `NamedConcreteElement` objects since they may, or may not, be referenced by an existing `UnsolvedReference` object.

- Step 2 - Obtain all the `UnsolvedReference` objects and iterate them to perform a lookup search on the `NamedConcreteElement` objects obtained in Step 1. This is achieved by comparing the value present in the `UnsolvedReference ref` attribute with the `NamedConcreteElement name` attribute.

- Step 3 - If a match is found then XsdParser performs a copy of the object wrapped by the `NamedConcreteElement` and replaces the element wrapped in the `UnsolvedReference` object that served as a placeholder.

Having created these classes it is expected that at the end of a successful file parsing only `ConcreteElement` and/or `NamedConcreteElement` objects remain. In case there are any remainder `UnsolvedReference` objects the programmer can query the parser, using the function `getUnsolvedReferences` of the XsdParser class, to discover which elements are missing and where were they used. The programmer can then correct the missing elements by adding them to the XSD file and repeat the parsing process or just acknowledge that those elements are missing.

4.1.3 Validations

As it was already referred in Section 4.1.1 the parser uses some strategies to validate the rules of the XSD language. We already referred the usage of `Enum` classes for attribute values that have a set of possible values but there are more validations. This solution also validates the types of data received, e.g. validating if a given attribute is a positive `Integer` value. There are more intricate restrictions relating to the organization between elements, for example the `xsd:element` element is not allowed to have a `ref` attribute value if the `xsd:element` is a direct child of the top-level `xsd:schema` element. All those rules were extracted from the XSD language standard and each time a concrete element is created the respective rules are verified when the `validateSchemaRules` method is called, in line 21 of Listing 4.7.

Each time any of these rules is violated a `ParsingException` is thrown containing a message detailing the rule that was violated, either being an attribute that does not match its expected type, an attribute that has a value that is not within the possible values for that attribute or any other more complex rule of the XSD language. With this strategy the user of the XsdParser solution has the information needed to fix the existing problems in the XSD file.

4.2 XsdAsm

XsdAsm is a library dedicated to generate a Java *fluent interface* based on a XSD file. It uses the previously introduced XsdParser library to parse the XSD file contents into a list of Java objects that XsdAsm will use to obtain the information needed to generate the correspondent classes.

To generate classes this library also uses the ASM[2] library, which is a library that provides a Java interface that allows *bytecode* manipulation providing methods for creating classes, methods, etc. There were other alternatives to the ASM library but most of them are simply libraries that were built on top of ASM to simplify its usage. It supports the creation of Java classes up until Java 11 and is still maintained, the most recent version, 7 beta, was release in 29 September of 2018. ASM also has some tools to help the new programmers understand how the library works. These tools help the programmers to learn faster how the code generation works and allows to generate more complex code. In Listing 4.10 we present a class that is used as an example of code generation. It is a simple class,

with a field and a method. The ASM library provides a tool, `ASMifier`, which receives a `.class` file and returns the ASM code needed to generate it, as shown in Listing 4.11.

```
1 public class SumExample {
2
3     private int sum;
4
5     void setSum(int a, int b){
6         sum = a + b;
7     }
8 }
```

Listing 4.10: ASM Example - Code Generation Objective

```
1 ClassWriter classWriter = new ClassWriter(0);
2
3 classWriter.visit(V9, ACC_PUBLIC + ACC_FINAL + ACC_SUPER,
4   "Samples/HTML/SumExample", null, "java/lang/Object", null);
5
6 FieldVisitor fieldVisitor =
7   classWriter.visitField(ACC_PRIVATE, "sum", "I", null, null);
8 fieldVisitor.visitEnd();
9
10 MethodVisitor methodVisitor =
11   classWriter.visitMethod(0, "setSum", "(II)V", null, null);
12 methodVisitor.visitCode();
13 methodVisitor.visitVarInsn(ALOAD, 0);
14 methodVisitor.visitVarInsn(ILOAD, 1);
15 methodVisitor.visitVarInsn(ILOAD, 2);
16 methodVisitor.visitInsn(IADD);
17 methodVisitor.visitFieldInsn(PUTFIELD, "Samples/HTML/SumExample",
18   "sum", "I");
19 methodVisitor.visitInsn(RETURN);
20 methodVisitor.visitMaxs(3, 3);
21 methodVisitor.visitEnd();
22
23 classWriter.visitEnd();
24
25 writeByteArrayToFile(classWriter.toByteArray());
```

Listing 4.11: ASM Example - Required Code

The strategy while creating `xmlet` was to manually create classes that represent a certain type of class that XsdAsm will need to generate, such as element and

attribute classes. By using the `ASMifier` tool with those template-like classes the programming process was expedited.

There are other ways to generate code, such as tools that generate source code, which can then be compiled into the binary class files. Those tools were not used since we started by using this method, i.e. directly generating bytecodes, and never ran into any type of issue that made us consider explore other options.

4.2.1 Supporting Infrastructure

To support the foundations of the XSD language there is a common infrastructure in every *fluent interface* generated by this project. This infrastructure is composed by a set of classes, which is divided into three different groups:

Element classes:

- `Element` - An interface that every class generated based on a XSD `xsd:element` implements.
- `AbstractElement` - An abstract class that implements most of methods of the `Element` interface. All classes generated based on a XSD `xsd:element` extend this class.

Attribute classes:

- `Attribute` - An interface that every class generated based on a XSD `xsd:attribute` implements.
- `BaseAttribute` - A class that implements the `Attribute` interface. All classes generated based on a XSD `xsd:attribute` extend this class.

Visitor class:

- `ElementVisitor` - An abstract class that defines `visit` methods for all the generated element and attribute classes, which can be visited with the Visitor pattern. All the implemented methods point to a single method, e.g. every `visit` method related with element classes invoke a `visitElement` method. This behaviour aims to reduce the amount of code needed to create concrete implementations of this class.

Taking in consideration those classes, a very simplistic *fluent interface* could be represented with the class diagram shown in Figure 4.1. In this example we have two generated classes, the `Html` class, which extends `AbstractElement` and the `AttrManifest` class, which extends `BaseAttribute`. The classes above the line represent all the classes shared by all `xmlet` *fluent interfaces*. The classes below the line represent the classes generated based on the XSD file contents.

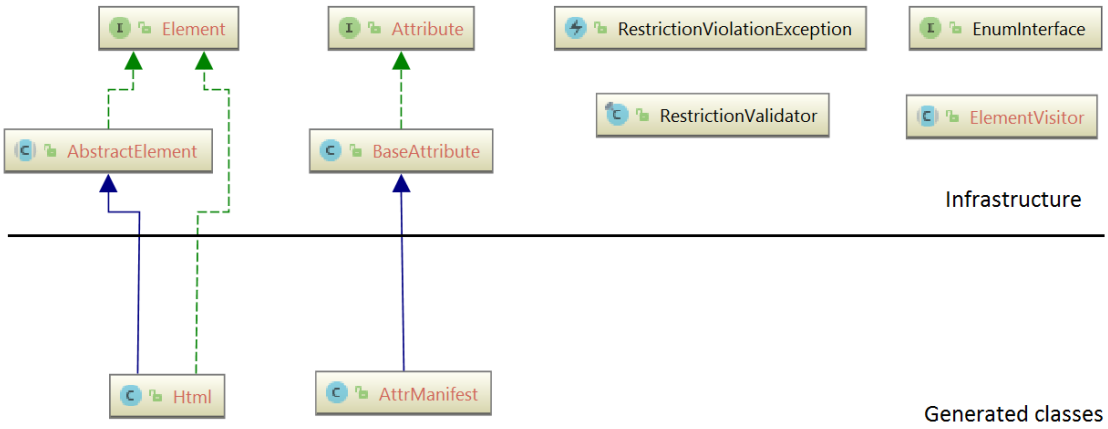


Figure 4.1: Fluent Interfaces - The Supporting Infrastructure

4.2.2 Code Generation Strategy

As we already presented before in the Section 2.2, this solution focus on how the code is organized instead of making complex code. All the methods present in the generated classes have very low complexity, mainly adding information to the element children or to the attribute list. To reduce repeated code many interfaces with default methods are created so different classes can implement them and reuse the code. The complexity of the generated code is mostly present in the `AbstractElement` class, which implements most of the `Element` interface methods. Another very important aspect of the generated classes is the extensive use of *type arguments*, also known as generics, which allows the navigation in the element tree while maintaining type information, which is essential to guarantee the specific language restrictions.

