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

Outubro, 2018

Júri:

Presidente:	[Grau e Nome do presidente do júri]
Vogais:	[Grau e Nome do primeiro vogal] [Grau e Nome do segundo vogal] [Grau e Nome do terceiro vogal] [Grau e Nome do quarto vogal]

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

Outubro, 2018

Júri:

Presidente:	[Grau e Nome do presidente do júri]
Vogais:	[Grau e Nome do primeiro vogal] [Grau e Nome do segundo vogal] [Grau e Nome do terceiro vogal] [Grau e Nome do quarto vogal]

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* 28

DSL *Domain Specific Language* 1

HTML *HyperText Markup Language* xi

IDE *Integrated Development Environment* 16

JAR *Java ARchive* 65

JMH *Java Microbenchmark Harness* 61

JVM *Java Virtual Machine* 23

POM *Project Object Model* 53

SAX *Simple Application Programming Interface for eXtensive Markup Language* . . 28

SQL *Structured Query Language* 2

XHTML *eXtensive HyperText Markup Language* 28

XML *eXtensive Markup Language* 1

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. Currently in order to tackle this problem the most used solution is *template engines*, which base their solution on the usage of an external files, do not ensure well formed documents and introduces the overhead of loading the *template* files to memory which degrades the overall performance.

Our objective is to create the required tools to generate *fluent interfaces* based on a language definition file, *eXtensive Markup Language Schema Definition* (XSD), while enforcing the restrictions of the given language. The generation of the *fluent interface* should be automated in order to avoid human error and expedite the coding process. By automating the *fluent interface* generation we also create a uniform approach to these domain languages.

To achieve our objectives we will use the Java language to extract the data from the language definition file. Based on the information provided by the language definition file we can then generate the adequate *bytecodes* to reflect the language definition to the Java language. To implement the language restrictions in Java we will always prioritize compile time validations, only performing run time validations of information that is not available when the *fluent interface* is generated.

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, XSD, Automatic Code Generation, `fluent interface`.

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. Actualmente essa tarefa é realizada por *template engines*, tendo como base ficheiros externos com *templates* de resposta, o que não garante que estes sejam bem formados e acrescenta o *overhead* do carregamento do ficheiro para memória.

O nosso objectivo é criar as ferramentas necessárias para gerar *interfaces fluentes* tendo em conta a sua definição sintática, expressa em XSD, garantindo que as restrições dessa mesma linguagem são verificadas. A geração de *interfaces fluentes* deve ser automatizada de modo a evitar erro humano e tornar a geração de código mais rápida. Automatizando a geração das *interfaces fluentes* cria-se também uma abordagem uniforme às diferentes linguagens de domínio utilizadas.

Para alcançar os nossos objectivos vamos utilizar a linguagem Java para extrair informação sintática da linguagem do seu ficheiro de definição. Tendo essa informação em conta vão ser gerados `bytecodes` para refletir a definição da linguagem para a linguagem Java. Para implementar as restrições em Java é sempre prioritizada a validação de restrições em tempo de compilação, apenas validando em tempo de execução a informação que não existe aquando da geração da *interface fluente*.

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, XSD, Geração Automática de Código, `interface` fluente.

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*[7] recreating a *Domain Specific Language* (DSL)[8] specified by an *eXtensive Markup Language* (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 create a Java DSL for HTML5, named *HtmlFlow*[5]. 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[23], Pebble[21], Trimou[28] or Mustache[19], in some of the more challenging benchmarks such as *template-benchmarks*[4] and *spring-comparing-template-engines*[22].

1.1 Introduction to Domain Specific Languages

High-level programming languages such as Java, C# and 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[9] we can find the regular expression presented in Listing 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, after a while it may become understandable. 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 it. Other examples of DSLs are languages such as *Structured Query Language* (SQL)[26], `Apache Ant`[24] or `make`[25].

DSLs can be divided in two types: external or internal. External DSLs are languages created without any affiliations 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 the JMock[13] which is a Java library that provides tools for test-driven development as shown in Listing 1.2.

```java
final GreetingTime gt = context.mock(GreetingTime.class);
Greeting g = new Greeting();
g.setGreetingTime(gt);

context.checking(new Expectations(){{
  one(gt).getGreeting(); will(returnValue("Good afternoon"));
}});
```

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` and asserts if the value `will` be `Good afternoon`. In this case the semantics of the methods used by JMock is meant to make it easier for the programmer to understand the tests that are being performed.

Internal DSLs can also be referred to as embedded DSLs since they are embedded in the programming language that 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 an XML based external DSL into an internal DSL. One of the requirements of this approach is that the external DSL 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 the rules of that language 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.

```java
private static void staticView(StaticHtml view){
    view.html()
            .body()
                .h1()
                    .text("This is a static view h1 element.")
                .°()
            .°()
        .°();
}
```

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, for example, that the `html` element can only contain either `head` or `body` element as children.

1.2 Template Engines

`Template engines` are solutions that use views, more specifically *dynamic views*, to build a 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`, 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.

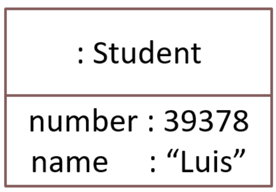

Figure 1.1: Student Object

After the template in Listing 1.4 receives the *context object* presented in Figure 1.1 the resulting HTML document is as presented 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 `Student`.

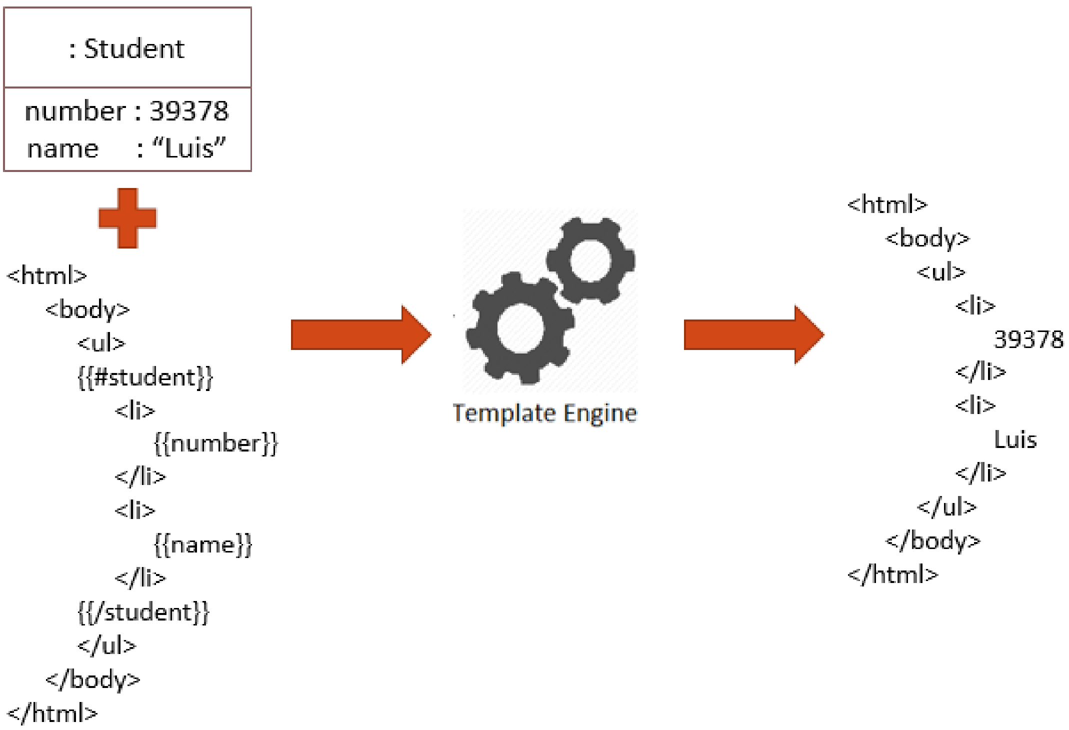

Figure 1.2: Template Engine Process which combines a Template Document with a Context Object

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[31] all of them share the same approach based on a text 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 which have to be loaded and therefore slow the overall performance of the application and the heavy usage 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 `for` operation to loop data.
- Complexity - It introduces one more syntax in the programming environment. For example a Java application using the Mustache[19] *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 a 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 to perform operations on templates is replaced to 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.

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 with the `xmlet` solution. The specific details on how the code presented in this example works will be provided in Chapter 4.

```
String document = DynamicHtml.view(CurrentClass::studentView)
                              .render(new Student("Luis", 39378));

static void studentView(DynamicHtml<Student> view, Student student){
    view.html()
          .body()
            .ul()
              .li().dynamic(li -> li.text(student.getName())).°()
              .li().dynamic(li -> li.text(student.getNumber())).°()
            .°()
          .°()
        .°();
}
```

Listing 1.6: Xmlet Template with Student information

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

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

- XsdAsm - Which uses XsdParser to gather the required information 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 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. To show that this solution is viable with other XSD files we used another XSD file that detailed the rules of the XML syntax used to generated Android[1] visual layouts.