4.2.3 Type Parameters

As this solution was designed an objective became clear, the generated *fluent interface* should be easily navigable. This is crucial to provide a good user experience while creating templates through the `xmlet` *fluent interfaces*. There are two

main aspects, the *fluent interface* should be easily navigable and always implement the concrete language restrictions. We tackle this issue through the use of *type parameters*, which allow us to keep track of the tree structure of the elements that are being created and keep adding elements, or moving up in the tree structure without loosing the type information of the parent. In Listing 4.12 we can observe how the type arguments work.

```
1 Html<Element> html = new Html<>();
2 Body<Html<Element>> body = html.body();
3
4 P<Header<Body<Html<Element>>>> p1 = body.header().p();
5 P<Div<Body<Html<Element>>>> p2 = body.div().p();
6
7 Header<Body<Html<Element>>> header = p1.__();
8 Div<Body<Html<Element>>> div = p2.__();
```

Listing 4.12: Example of the Explicit Use of Type Arguments

When we create the `Html` element we should indicate that he has a parent, for consistency. Then, as we add elements such as `Body` we automatically return the recently created `Body` element, but with parent information that indicates that this `Body` instance is descendant of an `Html` element. After that, we create two distinct `P` elements, `p1`, which has an `Header` parent, and `p2`, which has a `Div` parent. This information is reflected in the type of both variables, in line 4 and 5 of Listing 4.12 respectively. Lastly, we can invoke the ___ method, line 7 and 8 of Listing 4.12, which returns the current element parent, and observe that each `P` instance returns its respective parent object, with the correct type.

In the example presented in Listing 4.12 the usage of the *fluent interface* might seem to be excessive verbose to define a simple HTML document. For specific purposes it might be needed to extract variables, but in the most common usage of the *fluent interface* the code should be similar to Listing 4.13.

```
1 new Html<>()
2     .body()
3         .header()
4             .p().__()
5         .__()
6         .div()
7             .p().__();
```

Listing 4.13: Example of the Implicit Use of Type Arguments

To provide a better understanding on how this works we need to showcase three distinct classes. First we have the `AbstractElement` class, Listing 4.15, which is the class from where all classes generated based on a XSD `xsd:element` derive. This class receives two *type parameters*:

- **T** - Represents the type of the concrete element;
- **Z** - Represents the type of the parent of the concrete element.

In the __ method, shown in line 8 of Listing 4.15, the parent of any deriving class is returned. The type information is kept since the method returns `Z`, the *type parameter* that is the type of the parent of the deriving class, as shown in lines 7 and 8 of Listing 4.12.

```
1 class AbstractElement<T extends Element, Z extends Element>
2   protected Z parent;
3
4     protected AbstractElement(Z parent) {
5         this.parent = parent;
6     }
7
8   public Z __() {
9     return this.parent;
10  }
11
12  // (...)
13 }
```

Listing 4.14: AbstractElement Class Type Arguments

The second class is the `Table` class, which represents the `table` XSD element present on the HTML *fluent interface* generated by `xmlet`. It has a single *type parameter*, Z, which represents the type of its parent. It extends `AbstractElement` and therefore indicates that his type is `Table<Z>` and its parent type is `Z`. Any interface implemented by a class generated from a XSD `xsd:element`, such as the `Table` class, should receive the same type information as the `AbstractElement` class, as shown with `TableChoice0`. Regarding the `attrBorder` method, it indicates that it returns the exact same type, since it returns the `this` object, i.e. `Table<Z>`.

```
1 class Table<Z extends Element> extends AbstractElement<Html<Z>, Z>
2                                implements TableChoice0<Html<Z>, Z>
3     // (...)
4
5     public Table<Z> attrBorder(EnumBorderType attrBorder) {
6         // ...
7     }
8 }
```

Listing 4.15: Html Class Type Arguments

As the third class we have the `TableChoice0` interface, which has methods for each kind of element that is allowed as children in `table` elements. It should receive the same *type parameters* as `AbstractElement`. In the current case, if an instance of `Table<Element>` invokes the `tbody` method, line 2 of Listing 4.16, the type returned would be `Tbody<Table<Element>>` since the `T` type of `TableChoice0` is `Table<Z>` when the type `Table` implements it.

```
1 interface TableChoice0<T extends Element<T, Z>, Z extends Element>
                                                    extends Element<T, Z> {
2     default Tbody<T> tbody() {
3         // ...
4     }
5 }
```

Listing 4.16: TableChoice0 Interface Type Arguments

4.2.4 Restriction Validation

In the description of any given XSD file there are many restrictions in the way the elements are contained within each other and which attributes are allowed. To reflect those restrictions to Java language there are two alternatives, validation in runtime or in compile time. This library tries to validate most of the restrictions in compile time, as shown above by the way classes are created. But some restrictions cannot be validated in compile time, an example of this is the following attribute with a restriction shown in Listing 4.17.

```
1  <xs:schema>
2      <xs:element name="testElement">
3          <xs:complexType>
4              <xs:attribute name="intList" type="valuelist"/>
5          </xs:complexType>
6      </xs:element>
7
8      <xs:simpleType name="valuelist">
9          <xs:restriction>
10             <xs:maxLength value="5"/>
11             <xs:minLength value="1"/>
12         </xs:restriction>
13         <xs:list itemType="xsd:int"/>
14     </xs:simpleType>
15 </xs:schema>
```

Listing 4.17: Restrictions Example XSD

In this example, Listing 4.17, we have an element, i.e. `testElement` on line 2, which has an attribute called `intList`, on line 4. This attribute has some restrictions, it is represented by a `xs:list`, the list elements have the `xsd:int` type and its element count should be between 1 and 5, restriction element on line 9 of Listing 4.17. Transporting this example to the Java language will result in the following class shown on Listing 4.18.

```
1 public class AttrIntList extends BaseAttribute<List<Integer>> {
2     public AttrIntList(List<Integer> list) {
3         super(list);
4     }
5 }
```

Listing 4.18: Attribute Class Receiving a List

But with this solution the `xs:maxLength` and `xs:minLength` values are ignored. To solve this problem the existing restrictions in any given attribute are *hardcoded* in the class constructor, which invokes methods present in the `RestrictionValidator` class that validate each type of restriction, e.g. `xs:maxLength` and `xs:minLength`. The values present in the restrictions on the XSD document are *hardcoded* in the *bytecodes* and help validate each attribute object that is created. This results in the generation of a constructor as shown in Listing 4.19.

```
public class AttrIntList extends BaseAttribute<List<Integer>> {
    public AttrIntList(List<Integer> attrValue) {
        super(attrValue, "intlist");
        RestrictionValidator.validateMaxLength(5, attrValue);
        RestrictionValidator.validateMinLength(1, attrValue);
    }
}
```

Listing 4.19: Attribute Constructor Enforcing Restrictions

There are a total of thirteen different restrictions on the XSD language. The `RestrictionValidator` class is a class with static methods that allow to validate most of those restrictions, the only restrictions that are not validated by this class are `xsd:enumeration` restrictions, which are already validated by the usage of `Enum` classes and `xsd:whitespace` since it represents an indication instead of an actual restriction on the language. In Listing 4.20 we can observe how simple is to validate the `xs:maxLength` and `xs:minLength` restrictions that were used in the previous example. All the methods work in the exact same way, a condition is verified and if the verification fails it will throw a `RestrictionViolationException` with a message describing the nature of the violated restriction.

```
public class RestrictionValidator {
    public static void validateMaxLength(int maxLength, List list){
        if (list.size() > maxLength){
            throw new RestrictionViolationException("Violation of
                maxLength restriction");
        }
    }

    public static void validateMinLength(int minLength, List list){
        if (list.size() < minLength){
            throw new RestrictionViolationException("Violation of
                minLength restriction");
        }
    }
}
```

Listing 4.20: RestrictionValidator Class - The Validation Methods

4.2.4.1 Enumerations

Regarding restrictions there is one that can be enforced at compile time, the `xs:enumeration`. To obtain that validation at compile time the XsdAsm library generates `Enum` classes that contain all the values indicated in the `xs:enumeration` elements. In the following example, Listing 4.21, we have an attribute with three possible values: `command`, `checkbox` and `radio`.

```
<xs:attribute name="type">
    <xs:simpleType>
        <xs:restriction base="xsd:string">
            <xs:enumeration value="command" />
            <xs:enumeration value="checkbox" />
            <xs:enumeration value="radio" />
        </xs:restriction>
    </xs:simpleType>
</xs:attribute>
```

Listing 4.21: Example of an Enumeration in XSD Definition

This results in the creation of an `Enum` class, `EnumTypeCommand`, presented in Listing 4.22. The attribute class will then receive an instance of `EnumTypeCommand`, ensuring that only allowed values are used (Listing 4.23).

```
public enum EnumTypeCommand {
    COMMAND(String.valueOf("command")),
    CHECKBOX(String.valueOf("checkbox")),
    RADIO(String.valueOf("radio"))
}
```

Listing 4.22: Example of a Generated Enumeration Class

```
public class AttrTypeEnumTypeCommand extends BaseAttribute<String> {
    public AttrTypeEnumTypeCommand(EnumTypeCommand attrValue) {
        super(attrValue.getValue());
    }
}
```

Listing 4.23: Attribute Receiving An Enumeration Instance

4.2.5 Element Binding

To support the definition of reusable templates the `Element` and `AbstractElement` classes were modified to support binders. This allows programmers to postpone the addition of information to the defined element tree. An example is presented in Listing 4.24 using the HTML5 *fluent interface*.

```
1  public class BinderExample{
2      public void bindExample(){
3          Html<Element> root = new Html<>()
4              .body()
5                  .table()
6                      .tr()
7                          .th()
8                              .text("Title")
9                          .__()
10                     .__()
11                     .<List<String>>binder((elem, list) ->
12                         list.forEach(tdValue ->
13                                 elem.tr().td().text(tdValue)
14                         )
15                     )
16                 .__()
17             .__()
18         .__();
19     }
20 }
```

Listing 4.24: Binder Usage Example

In this example we use the HTML language to create a document that contains a table with a title in the first row as a title header , i.e. `th()`. Regarding the values presented in the table, instead of having them inserted right away, it is possible to delay that insertion by postponing a function, as shown on line 11 of Listing 4.24, to be executed when the information is received, i.e. the `list` on line 11 of Listing 4.24. This is achieved by implementing an `ElementVisitor` that supports binding.