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 the `xmlet` solution to other existing 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* [6]. Markup languages work by adding annotations to text, the annotations being also known as tags, that 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

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

process that can either pass of fail. The most that a text editor can do is highlight the errors to the user.

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.

In the following we will present different problems resultng from the use of *template engines* to build HTML views. The examples provided in this section use eight different *template engines*: Freemarker[2], Handlebars[14], Mustache[19], Pebble[21], Thymeleaf[27], Trimou[28], Velocity[29] and Rocker[23]. These templates were used in experiment tests for different benchmarks presented in Chapter 5.

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

```
<html>
  <!-- -->
</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 that the `html` element does not have a matching closing tag. It would be expected that in the very least *template engine* would issue an error while reading the file at run time. But every one of the *template engines* used with this example 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 open and closing them should be performed by the DSL library and not by the person who is writing the template.

The second problem 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 2.2: HTML Template with Placeholders

The template of Listing 2.2 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 2.3.

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

Listing 2.3: Template Engine with a valid Context Object

The previous example, Listing 2.3, is correct since there is one object in the *context object* with the `student` key with an instance of a `Student` object, which contains a `name` and `number` fields, which corresponds with the usage performed in the template defined in Listing 2.2. Yet in Listing 2.4 and Listing 2.5 we show another situation of illegal `context objects` which turn their use invalid by the template of Listing 2.2.

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

Listing 2.4: Template Engine with a Context Object with a wrong key

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

Listing 2.5: Template Engine with a Context Object with a wrong type

The first `context object` of Listing 2.4 has a wrong key, `teacher`, whereas the template is expecting an object with the `student` key. The second `context object` of Listing 2.5 has the right key but has a different type, which does not match the fields expected by template of Listing 2.2.

With this information in mind how will the eight template engines react when receiving these two wrongly defined *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 converted in a Java class at compile time 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 2.4 and in the second case, Listing 2.5, only Freemarker and Thymeleaf throw an exception. The remaining solutions ignore the fact that something that is expected is not there 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. Since the template would be defined as a Java function its `context object` would be represented by the method arguments, which have their type validated at compile time.

Another improvement of using an internal DSL over the use of `template engines` is the removal of language heterogeneity. For example, even for the simplest templates we have to atleast use three distinct syntaxes. In the following example we will use the Pebble *template engine*, one who requires less verbose. 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` of `Student` as shown in Listing 2.6.

```
<html>
  <body>
      <ul>
      {% for student in students %}
        <li>{{student.name}}</li>
      {% endfor %}
    </ul>
    </body>
</html>
```

Listing 2.6: List of Student Names Template 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 received type. Apart from the template definition we also need the Java code to generate the complete document, as shown in Listing 2.7.

```
PebbleEngine engine = new PebbleEngine.Builder().build();
template = engine.getTemplate("templateName.html");
StringWriter writer = new StringWriter();

Map<String, Object> context = new HashMap<>();
context.put("students", getListOfStudents());

template.evaluate(writer, context);

String document = writer.toString();
```

Listing 2.7: List of Student Names building in Java using Pebble

Even though the template in Listing 2.6 is simple the usage of multiple syntaxes introduces more complexity to the problem. If we escalate 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 2.8 and the template building in Listing 2.9.

```
static void studentListTemplate(DynamicHtml<Iterable<Student>> view,
                                Iterable<Student> students){
    view.html()
            .body()
                .ul()
                    .dynamic(ul ->
                      students.forEach(student ->
                        ul.li().text(student.getName()).°()))
                .°()
            .°()
        .°();
}
```

Listing 2.8: List of Student Names Template using HtmlFlow/xmlet

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

Listing 2.9: List of Student Names building using HtmlFlow/xmlet

With this solution we have a very compact template definition, where the context object, i.e. the `Iterable<Student> students`, 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.

Another quality of life improvement that we obtain by an internal DSL is navigability. One aspect that is very common in template engines solutions is to define *partial views* that can be reused in different views, but with regular text editors sometimes is hard to navigate back and forth between partial/regular templates. By using templates within the language we are able to quickly move between templates, since the template is either a method or a field and most *Integrated Development Environment* (IDE)s allow to quickly access both of them.

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* that it introduces and the issues that it was when the complexity increases, as it was presented in the previous 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 which consists on the `html` element, Listing 2.10, described in XSD of the HTML5 language definition. The presented example is simplified for explanation purposes.