In Listing 4.25 we can observe how an `ElementVisitor` implementation that supports binders would work, while maintaining the default behaviour for the elements that are not bound, i.e. `else` clause in line 16 of Listing 4.25. If the element is bound to a function this implementation will clone the element, i.e. the `cloneElem` function in line 12 of Listing 4.25, and apply a model to the cloned

object, i.e. the model being a `List<String>` object following the example of shown Listing 4.24, effectively executing the function supplied in the previously shown `binder` method, i.e. line 11 of Listing 4.24. This function call will generate new children on the cloned `table` instance that will be iterated as if they belonged to the original element tree. This behaviour ensures that the original element tree is not affected since all these changes are performed in a clone of the bound element, meaning that the template can be reused.

```
1  public class CustomVisitor<R> implements ElementVisitor<R> {
2
3      private R model;
4
5      public CustomVisitor(R model){
6          this.model = model;
7      }
8
9      public <T extends Element> void sharedVisit(Element<T,?> element) {
10         // ...
11         if(element.isBound()) {
12             List<Element> children = element.cloneElem()
13                                             .bindTo(model)
14                                             .getChildren();
15             children.forEach( child -> child.accept(this));
16         } else {
17             element.getChildren().forEach(item -> item.accept(this));
18         }
19         // ...
20     }
21 }
```

Listing 4.25: Visitor with Binding Support

4.2.6 Using the Visitor Pattern

In the previous sections we presented how the *fluent interface* is generated and how it implements the language restrictions, but what can the *fluent interface* actually be used for? That is strictly up to the user of the generated *fluent interface*. To achieve this we use the `Visitor` pattern[9]. There are multiple `visit` methods that are invoked by the generated classes and the user can define the behaviour for each one of them by creating a concrete implementation of the

`ElementVisitor` class. This way the generated code delegates the responsibility of defining the output of the `fluent interface` usage. The generated `ElementVisitor` class defines four main `visit` methods, Listing 4.26:

- `sharedVisit(Element<T, ?> element)` - This method is called whenever a class generated based on a XSD `<xsd:element>` has its `accept` method called. By receiving the `Element` we have access to the element children and attributes.

- `visit(Text text)` - This method is called when the `accept` method of the special `Text` element is invoked.

- `visit(Comment comment)` - This method is called when the `accept` method of the special `Comment` element is invoked.

- `visit(TextFuction<R, U, ?> textFunction)` - This method is called when the `accept` method of the special `TextFunction` element is invoked.

```
1 public abstract class ElementVisitor<R> {
2     <T extends Element> void sharedVisit(Element<T, ?> element);
3
4     void visit(Text text);
5
6     void visit(Comment comment);
7
8     <U> void visit(TextFunction<R, U, ?> textFunction);
9 }
```

Listing 4.26: ElementVisitor Generated by XsdAsm - The Core Methods

Apart from these four main method we also create specific methods, as shown in Listing 4.27. These methods default behaviour is to invoke the main `sharedVisit(Element<T, ?> element)` method, but they can be redefined to perform a different action, providing the option of having a very simple implementation of only four methods or redefine all the methods for a concrete purpose for the respective DSL.

```
public abstract class ElementVisitor {
    // (...)

    default void visit(Html html) {
        this.sharedVisit(html);
    }
}
```

Listing 4.27: ElementVisitor Generated by XsdAsm - The Specific Methods

4.2.7 Performance - XsdAsmFaster

The `xmlet` developed two alternative solutions to generate *fluent interfaces*. The first solution that was implemented was XsdAsm, which generated a *fluent interface* that defined element and attribute classes. When interacting with those elements it was possible to add children or attributes that were stored in a data structure as seen by the implementation of `AbstractElement` and the snippet of the `Html` code present in Listing 4.28 and Listing 4.29, respectively.

```
abstract class AbstractElement<T extends Element, Z extends Element>
    implements Element<T, Z> {
    protected List<Element> children = new ArrayList();
    protected List<Attribute> attrs = new ArrayList();
    // (...)

    public <R extends Element> R addChild(R child) {
        this.children.add(child);
        return child;
    }
    public T addAttr(Attribute attribute) {
        this.attrs.add(attribute);
        return this.self();
    }
}
```

Listing 4.28: AbstractElement Class Generated by XsdAsm

```
class Html<Z extends Element> extends AbstractElement<Html<Z>, Z> {
   public void accept(ElementVisitor visitor) { visitor.visit(this); }

   public Html<Z> attrManifest(String attrManifest) {
      return (Html)this.addAttr(new AttrManifestString(attrManifest));
   }

   public Body<T> body() { return this.addChild(new Body(this)); }

   public Head<T> head() { return this.addChild(new Head(this)); }
}
```

Listing 4.29: Html Class Generated by XsdAsm

By using the solution generated by XsdAsm we ended up with a *fluent interface* that works in a two steps basis:

- Creating the `Element` tree - We need to create the element tree by adding all elements and attributes, as shown in Listing 4.30;
- Visiting the `Element` tree - We need to invoke the `accept` method of the root of the tree in order for the whole tree to be visited, as shown in Listing 4.31.

```
Html<Html> root = new Html<>();

root.head()
        .title()
            .text("Title")
        .__()
    .__()
    .body().attrClass("clear")
        .div()
          .h1()
            .text("H1 text")
          .__()
        .__()
    .__()
.__();
```

Listing 4.30: HTML5 Tree Creation using XsdAsm

```
CustomVisitor customVisitor = new CustomVisitor();
// root variable created in the previous Listing.
root.accept(customVisitor);
```

Listing 4.31: HTML5 Tree Visit using XsdAsm

Even though that this solution worked fine it had a performance issue. Why were we adding elements to a data structure just for it to be iterated at a later time? From this idea a new solution was born, XsdAsmFaster. This new solution aims to perform the same operations, faster, while providing a very similar user experience to the *fluent interface* generated by XsdAsm. To achieve that instead of storing information on a data structure we directly invoke the `ElementVisitor visit` method, this removes the need of storing and iterating information while maintaining all the expected behaviour. The two main moments that are affected by this change are the moments when an element is added to the tree and when an attribute is added to a previously created element. The code generated by XsdAsmFaster to add elements is as shown in Listing 4.32.

```
public final class Html<Z extends Element> {
   protected final Z parent;
   protected final ElementVisitor visitor;

   public Html(ElementVisitor visitor) {
      this.visitor = visitor;
      visitor.visitElementHtml(this);
   }

   public Html(Z parent) {
      this.parent = parent;
      this.visitor = parent.getVisitor();
      this.visitor.visitElementHtml(this);
   }

   public final Html<Z> attrManifest(String attrManifest) {
      this.visitor.visitAttributeManifest(attrManifest);
      return this;
   }

   public Body<T> body() { return new Body(this); }
   public Head<T> head() { return new Head(this); }
}
```

Listing 4.32: Html Class Generated by XsdAsmFaster

As we can see in Listing 4.32 we can invoke the `visit` method in the constructor of the classes generated based on a XSD `<xsd:element>`, such as `Html` or `Body`, since the `ElementVisitor` instance is accessible for all the elements on the element tree. Since adding elements results in the creation of new objects, such as `Body` and `Head` in lines 22 and 24 of Listing 4.32 respectively, it results in the invocation of their respective `visit` method due to the `visit` method being called in each class constructor. The attributes have a very similar behaviour, although they do not have instances created their restrictions are validated by invoking a static validate method present in each attribute class. If the attribute has no restrictions then the behaviour is as shown in Listing 4.32, where the respective `visit` method is called the method ends returns the object `this` to continue with the fluent tree creation.

The XsdAsmFaster solution also adds many other performance improvements. The `ElementVisitor` methods were changed to receive `String` objects instead of `Attribute` types. Changing this removes the requirement to instantiate attribute classes since we can directly pass the `name` of the attribute and its `value` as shown in `attrManifest` method in Listing 4.32. This change was performed since the only contained fields in attribute classes were `name` and `value`. The `ElementVisitor` class of XsdAsmFaster is as present in Listing 4.33.

```
1  public abstract class ElementVisitor {
2     public abstract void visitElement(Element element);
3     public abstract void visitAttribute(String attributeName,
4                                         String attributeValue);
5     public abstract void visitParent(Element element);
6     public abstract <R> void visitText(Text<? extends Element, R> t);
7     public abstract <R> void visitComment(Text<? extends Element, R> t);
8
9     public void visitOpenDynamic() {    }
10    public void visitCloseDynamic() {    }
11
12    // The methods below are generated based on the generated elements.
13    public void visitParentHtml(Html element) {
14       this.visitParent(element);
15    }
16    public void visitElementHtml(Html element) {
17       this.visitElement(element);
18    }
19 }
```

Listing 4.33: ElementVisitor Generated by XsdAsmFaster

Another feature that was introduced with XsdAsmFaster is both methods `visitOpenDynamic` and `visitCloseDynamic`. These methods have the objective to inform the concrete implementation of the `ElementVisitor` type that every `visit` method called in between calls of `visitOpenDynamic` and `visitCloseDynamic` represent dynamic data. That also means that every other `visit` method call outside of the dynamic spectrum is static. In Section 4.3.3 we will show how this feature can be used to improve the performance of the resulting solution.

4.3 Client

To use and test both XsdAsm and XsdParser we needed to implement a client for XsdAsm. Three different clients were implemented, one using the HTML5 specification, one using the specification for Android visual layouts and another one by creating a specification for the regular expression language. In this section we are going to explore how the HTML5 *fluent interface* is generated using the XsdAsm library and how to use it. Other generated *fluent interfaces*, such as the Android Layouts and Regex, follow the exact same process in their creation and usage.

4.3.1 HtmlApi

To generate the HTML5 *fluent interface* we need to obtain its XSD file. After that there are two options, the first one is to create a Java project that invokes the XsdAsm `main` method directly by passing the path of the specification file and the desired *fluent interface* name that will be used to create a custom package name, shown in Listing 4.34.

```
1 void generateApi(String xsdFilePath, String apiName){
2     XsdAsmMain.main(new String[] {xsdFilePath, apiName} );
3 }
```

Listing 4.34: Fluent Interface Creation

The second option is using the Maven[16] build lifecycle[17] to make that same invocation by adding an extra execution to the *Project Object Model* (POM) file, shown in Listing 4.35, to execute a batch file that invokes the XsdAsm `main` method, shown in Listing 4.36. More information about Maven will be provided in Section 5.1.

```
1  <plugin>
2      <artifactId>exec-maven-plugin</artifactId>
3      <groupId>org.codehaus.mojo</groupId>
4      <version>1.6.0</version>
5      <executions>
6          <execution>
7              <id>create_classes1</id>
8              <phase>validate</phase>
9              <goals>
10                 <goal>exec</goal>
11             </goals>
12             <configuration>
13                 <executable>
14                     ${basedir}/create_class_binaries.bat
15                 </executable>
16             </configuration>
17         </execution>
18     </executions>
19 </plugin>
```

Listing 4.35: Maven - Compiling Classes using a Plugin

```
1  if exist "./src/main/java" rmdir "./src/main/java" /s /q
2
3  if not exist "./target/classes/org/xmlet/htmlapi"
4      mkdir "./target/classes/org/xmlet/htmlapi"
5
6  call
7      mvn exec:java -D"exec.mainClass"="org.xmlet.xsdasm.main.XsdAsmMain"
8      -D"exec.args"="./src/main/resources/html_5.xsd htmlapi"
```

Listing 4.36: Maven - The Code that creates the Fluent Interface Classes (create_class_binaries.bat)

This client uses the Maven lifecycle option by adding an execution at the `validate` phase, shown in line 8 of Listing 4.35, which invokes XsdAsm `main` method to create the *fluent interface*. This invocation of XsdAsm creates all the classes in the target folder of the HtmlApi project. Following these steps would be enough to allow any other Maven project to add a dependency to the HtmlApi project and use its generated classes as if they were manually created. But this way the source files and Java documentation files are not created since XsdAsm only generates the class binaries. To tackle this issue we added another execution to the POM. This execution uses the Fernflower[31] decompiler, the Java decompiler used by Intellij[11] *Integrated Development Environment* (IDE), to decompile

the classes that were automatically generated, shown in Listing 4.37 and Listing 4.38.

```
<execution>
    <id>decompile_classes</id>
    <phase>validate</phase>
    <goals>
        <goal>
            exec
        </goal>
    </goals>
    <configuration>
        <executable>
            ${basedir}/decompile_class_binaries.bat
        </executable>
    </configuration>
</execution>
```

Listing 4.37: Maven - Decompiling Classes Using the Fernflower Plugin

```
if not exist "./src/main/java/org/xmlet/htmlapi" mkdir "./src/main/java
    /org/xmlet/htmlapi"

call
    mvn exec:java
    -D"exec.mainClass"="org.jetbrains.java.decompiler.main.decompiler.
       ConsoleDecompiler"
    -D"exec.args"="-dgs=true ./target/classes/org/xmlet/htmlapi ./src/
       main/java/org/xmlet/htmlapi"

if exist "./target/classes/org" rmdir "./target/classes/org" /s /q
```

Listing 4.38: Maven - The Code to Decompile the Generated Classes (decompile_class_binaries.bat)

By decompiling those classes we obtain the source code, which allows us to delete the automatic generated classes and allow the Maven build process to perform the normal compiling process, which generates the Java documentation files and the class binaries, along with the source files obtained from the decompilation process. This process, apart from generating more information to the programmer that will use the *fluent interface* in the future, also allows to find any problem with the generated code since it forces the compilation of all the classes previously generated.

4.3.2 Using the HtmlApi

After the previously described compilation process of the HtmlApi project we are ready to use the generated *fluent interface*. To start using it the first step is to implement the `ElementVisitor` class, which defines what to do when the created element tree is visited. A very simple example is presented in Listing 4.39, which writes the HTML tags based on the name of the element type visited, i.e. the opening tag in line 8 and the closing tag on line 19 of Listing 4.39, and navigates in the element tree by accessing the children of the current element type, i.e. the `getChildren()` method call followed by the invocation of the `accept` method call of every child present in the current element.

```
1  public class CustomVisitor<R> implements ElementVisitor<R> {
2
3      private PrintStream printStream = System.out;
4
5      public CustomVisitor(){ }
6
7      public <T extends Element> void sharedVisit(Element<T,?> element) {
8          printStream.printf("<%s", element.getName());
9
10         element.getAttributes()
11                 .forEach(attribute ->
12                     printStream.printf(" %s=\"%s\"",
13                         attribute.getName(), attribute.getValue()));
14
15         printStream.print(">\n");
16
17         element.getChildren().forEach(item -> item.accept(this));
18
19         printStream.printf("</%s>\n", element.getName());
20     }
21 }
```

Listing 4.39: Custom Visitor Example that Implements the ElementVisitor Generated by XsdAsm

After creating the `CustomVisitor` presented in Listing 4.39 we can start to create the element tree of the document that we want to present. To start we should create a `Html` object, since all the HTML documents have it as a base element. Upon creating that root element we can start to add other elements or attributes that will appear as options based on the specification rules. To help with the navigation on the element tree a method was created to allow the navigation to the parent of any given element. This method is named ___, a short method name to keep the code as clean as possible. In Listing 4.40 we can see a code example that uses a good amount of the *fluent interface* features:

- Element creation - For example, the `root.head()` method call adds an `Head` instance to the `Html` root element;

- Attribute assignment - For example, the `body().attrClass("clear")` method call adds the attribute `class` with the value `clear` to the `Body` instance created with the `body()` method call;

- Attributes receiving `Enum` classes - The `attrType(EnumTypeContentType.TEXT_CSS)` call, which indicates that the type value should be `text/css`, which is the value present in `EnumTypeContentType.TEXT_CSS`;

- Parent navigation - Both `(__())` method calls at the end of line 10 of Listing 4.40 will result in the current context being changed from the `Link` type to the `Head` type by the first call, followed by another call with changes the context from the `Head` type to the `Html` type. This allows to proceed with the definition of the `Body` type, which can only be contained in the `Html` type.

```
Html<Html> root = new Html<>();

root.head()
        .meta().attrCharset("UTF-8").__()
        .title()
            .text("Title").__()
        .link().attrType(EnumTypeContentType.TEXT_CSS)
                .attrHref("/assets/images/favicon.png").__()
        .link().attrType(EnumTypeContentType.TEXT_CSS)
                .attrHref("/assets/styles/main.css").__().__()
    .body().attrClass("clear")
        .div()
            .header()
                .section()
                    .div()
                        .img().attrId("brand")
                              .attrSrc("./assets/images/logo.png").__()
                        .aside()
                            .em()
                                .text("Advertisement")
                            .span()
                                .text("HtmlApi is great!");

CustomVisitor customVisitor = new CustomVisitor();

customVisitor.visit(root);
```

Listing 4.40: HtmlApi - The Definition of the Element Tree

With this element tree presented in Listing 4.40 and the previously presented `CustomVisitor`, shown in Listing 4.39, we obtain the following result as shown in Listing 4.41. The indentation was added for readability purposes, since the `CustomVisitor` implementation in Listing 4.39 does not indent the resulting HTML.

```
<html>
    <head>
        <meta charset="UTF-8">
        </meta>
        <title>
            Title
        </title>
        <link type="text/css" href="/assets/images/favicon.png">
        </link>
        <link type="text/css" href="/assets/styles/main.css">
        </link>
    </head>
    <body class="clear">
        <div>
            <header>
                <section>
                    <div>
                        <img id="brand" src="./assets/images/logo.png">
                        </img>
                        <aside>
                            <em>
                                Advertisement
                                <span>
                                    HtmlApi is great!
                                </span>
                            </em>
                        </aside>
                    </div>
                </section>
            </header>
        </div>
    </body>
</html>
```

Listing 4.41: HtmlApi - The Result of the Element Tree Visit

The `CustomVisitor` of Listing 4.41 is a very minimalist implementation since it does not indent the resulting HTML, does not simplify elements with no children (i.e. the `link`/`img` elements) and other aspects that are particular to HTML syntax. That is where the HtmlFlow library comes in, it implements the particular aspects of the HTML syntax in its `ElementVisitor` implementation that deals with how and where the output is written.