```xml
<xs:attributeGroup name="commonAttributeGroup">
    <xs:attribute name="someAttribute" type="xs:string">
</xs:attributeGroup>

<xs:element name="html">
    <xs:complexType>
        <xs:choice>
            <xs:element ref="body"/>
            <xs:element ref="head"/>
        </xs:choice>
        <xs:attributeGroup ref="commonAttributeGroup" />
        <xs:attribute name="manifest" type="xs:string" />
    </xs:complexType>
</xs:element>
```

Listing 2.10: <hmtl> Element Description in XSD

With this example there is a multitude of classes that need to be created, apart from the always present supporting infrastructure that will be presented in Section 4.2.1.

- `Html` Element - A class that represents the `Html` element (Listing 2.11), deriving from `AbstractElement`.

- `Body` and `Head` Methods - Both methods present in the `Html` class (Listing 2.11) that add `Body` (Listing 2.12) and `Head` (Listing 2.13) instances to `Html` children list.

- `Manifest` Method - A method present in `Html` class (Listing 2.11) that adds an instance of the `Manifest` attribute (Listing 2.14) to the `Html` attribute list.

```java
class Html extends AbstractElement implements CommonAttributeGroup {
    public Html() { }

    public void accept(Visitor visitor){
    visitor.visit(this);
    }

    public Html attrManifest(String attrManifest) {
        return this.addAttr(new AttrManifest(attrManifest));
    }

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

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

Listing 2.11: Generated Html Element Class

- `Body` and `Head` classes - Classes for both `Body` (Listing 2.12) and `Head` (Listing 2.13) elements, similar to the generated `Html` class (Listing 2.11). The class contents will be dependent on the contents present in the concrete `xsd:element` nodes.

```java
public class Body extends AbstractElement {
    //Similar to Html, based on the contents of the respective
    //xsd:element node.
}
```

Listing 2.12: Generated Body Element Class

```java
public class Head extends AbstractElement {
    //Similar to Html, based on the contents of the respective
    //xsd:element node.
}
```

Listing 2.13: Generated Head Element Class

- `Manifest` Attribute - A class that represents the `Manifest` attribute (Listing 2.14), deriving from `BaseAttribute`.

```java
public class AttrManifestString extends BaseAttribute<String> {
    public AttrManifestString(String attrValue) {
        super(attrValue);
    }
}
```

Listing 2.14: Generated Manifest Attribute Class

- `CommonAttributeGroup` Interface - An interface with default methods that add the group attributes to the concrete element (Listing 2.15).

```java
public interface CommonAttributeGroup extends Element {
    default Html attrSomeAttribute(String attributeValue) {
        this.addAttr(new SomeAttribute(attributeValue));
        return this;
    }
}
```

Listing 2.15: Generated CommonAttributeGroup Interface

By analyzing this little example we can observe how the `xmlet` solution 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 in the XSD file of the given language, e.g. the user can only add `Head` and `Body` elements as children to the `Html` element and the same goes for attributes as well, the user can only add a `Manifest` or `SomeAttribute` objects as attribute. This solution effectively uses the Java compiler to enforce the specific language restrictions, most of them at compile time. The other handicaps are also solved, the template can now be defined within the Java language eradicating the necessity 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, it removes the necessity of additional syntax and now the Java syntax can be used 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 will be an utility project which is needed in order to parse all the external DSL rules present in the XSD document into structured Java classes.

The XsdAsm is the most important aspect of the `xmlet` solution, since it is the aspect which 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 much similar as possible to the language definition.

The HtmlApi will be a representation of client aspect of `xmlet` solution. It is a concrete client of the XsdAsm project, it will use the HTML5 language definition file in order to request of XsdAsm 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 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 more 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 for 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 then 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[30]. 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 the 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 that 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 Before Xmlet

The HtmlFlow[5] solution was the first to be approached in the developing process of the `xmlet` solution. The HtmlFlow motivation is to provide a library that allowed its users to write well formed type-safe HTML documents. The solution that existed prior to this project 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 grants 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. This problem led to the requirement of creating an automated process to translate the language rules to the Java language.

3.2.2 J2html

The J2html[11] solution 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 usage of the most of the HTML language, which is probably the reason why it has more garnered more attention than HtmlFlow. This library also shows that the issue we are trying to solve with the `xmlet` solution is relevant since this library has quite a few forks and watchers on their Github page[1].

3.2.3 Rocker

The Rocker[23] solution is very different from the two solutions presented before. Its approach is at is core very similar to the classic *template engine* solution since it still has 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 the loading of the template defined in a text file 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 is very good performance wise due to having all the static parts of the template `hardcoded` into the Java class that defines a specific template. This was by far the best competitor with `xmlet` performance wise. The biggest downside of this solution is that it does not verify the HTML language rules or even well formed XML documents.