4.3.3 HtmlFlow 3

The HtmlFlow 3 library suffered some significant changes from its first version. At the moment it defines two ways of defining templates, a `StaticView` type, which allows the creation of template that does not receive any input data, and a `DynamicHtml` type, which receives a Java function that defines the template. We have already presented some code examples of both types along this document, but here we are going to analyze them. For the `StaticHtml` we have Listing 4.42, which is the same as Listing 1.3, duplicated here for explanation purposes.

```
1 private static void staticView(StaticHtml view){
2     view.html()
3             .body()
4                 .h1()
5                     .text("This is a static view h1 element.")
6                 .__()
7             .__()
8         .__();
9 }
```

Listing 4.42: HtmlFlow - Static View Example

With the definition of this template, i.e. Listing 4.42, we observe that this is a straightforward template, it has a couple of elements and does not require any external input. This type of template can be defined with any function that returns `void` and receives a `StaticView` object, i.e. a `Consumer<StaticView>` object.

```
1 String document = DynamicHtml.view(CurrentClass::studentView)
2                               .render(new Student("Luis", 39378));
3
4 static void studentView(DynamicHtml<Student> view, Student student){
5     view.html()
6           .body()
7             .ul()
8               .li().dynamic(li -> li.text(student.getName())).__()
9               .li().dynamic(li -> li.text(student.getNumber())).__()
10            .__()
11          .__()
12        .__();
13 }
```

Listing 4.43: HtmlFlow - Xmlet Template with Student Information

Regarding `DynamicHtml` templates, we can use and define them as shown in Listing 4.43. Their use is very similar, but there are a few noticeable changes. The first one is that now the function that is defining the template receives the *context object* associated with the template, which in Listing 4.43 is a `Student` object. The second change is that now the template should use a function, `dynamic` in lines 8 and 9 of Listing 4.43, to input dynamic information in the defined template while in the `StaticHtml` there was not such a call. This `dynamic` method has a very important significance in HtmlFlow 3, it provides information that the actions that will be performed inside the `Consumer` used as its parameter are dynamic and are subject to changes depending on the input received. Through the invocation of this method and thanks to strategy implemented by HtmlApiFaster the HtmlFlow 3 library has enough information to implement a caching strategy. In Listing 4.44 we can observe how HtmlApiFaster defines its `dynamic` methods, and how it notifies the `ElementVisitor` instance of the start, i.e. the invocation of `visitOpenDynamic()` method, and the end, i.e. the invocation of `visitCloseDynamic()` method, of the dynamic block.

```
1  public final class Li<Z extends Element> {
2
3     private ElementVisitor visitor;
4
5     public Li<Z> dynamic(Consumer<Li<Z>> consumer){
6        visitor.visitOpenDynamic();
7        consumer.accept(this);
8        visitor.visitCloseDynamic();
9        return this;
10    }
11 }
```

Listing 4.44: Li Class - The dynamic method

The caching strategy of HtmlFlow 3 uses a simple assumption, every action performed on the template elements that is not inserted in a dynamic block, i.e. is not performed inside of the `Consumer` used as a parameter to the `dynamic` method, is static information. This means that HtmlFlow 3 has sufficient information to cache different components of the template, e.g. in Listing 4.43 HtmlFlow can store three components of the template:

- The `String` representing the HTML text representing every element and attribute created before the call to the first `dynamic` method;

- The `String` representing the HTML text representing every element and attribute created between the call to the first `dynamic` and second `dynamic` methods;

- The `String` representing the HTML text representing every element and attribute created after the call to the second `dynamic` method;

This greatly improves the performance of the solution. Without this caching strategy the library would perform many more `StringBuilder` operations, since the result of creating elements, adding attributes or closing elements in this particular implementation of the `ElementVisitor` type of the HtmlApiFaster *fluent interface* is to use a `StringBuilder` object to `append` multiple `Strings`, added with the execution of the `visit` methods. By using the caching strategy the HtmlFlow 3 library avoids the invocation of the `append` method multiple times, which is decisive when it comes to performance.

This version of the HtmlFlow library also supports another interesting feature, partial views. This feature is present in many *template engines* since it allows for the same template to be used in many other templates, avoiding repetition. In Listing 4.45 we present a simplified example that uses partial views. We have the presented method, i.e. `presentationsView`, which defines a template that receives multiple `Presentation` instances to present. Instead of defining the whole template, it uses another template that is responsible for specifying how a single `Presentation` instance is presented. In line 5 of Listing 4.45 we can see how that works, we iterate the received `Iterable<Presentation> presentations` object and for each one of the objects we invoke the `addPartial` method, which receives the partial view instance, i.e. `PresentationView.view`, and the object that should be used to create the partial view.

```
1 private static void presentationsView(DynamicHtml<Iterable<Presentation
    >> view, Iterable<Presentation> presentations){
2     view.html()
3             .body()
4                 .div()
5                     .dynamic(div -> presentations.forEach(presentation
                          -> view.addPartial(PresentationView.view,
                          presentation)))
6                 .__()
7             .__()
8         .__();
9 }
```

Listing 4.45: HtmlFlow Partial Views

5

Deployment and Validation

This project and all its components belong to a Github organization called `xmlet`[1]. The aim of that organization is to contain all the related projects to this dissertation. All the generated DSLs are also created within this organization. With this approach all the existing projects and future generated DSLs can be accessed in one single place.

5.1 Maven

In order to manage the developed projects a tool for project organization and deployment was used, named Maven[16]. Maven has the goal of organizing a project in many different ways, such as creating a standard of project building and managing project dependencies. Maven was also used to generate documentation and deploying the projects to a central code repository, Maven Central Repository[2]. All the releases of projects belonging to the `xmlet` Github organization can be found under the same `groupId`, `com.github.xmlet` in the following location `https://search.maven.org/#search%7Cga%7C1%7Ccom.github.xmlet`.

[1] `https://github.com/xmlet`
[2] `https://search.maven.org/`

5.2 Sonarcloud

Code quality and its various implications such as security, performance and bugs should always be an important issue to a programmer. With that in mind all the projects contained in the `xmlet` solution were evaluated in various metrics and the results made public for consultation. This way, either future users of those projects or developers trying to improve the projects can check the metrics as another way of validating the quality of the produced code. The tool to perform this evaluation was Sonarcloud, which provides free of charge evaluations and stores the results that are available for everyone. The `xmlet` sonarcloud page is `https://sonarcloud.io/organizations/xmlet/projects`. Sonarcloud also provides an Web API to show badges that allow to inform users of different metrics regarding a project. Those badges are presented in the `xmlet` modules Github pages, as shown in Figure 5.1 for the XsdParser project.

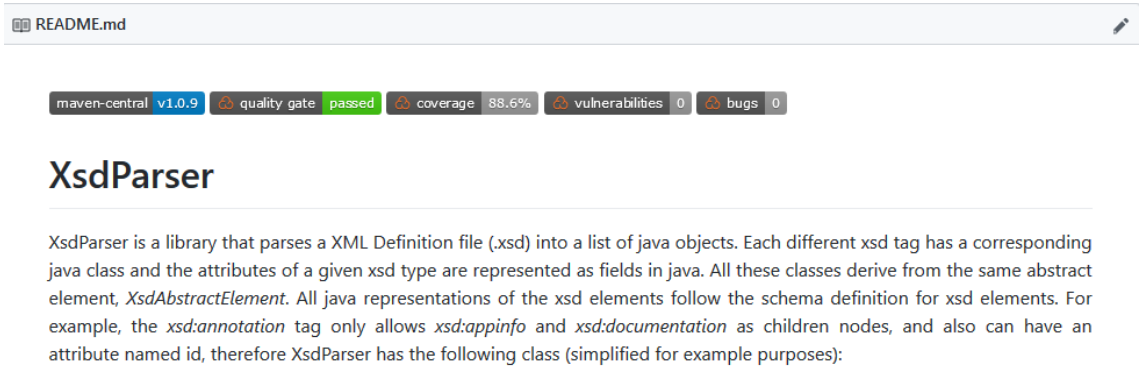


Figure 5.1: XsdParser with the Respective Sonarcloud Badges in Github

5.3 Testing metrics

To assert the performance of the `xmlet` solution we used the HTML5 use case to compare it against multiple other solutions. We used all the solutions that were presented in Chapter 3, i.e. J2Html, Rocker and KotlinX Html. To perform an unbiased comparison we searched on Github and used two popular benchmarks, this section will contain the results of these benchmarks. The computer used to perform all the tests present in this section has the following specifications:

Operative System: Windows 10 Education
Java Version: Java 8 Update 152

Processor: Intel Core i3-3217U 1.80GHz
RAM: 4GB

5.3.1 Spring Benchmark

This was the first benchmark solution we found, which is called `spring-comparing-template-engines`[21]. This benchmark uses the Spring[3] framework to host a web application that provides a route for each *template engine* to benchmark. Each *template engine* uses the same template and receives the same information to fill the template, which makes it possible to flood all the routes with an high number of requests and assert which route responds faster, consequently asserting which *template engine* is faster. This approach of measuring the *template engines* performance was dismissed because the render time of the *template* engines is dismissable when compared to the *overhead* introduced by the Spring framework and the tool used to flood the routes of the Web application. Even though that we did not end up using this specific benchmark we used the template that it used for its benchmark in another benchmark that added less *overhead*.

5.3.2 Template Benchmark

The second benchmark solution was `template-benchmark`[3]. The advantage of this benchmark is that it focus exclusively on evaluating the render process of each benchmark. In this case, it does not use any Web server to handle a request, which is a more consistent approach. The general idea of this benchmark is the same, it includes many *template engine* solutions that define the same template and use the same data to generate the complete document. But in this case instead of launching a Spring web application and issuing requests it uses *Java Microbenchmark Harness* (JMH)[19], which is a Java tool to benchmark code. With JMH we indicate which methods to benchmark with annotations and configure different benchmark options such as the number of warm-up iterations, the number of measurement iterations or the numbers of threads to run the benchmark method. This benchmark contained eight different *template engines* when we discovered it: Freemarker[1], Handlebars[13], Mustache[18], Pebble[20], Thymeleaf[26], Trimou[27], Velocity[28] and Rocker[22]. These *templates engines*, with the exception of Rocker that we already presented in Chapter

[3] `http://spring.io/`

3, are pretty much classic *template engines*, they all use a text file to define the template, using their own syntax to introduce the dynamic information. In addiction to these we added the solutions presented in the Chapter 3, J2Html and KotlinX Html.

The `template-benchmark` used only one template, which was the `Stocks` template. The template is shown in Listing 5.1 using the Mustache idiom.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Stock Prices</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <meta http-equiv="Content-Style-Type" content="text/css">
    <meta http-equiv="Content-Script-Type" content="text/javascript">
    <link rel="shortcut icon" href="/images/favicon.ico">
    <link rel="stylesheet" type="text/css" href="/css/style.css" media=
        "all">
    <script type="text/javascript" src="/js/util.js"></script>
    <style type="text/css">
      <!-- style content -->
    </style>
  </head>
  <body>
    <h1>Stock Prices</h1>
    <table>
      <thead>
          <tr>
            <th>#</th>
            <th>symbol</th>
            <th>name</th>
            <th>price</th>
            <th>change</th>
            <th>ratio</th>
          </tr>
        </thead>
        <tbody>
        {{#stockItems}}
          <tr class="{{rowClass}}">
            <td>{{index}}</td>
            <td>
                <a href="/stocks/{{value.symbol}}">{{value.symbol}}</a>
            </td>
            <td>
                <a href="{{value.url}}">{{value.name}}</a>
```

```
37              </td>
38              <td>
39                  <strong>{{value.price}}</strong>
40              </td>
41               <td{{negativeClass}}>{{value.change}}</td>
42               <td{{negativeClass}}>{{value.ratio}}</td>
43            </tr>
44          {{/stockItems}}
45          </tbody>
46      </table>
47    </body>
48 </html>
```

Listing 5.1: Stocks Template Defined in the Mustache Idiom

This template, of Listing 5.1, is pretty straightforward, it describes an HTML table that represents information regarding `Stock` objects, the `Stock` object is presented in Listing 5.2.

```
1 public class Stock {
2     private int index;
3     private String name;
4     private String url;
5     private String symbol;
6     private double price;
7     private double change;
8     private double ratio;
9 }
```

Listing 5.2: Stocks Data Type

Apart from this template and its associated data type that were already present in this benchmark solution we also used another template, the Presentations template, which was featured in the `spring-comparing-template-engines` benchmark. The Presentations template is as follow in Listing 5.3 and the respective `Presentation` object in Listing 5.4.

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width, initial-scale
          =1.0">
6     <meta http-equiv="content-language" content="IE=Edge">
```

```
7     <title>
8       JFall 2013 Presentations - htmlApi
9     </title>
10    <link rel="Stylesheet" href="/webjars/bootstrap/3.3.7-1/css/
        bootstrap.min.css" media="screen">
11  </head>
12  <body>
13    <div class="container">
14      <div class="page-header">
15        <h1>
16          JFall 2013 Presentations - htmlApi
17        </h1>
18      </div>
19      {{#presentationItems}}
20      <div class="panel panel-default">
21        <div class="panel-heading">
22          <h3 class="panel-title">
23            {{title}} - {{speakerName}}
24          </h3>
25        </div>
26        <div class="panel-body">
27          {{summary}}
28        </div>
29      </div>
30      {{/presentationItems}}
31    </div>
32    <script src="/webjars/jquery/3.1.1/jquery.min.js">
33    </script>
34    <script src="/webjars/bootstrap/3.3.7-1/js/bootstrap.min.js">
35    </script>
36  </body>
37 </html>
```

Listing 5.3: Presentations Template using the Mustache Idiom

```
1 public class Presentation {
2     private String title;
3     private String speakerName;
4     private String summary;
5 }
```

Listing 5.4: Presentation Data Type

By using two different templates the objective was to observe if the results were maintained throughout the different solutions. The main difference between both

templates are that the Stocks template introduces much more *placeholders* for two different reasons, it has more fields that will be accessed in the template and has twenty objects in the default data set while Presentations only has ten objects in his data set. This means that the Stocks template will generate more `String` operations to the classic *template engine* solutions and more Java method calls for the solutions that have the template defined within the Java language.

Now that we have two distinct templates implemented by over ten distinct solutions how will we benchmark these solutions? We have two methods in each *template engine* benchmark class, one for the Stocks template and other Presentations template. Those methods will load the respective template and apply the respective data set to it. Both these method are annotated with the `@Benchmark` annotation. We generate a *Java ARchive* (JAR) containing all these benchmark methods and will use the command line to perform the benchmark, removing the IDE *overhead*. The generated methods will then be benchmarked in two different variants, one with uses a single thread to run the benchmark method and other that uses four threads, the number of cores of the testing machine, to run the benchmark method. The results presented in this section are a result of the mean value of five forked iterations, each one of the forks running eight different iterations, performed after eight warm-up iterations. This approach intends to remove any outlier values from the benchmark. The benchmark values were obtained with the computer without any open programs, background tasks, only with the command line running the benchmark.