3.2.4 KotlinX

Kotlin[15] 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 to the standard Java syntax the JVM implementation of the Kotlin library allows interoperability between both languages. The main reasons to use this language is that it heavily reduces the verbose needed to create code by using type inference and other techniques.

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

Kotlin is relevant to this project since one of his children projects, KotlinX, defines a DSL for the HTML language. The solution KotlinX 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 the need of using three distinct syntaxes, the programmer only programs in Kotlin.

KotlinX[16] is probably the solution which resembles the `xmlet` solution the most. The only difference is that the `xmlet` solution 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 only difference between KotlinX and the `xmlet` solution is performance, which is very poor compared to the other *template engines* solutions.

3.2.5 HtmlFlow With Xmlet

After developing the `xmlet` solution and adapting the HtmlFlow solution to use it the solutions characteristics changed. The positive aspects of the solution are kept since the general idea for the solution is kept with the usage of the `HtmlApi` generated by the `xmlet` solution. Regarding the negative aspects there were three main ones:

- 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.

By using the `xmlet` solution the HtmlFlow solution was able to improve its performance. With the mechanics created by the usage of the `xmlet` solution it is now possible to replicate the performance improvements of the Rocker solution. Even though the template rendering using the HtmlFlow is made as the template is being defined it is possible to implement a caching strategy that caches the static parts of the template, which result in huge performance boosts when the template is reused.

3.2.6 Feature Comparison

To have a better overview on all the previously presented solutions we will now present a table that has a list of important features and which solution implements it or not.

	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* - Before `xmlet` / Using `xmlet`
*1 - Template generated inside the language at compile time

As we can see in the Table 3.1, most these solutions tend to move the template definition to the language in which they are used, removing the *overhead* of loading the textual files and parsing them at run-time. 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 is the solution that mostly resembles what `xmlet` pretends to implement but is heavily handicapped when it comes to performance ranking among the worse *template engines* in the benchmarks performed in Chapter 5.

4 Solution

This chapter will present the `xmlet` solution, 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 and XsdAsm. In this case the XsdAsm has a dependency to XsdParser.

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 in Java. 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.

```java
public class XsdAnnotation extends XsdAbstractElement {

    private String id;
    private List<XsdAppInfo> appInfoList = new ArrayList<>();
    private List<XsdDocumentation> documentations = new ArrayList<>();

    // (...)
}
```

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).

```java
private Node getSchemaNode(String filePath)
    throws IOException, SAXException, ParserConfigurationException {
    DocumentBuilderFactory dbFactory =
                                DocumentBuilderFactory.newInstance();
    DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
    Document doc = dBuilder.parse(xsdFile);    //Parses the XSD file.

    //Obtains the first node, which is the xs:schema node.
    return doc.getFirstChild();
}
```

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.

```
Node schemaNode = getSchemaNode(filePath);

if (isXsdSchema(schemaNode)){
    XsdSchema.parse(this, schemaNode);
}
```

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

The `XsdSchema` element `parse` function converts the `Node` attributes into a `Map` object, which `XsdSchema` receives in the constructor. Each class extracts their field information from that `Map` object in their `setFields` method (Listing 4.4). To guarantee that the information parsed by the classes is valid according to the rules in the XSD language standard there are multiple validations. To validate the possible values for any given attribute, e.g. the `formDefault` 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.

```
public class XsdSchema extends XsdAnnotatedElements {
    private XsdSchema(XsdParser parser, Map<String, String> fieldsMap){
        super(parser, fieldsMap);
    }

    @Override
    public void setFields(Map<String, String> fieldsMap) {
        super.setFields(fieldsMap);

        this.attributeFormDefault =
          AttributeValidations.belongsToEnum(FormEnum.UNQUALIFIED,

            elementFieldsMap.getOrDefault(ATTRIBUTE_FORM_DEFAULT,
              FormEnum.UNQUALIFIED.getValue()));
        this.elementFormDefault =
          AttributeValidations.belongsToEnum(FormEnum.UNQUALIFIED,
            elementFieldsMap.getOrDefault(ELEMENT_FORM_DEFAULT,
              FormEnum.UNQUALIFIED.getValue()));
        this.blockDefault =
            AttributeValidations.belongsToEnum(BlockFinalEnum.DEFAULT,
```

```
                elementFieldsMap.getOrDefault(BLOCK_DEFAULT,
                  BlockFinalEnum.DEFAULT.getValue()));
            this.finalDefault =
                AttributeValidations.belongsToEnum(FinalDefaultEnum.DEFAULT
                    ,
                elementFieldsMap.getOrDefault(FINAL_DEFAULT,
                  FinalDefaultEnum.DEFAULT.getValue()));
            this.targetNamespace =
              elementFieldsMap.getOrDefault(TARGET_NAMESPACE,
                 targetNamespace);
            this.version = elementFieldsMap.getOrDefault(VERSION, version);
            this.xmlns = elementFieldsMap.getOrDefault(XMLNS, xmlns);
        }

    public static ReferenceBase parse(XsdParser parser, Node node) {
            NamedNodeMap nodeAttributes = node.getAttributes();
            Map<String, String> attrMap = convertNodeMap(nodeAttributes);

            return xsdParseSkeleton(node, new XsdSchema(parser, attrMap));
        }
}
```