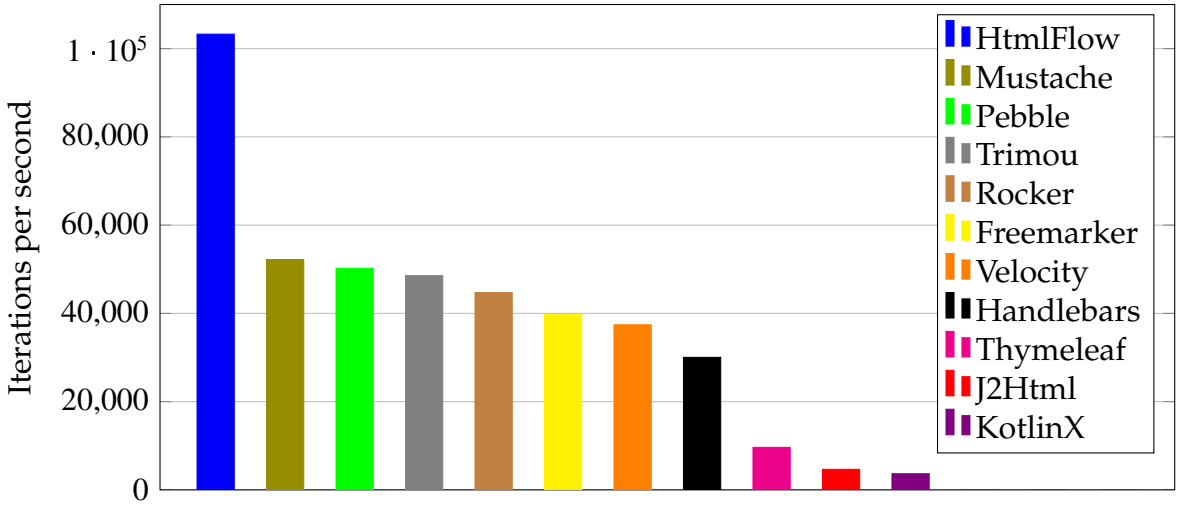


Figure 5.2: Benchmark Presentation - Results Gathered using One Thread

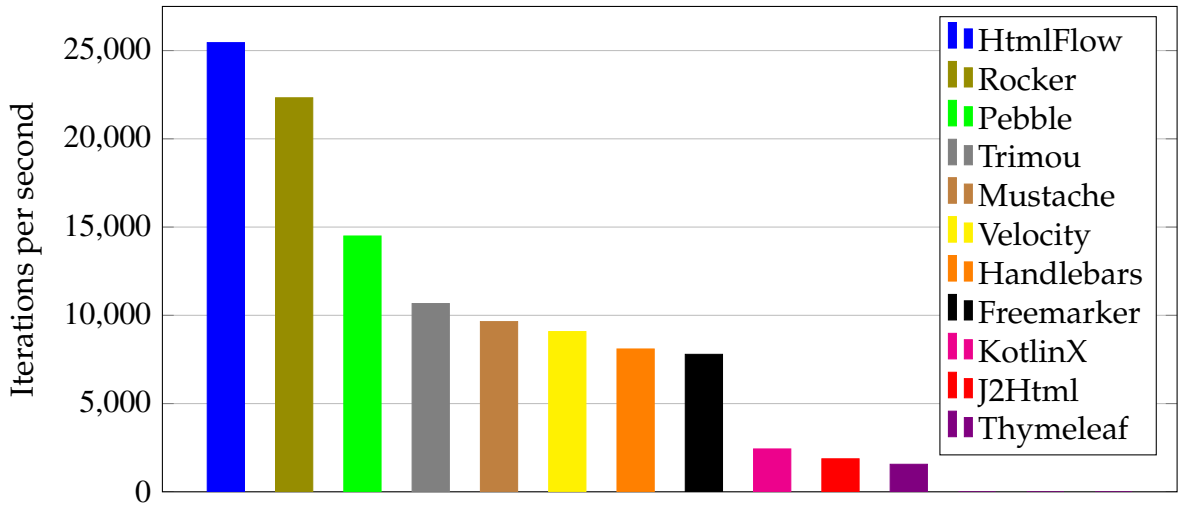


Figure 5.3: Benchmark Stocks - Results Gathered using One Thread

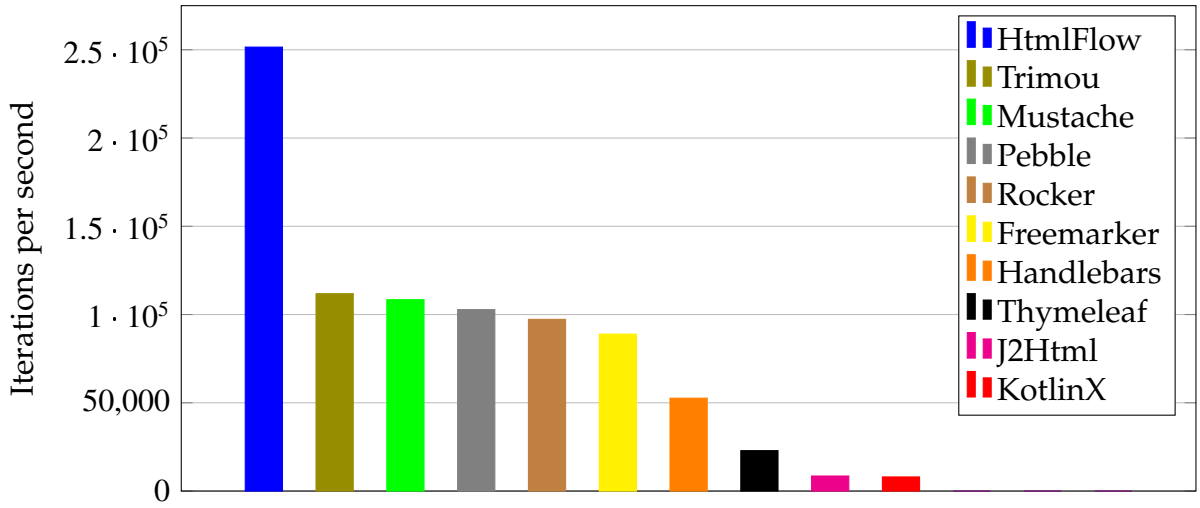


Figure 5.4: Benchmark Presentations - Results Gathered using Four Threads

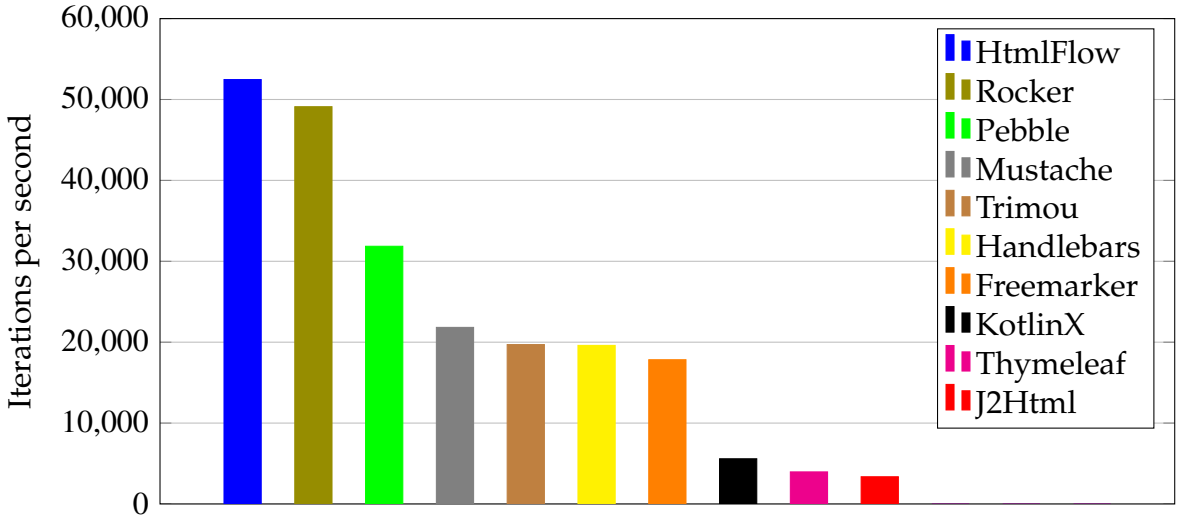


Figure 5.5: Benchmark Stocks - Results Gathered using Four Threads

To analyze these results we have to approach two different instances, the classical *template engines*, which use text files as their template, and the *template engines* that in some way diverge from that classical *template engine* solutions.

Regarding the classical *template engines*, i.e. Mustache/Pebble/Freemarker/Trimou/Velocity/Handlebars/Thymeleaf, we can observe that most of them share the same level of performance, which should be expected since they all roughly share the same methodology in their solution. The most notable outlier is Thymeleaf, which has a distinct difference to the other *template engines*.

Regarding the remaining *template engines*, i.e. Rocker/J2Html/KotlinX Html, the situation is diverse. On one hand we have Rocker, which presents a great performance when the number of *placeholders* increases, i.e. the Stocks benchmark, taking in consideration that it provides many compile time verifications regarding the *context objects* it presents a good improvement on the classical *template engine* solutions. On the other end of the spectrum we have J2Html and KotlinX Html. Regarding J2Html we observe that the trade off of moving the template to the language had a significant performance cost since it is consistently one of the two worst solutions performance-wise. Regarding KotlinX Html, the solution that is the most similar to the one that `xmlet` provides, the results are surprising, since they diverge so much from the results that the HtmlFlow achieves. KotlinX Html was definitely a step on the right direction since it validates the HTML language rules and introduces compile time validations but either due to the Kotlin language performance issues or poorly optimized code it did not achieve the level of performance that it could.

Lastly, the HtmlFlow library. The use case of the `xmlet` to the HTML language proved to be the best performance wise. The solution achieved values that surpass the second best solution by twice the iterations per second when using the Presentations benchmark and still held the top place in the Stocks benchmark even though the number of *placeholders* for dynamic information increased significantly. If we compare the HtmlFlow to the most similar solution, KotlinX, we observe a huge gain of performance on the HtmlFlow part. The performance improvement varies between HtmlFlow being nine times faster on the Stock benchmark with four threads, i.e. Figure 5.5, and thirty one times faster on the Presentations benchmark with four threads, i.e. Figure 5.4. In conclusion the `xmlet` solution introduces domain language rule verification, removes the requirements of text files and additional syntaxes, adds many compile time verifications and while doing all of that it still is the best solution performance wise.

6

Conclusion

In this dissertation we developed a structure of projects that can interpret a XSD document and use its contents to generate a Java *fluent interface* that allows to perform actions over the domain language defined in the XSD document while enforcing most of the rules that exist in the XSD syntax. The generated *fluent interface* only reflects the structure described in the XSD document, providing tools that allow any future usage to be defined according to the needs of the user. Upon testing the resulting solution we obtained better results than similar solutions while providing a solution with more safety validations and defined in a fluent language, which should be intuitive for users that have previously used the language.

The main language definition used in order to test and develop this solution was the HTML5 syntax, which generated the HtmlApiFaster project, containing a set of classes reflecting all the elements and attributes present in the HTML language. This HtmlApiFaster project was then used by the HtmlFlow 3 library in order to provide a library that safely writes well formed HTML documents. Other XSD files were used to test the solution, such as the Android layouts definition file, which defines the existing XML elements used to create visual layouts for the Android operating system and the attributes that each element contains, and a XSD file specifying the operations of the regular expressions language.

6.1 Main Contributions

Simply put, the work developed in this dissertation achieved the fastest Java *template engine* known to this date – HtmlFlow 3. This library not only shows the best performance than overall state of the art alternatives, but it also provides a full set of safety features not met together in any other library individually, such as:

- Well-formed documents;
- Fulfillment of all HTML rules regarding elements and attributes;
- Fully support of the HTML5 specification.

Despite HtmlFlow being developed in 2012 just as an academic use case of a fluent API for HTML, which was not released nor disseminated, it still attracted the attention of some developers looking for a Java library that helps them to dynamically produce papers, reports, emails and other kind of documents using HTML. Up to that date, textual template engines were still the main solution to produce dynamic HTML documents. Textual templates are a win-win approach for Web development fitting most HTML views requirements based on a strict domain model. Yet, textual templates are inadequate for more complex programming tasks involving the dynamic build of user-interface components, which may depend on run time introspection data.

The increasing attention around HtmlFlow raised the idea of developing a mechanism that automatically generates a *fluent interface* based on the HTML language specification, specified in a XSD file. The research work around such approach and its implementation was the main goal of this dissertation's thesis, which was successfully accomplished: Domain Specific Language generation based on a XML Schema.

This achievement fulfilled the HtmlFlow features and turns it into a complete Java DSL for HTML. Moreover, the usefulness of HtmlFlow was proven by more recent DSL libraries with the same purpose, such as J2Html and KotlinX.Html (both created in 2015), but which neither provide the same safety guards nor even the performance of HtmlFlow.

Despite my preliminary research not containing any proposal of a process to generate a DSL based on a XSD file, the solution evolved in that direction. Curiously I later discovered the KotlinX.Html solution, which follows this same idea. This

fact shows the effectiveness of the methodology proposed in this dissertation's thesis, which is already used by other library (i.e. KotlinX.Html) with a wide acceptance in the Kotlin community.

Performance would not be a major milestone at the beginning of this work. Yet, the J2Html assertion that it can be about a "thousand times faster than Apache Velocity" gave us one more purpose to this dissertation. First, we started to look on how J2Html compares its performance with other template engines and we found that comparison shared several flaws. Its dynamic render test `FiveHundredEmployees` was not really dynamic, because the context object is known before render time in case of the J2Html template, whereas for Velocity it is just provided at render and thus it favors the J2Html performance. Moreover, these tests use `junit-benchmarks` from `http://labs.carrotsearch.com/`, which is publicly announced as being deprecated in favor of JMH.

So, after some research, we start evaluating HtmlFow 3 in some of state of the art benchmarks and observing its good performance we realized that we could achieve a revolutionary landmark with `xmlet`. Here we found Rocker, which had a very curious approach, using static fields to store the static template components, which proved to perform very well. This, along with some other ideas created along the development of `xmlet`, led to creation of HtmlApiFaster, a *fluent interface* generated with many optimization techniques and support for the implementation of a caching strategy, in some way similar to the technique use by Rocker. These optimizations would make HtmlFlow 3 the most performant *template engine* in Java up to this date.

Finally, the `xmlet` platform is not only limited to the HTML language and we also tested it with other XSD files. In this case we successfully used it with the Android layouts definition file, which defines the existing XML elements in visual layouts for the Android operating system and the attributes that each element contains. To prove that this solution worked with a DSL not related with XML we described the operations of the Regular Expressions language in XSD and generated a *fluent interface* which supports the whole regular expressions syntax supported by Java. Both these projects were released and are available in the `xmlet` Github page.

Concluding, the main contributions resulting from the research work described in this dissertation are:

- `xmlet` - A Java platform for Domain Specific Language generation based on a XML Schema.

- XsdParser - A library that parses a XML Definition file, i.e. a XSD file, into a list of Java objects. This is the first Java library in this field that has already started attracting the attention of some developers.
- HtmlApi - A Java DSL for HTML complying with all the HTML 5 rules.
- HtmlFlow 3 - The most performant Java template engine.

6.2 Concluding Remarks and Future Directions

The `xmlet` solution in its current state achieved all the objectives that were proposed at the beginning of this dissertation as well as some other improvements that were identified along the development process. One of objectives from now on should be to find pertaining use cases ranging from markup languages which were the initial objective or any other domain language that can be defined through the XSD syntax.

Regarding the HtmlFlow, at the time of this work we are still preparing the release 3.0. A lot of effort has been made to create this release involving many aspects such as:

- The creation of a consistent API that gives an intuitive user experience to the end programmer;
- Ensuring code coverage tests close to 100%;
- Detecting and suppressing every bottleneck that could hurt performance;
- Many other time-consuming tasks involving the maintenance of an open-source project.

Once version 3.0 is released we must propose a pull request to the main Github `template-benchmark` repository including the comparison with HtmlFlow. Although we already have a fork of this repository with HtmlFlow, J2Html, Rocker and KotlinX.html integrated for performance comparison tests, now we must create a clean integration only with HtmlFlow release 3.0 for the pull request. This is the `template-benchmark` policy for integration of new template engines, i.e. one pull request per template engine.

Still related with HtmlFlow we have a paper in progress analyzing most recent type safe *template engines* and comparing different features provided by these engines. To the best of my knowledge all comparisons around Java *template engines* not only ignore the HTML safety aspects, but are also restricted to text template files.

Finally, to complete the HtmlFlow offer we would like to include a new tool that is able to translate HTML documents to an HtmlFlow definition. This tool has a similar role to the `ASMFier` in ASM translating Java source code to the equivalent definition in ASM. We think that tool will help programmers to migrate existing templates from other technologies to HtmlFlow.

To finish my dissertation, I would like to reinforce that over past 2 decades, text templates are still the de facto standard for dynamic HTML documents. This approach is great and fits the main web development requirements. However, we leave here two considerations:

- They are slow;
- Most of the are not safe.

In this context, I think that we should have better tools that suppress these issues and I believe the result of this dissertation's thesis is a significant step towards achieving this goal.

Bibliography

[1] Apache. Apache freemarker, February 2015. URL https://freemarker.apache.org/. (pp. 12 and 73)

[2] ASM. Asm. URL https://asm.ow2.io/. (p. 42)

[3] Mitchell Bösecke. Template benchmark. URL https://github.com/mbosecke/template-benchmark. (pp. 1 and 73)

[4] Fernando Miguel Carvalho. Htmlflow. URL https://github.com/xmlet/HtmlFlow. (pp. 1 and 24)

[5] Per. Christensson. Markup language definition., 2011. URL https://techterms.com/definition/markup_language. (p. 11)

[6] Martin Fowler. Fluent interfaces, 2005. URL https://martinfowler.com/bliki/FluentInterface.html. (p. 1)

[7] Martin Fowler. Domain specific languages, May 2008. URL https://martinfowler.com/bliki/DomainSpecificLanguage.html. (p. 1)

[8] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010. ISBN 0321712943, 9780321712943. (p. 1)

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994. ISBN 0201633612. URL http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1. (pp. 36 and 53)

[10] J2Html. J2html. URL https://j2html.com/. (p. 25)

[11] Jetbrains. Intellij integrated development environment. URL https://www.jetbrains.com/idea/. (p. 61)

[12] JMock. Jmock. URL http://jmock.org/. (p. 2)

[13] Yehuda Katz. Handlebars. URL http://handlebarsjs.com/. (pp. 12 and 73)

[14] Kotlin. Kotlin. URL https://kotlinlang.org/. (p. 29)

[15] Kotlin.Html. Kotlinx.html. URL https://github.com/Kotlin/kotlinx.html. (p. 30)

[16] Apache Maven. Maven, . URL https://maven.apache.org/. (pp. 60 and 71)

[17] Apache Maven. Introduction to the build lifecycle, . URL http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html. (p. 60)

[18] Mustache. Mustache. URL https://mustache.github.io/. (pp. 1, 7, 12, and 73)

[19] Oracle. Java microbenchmark harness. URL http://openjdk.java.net/projects/code-tools/jmh/. (p. 73)

[20] Pebble. Pebble. URL https://github.com/PebbleTemplates/pebble. (pp. 1, 12, and 73)

[21] Jeroen Reijn. spring-comparing-template-engines. URL https://github.com/jreijn/spring-comparing-template-engines. (pp. 1 and 73)

[22] Rocker. Rocker. URL https://github.com/fizzed/rocker. (pp. 1, 12, 26, and 73)

[23] Technopedia. Apache ant, . URL https://www.techopedia.com/definition/16219/apache-ant. (p. 2)

[24] Technopedia. Make, . URL https://www.techopedia.com/definition/16406/make. (p. 2)

[25] Technopedia. Structured query language, . URL https://www.techopedia.com/definition/1245/structured-query-language-sql. (p. 2)

[26] Thymeleaf. Thymeleaf. URL https://www.thymeleaf.org/. (pp. 12 and 73)

[27] Trimou. Trimou. URL http://trimou.org/. (pp. 1, 12, and 73)

[28] Velocity. Velocity. URL http://velocity.apache.org/. (pp. 12 and 73)

[29] Priscilla Walmsley. Xml schema, January 2004. URL http://www.datypic.com/sc/xsd/s-xmlschema.xsd.html. (p. 24)

[30] Wikipedia. Comparison of web template engines. URL https://en.wikipedia.org/wiki/Comparison_of_web_template_engines. (p. 7)

[31] Windup. Fernflower decompiler github. URL https://github.com/windup/windup/tree/master/decompiler/impl-fernflower. (p. 61)