Listing 4.4: XsdSchema Extracting Information from the received Node

The parsing of the `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 (Listing 4.5). This function will iterate in all the children of a given node, invoke the respective `parse` function of each children and then notify the parent element, using the Visitor pattern[10].

In the 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 A can reject the element B as his children if the element A does not support children of type B.

By using this strategy we ensure that the whole XSD file is parsed just by invoking the `parse` function of `XsdSchema`. This happens because the `XsdSchema` element is the top level element of all the XSD files and having all the concrete element types parsing their children will result in the parsing of the whole XSD file.

```java
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 4.5: XsdParseSkeleton Function - Parsing Children From a Node

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 4.6.

```xml
<xs:schema>
    <xs:element name="html">
        <xs:complexType>
            <!-- -->
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Listing 4.6: Parsing Concrete Example

Step 1 - DOM parsing:

The parsing starts with the DOM library parsing the code (Listing 4.6), 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 which will result in the invocation of the `setFields` method (line 7 of Listing 4.4), which will extract the information from the `Map` object to the class fields.

Step 3 - XsdSchema Children:

To parse the `XsdSchema` children the `XsdAbstractElement` `xsdParseSkeleton` (Listing 4.5) function is called (line 24 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 `setFields` function.

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 his children, since different types of objects should be treated differently.

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 existing elements with `ref` attributes referencing 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. That process starts by obtaining all the `NamedConcreteElement` objects since they may or may not be referenced by an existing `UnsolvedReference` object. The second step is to obtain all the `UnsolvedReference` objects and iterate them to perform a lookup search on the `NamedConcreteElement` objects obtained previously. This is achieved by comparing the value present in the `UnsolvedReference ref` attribute with the `NamedConcreteElement name` attribute. 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. A concrete example of how this process works is in Listing 4.7.

```
<xsd:schema xmlns='http://schemas.microsoft.com/intellisense/html-5'
    xmlns:xsd='http://www.w3.org/2001/XMLSchema'>

    <!-- NamedConcreteType wrapping a XsdGroup -->
    <xsd:group id="replacement" name="flowContent">
        <!-- (...) -->
    </xsd:group>

    <!-- ConcreteElement wrapping a XsdChoice -->
    <xsd:choice>
        <!-- UnsolvedReference wrapping a XsdGroup -->
        <xsd:group id="toBeReplaced" ref="flowContent"/>
    </xsd:choice>
</xsd:schema>
```

Listing 4.7: Reference Solving Example

In this short example we have a `XsdChoice` element that contains a `XsdGroup` element with a `ref` attribute. When replacing the `UnsolvedReference` objects the `XsdGroup` with the `ref` attribute is going to be replaced by a copy of the already parsed `XsdGroup` with the `name` attribute. This is achieved by accessing the parent of the element, in this case accessing the parent of the `XsdGroup` with the `ref` attribute, in order to remove the element identified by "toBeReplaced" and adding the element identified by "replacement".

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 other 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 create the respective rules are verified as seen with the `validateSchemaRules` method call in line 21 of Listing 4.5.

Each time any of these rules are violated a `ParsingException` is thrown containing a message detailing the rule that was violated, either being an attribute that does not match its type, an attribute that has a value that is not within the possible values for that attribute or the other more complex rules 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 elements that XsdAsm will use to obtain the information needed to generate the correspondent classes.

To generate classes this library also uses the ASM[3] 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 9 and is still maintained, the most recent version, 6.2.1, was release in 5 August 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 allow to increase the complexity of the generated code. In Listing 4.8 we present a class that is the objective of our code generation, it is a simple class, with a field and a method. The ASM library provides a tool, `ASMifier`, that receives a `.class` file and returns the ASM code needed to generate it, as shown in Listing 4.9.

```java
public class SumExample {

    private int sum;

    void setSum(int a, int b){
        sum = a + b;
    }
}
```

Listing 4.8: ASM Example - Code Generation Objective

```java
ClassWriter classWriter = new ClassWriter(0);

classWriter.visit(V9, ACC_PUBLIC + ACC_FINAL + ACC_SUPER,
  "Samples/HTML/SumExample", null, "java/lang/Object", null);

FieldVisitor fieldVisitor =
  classWriter.visitField(ACC_PRIVATE, "sum", "I", null, null);
fieldVisitor.visitEnd();

MethodVisitor methodVisitor =
  classWriter.visitMethod(0, "setSum", "(II)V", null, null);
methodVisitor.visitCode();
methodVisitor.visitVarInsn(ALOAD, 0);
methodVisitor.visitVarInsn(ILOAD, 1);
methodVisitor.visitVarInsn(ILOAD, 2);
methodVisitor.visitInsn(IADD);
methodVisitor.visitFieldInsn(PUTFIELD, "Samples/HTML/SumExample",
  "sum", "I");
methodVisitor.visitInsn(RETURN);
methodVisitor.visitMaxs(3, 3);
methodVisitor.visitEnd();

classWriter.visitEnd();

writeByteArrayToFile(classWriter.toByteArray());
```

Listing 4.9: ASM Example - Required Code

The strategy while creating the `xmlet` solution was to have manually created classes that represent a certain type of class that XsdASm will need to generate, i.e. the concrete element, attribute classes. By using the `ASMifier` tool with those template-like classes the programming process was expedited.

4.2.1 Supporting Infrastructure

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

Element classes:

- `Element` - An interface that serves as a base to every parsed XSD element.
- `AbstractElement` - An abstract class from where all the XSD element derive. This class implements most of the methods present on the `Element` interface.

Attribute classes:

- `Attribute` - An interface that serves as a base to every parsed XSD attribute.
- `BaseAttribute` - A class that implements the `Attribute` interface.

Visitor class:

- `ElementVisitor` - An abstract class that defines methods for all the generated elements that can be visited with the Visitor pattern. All the implemented methods point to a single 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 (Figure 4.1). In this example we have an element, `Html`, that extends `AbstractElement` and an attribute, `AttrManifestString`, that extends `BaseAttribute`.

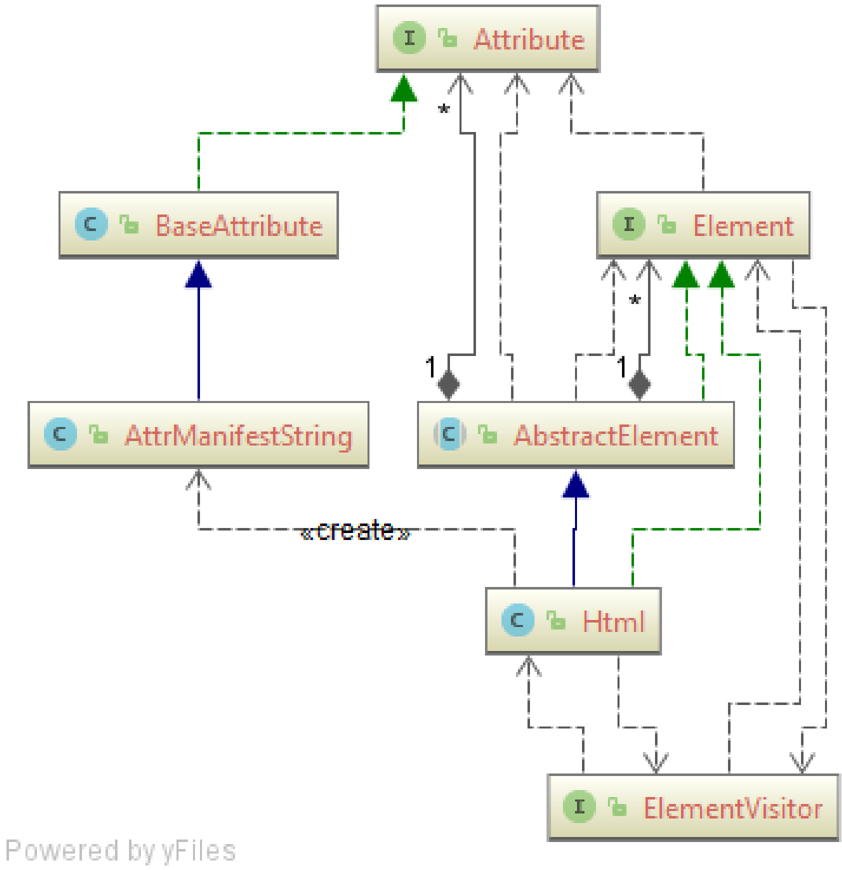

Figure 4.1: Fluent Interfaces - The Supporting Infrastructure

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 and 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* 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. To tackle this issue we rely on *type arguments*. Through *type parameters* we can always keep track of the tree structure of the elements that are being created and keep adding elements or moving up in the tree structure. In Listing 4.10 we can observe how the type arguments work.

```java
Html<Element> html = new Html<>();
Body<Html<Element>> body = html.body();
Div<Body<Html<Element>>> div = body.div();
Div<Body<Html<Element>>> divAfterAttribute =
                         div.attrClass("attrClassValue");
Body<Html<Element>> bodyAgain = divAfterAttribute.°();
Html<Element> htmlAgain = bodyAgain.°();
```

Listing 4.10: 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. The same happens with the `Div` element. This behavior changes when adding attributes, in which case we return the object which had the attribute added to them, while maintaining all the parent information. The last two lines show how to move up in the element tree by using the method `°`. The method name is meant to be as short as possible and to avoid distractions while reading the template.

While in the example presented in Listing 4.10 the usage of the *fluent interface* might seem to have excessive verbose to define a simple HTML document that verbose is not exactly needed. For specific purposes it might be needed to extract variables but the most common usage of the fluent interface should be more similar to Listing 4.11.

```
new Html<>()
      .body()
           .div().attrClass("attrClassValue").°()
      .°();
```

Listing 4.11: Example of the Implicit Use of Type Arguments

To provide a better understanding on how this is possible we need to showcase three distinct classes. First we have the `AbstractElement` class, Listing 4.14, from which all concrete elements 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, which returns the parent of any concrete element, the *type parameter* is guaranteed by returning `Z`, which is the type of the parent of the current element, as shown in the last two lines of code of Listing 4.10.

```
class AbstractElement<T extends Element, Z extends Element>
  protected Z parent;

    protected AbstractElement(Z parent) {
        this.parent = parent;
    }

  public Z °() {
    return this.parent;
  }

  // (...)
}
```

Listing 4.12: AbstractElement Class Type Arguments

The second class is the `Html` class, which is representative of every concrete element of an `xmlet` *fluent interface*. It has a single *type parameter*, Z which represents the type of its parent. It extends `AbstractElement` and therefore indicates that his type is `Html<Z>` and its parent type is `Z`. Any interface implemented by the concrete elements should receive the same type information as the `AbstractElement` class, as shown with `HtmlChoice0`. Regarding the `attrManifest` method, it indicates that returns the exact same instance as the one from which the method is called, keeping the type information by returning Html<Z>.

```
class Html<Z extends Element> extends AbstractElement<Html<Z>, Z>
                                      implements HtmlChoice0<Html<Z>, Z>
    // (...)

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

Listing 4.13: Html Class Type Arguments

As the third class we have the HtmlChoice0 interface which is representative of most interfaces of an xmlet *fluent interface*. It should receive the same *type parameters* as AbstractElement:

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

Since the Html is the only class implementing this interface its T *type argument* is always Html<Z> which results in a return type of Body<Html<Z>> when the method body is called.

```
interface HtmlChoice0<T extends Element<T, Z>, Z extends Element>
                                                   extends Element<T, Z> {
    default Body<T> body() {
        return (Body)this.addChild(new Body(this.self()));
    }
}
```

Listing 4.14: HtmlChoice0 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 in 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 restriction (Listing 4.15):

```
<xs:schema>
    <xs:element name="testElement">
        <xs:complexType>
            <xs:attribute name="intList" type="valuelist"/>
        </xs:complexType>
    </xs:element>

    <xs:simpleType name="valuelist">
        <xs:restriction>
            <xs:maxLength value="5"/>
            <xs:minLength value="1"/>
        </xs:restriction>
        <xs:list itemType="xsd:int"/>
    </xs:simpleType>
</xs:schema>
```

Listing 4.15: Restrictions Example XSD

In this example (Listing 4.15) we have an element (i.e. testElement) that has an attribute called `intList`. 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. Transporting this example to the Java language will result in the following class (Listing 4.16):

```
public class AttrIntListObject extends BaseAttribute<List<Integer>> {
    public AttrIntListObject(List<Integer> list) {
        super(list);
    }
}
```

Listing 4.16: Attribute Class with a List as its value

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 `RestrictionValidator` 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 4.17.

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

Listing 4.17: Attribute Constructor Enforcing Restrictions

In total there are 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` which is 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 4.18 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 4.18: 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 4.19) 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 4.19: Enumeration XSD Definition

This results in the creation of an `Enum` class, `EnumTypeCommand`, presented in Listing 4.20. The attribute class will then receive an instance of `EnumTypeCommand`, ensuring that only allowed values are used (Listing 4.21).

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

Listing 4.20: Enumeration Class

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

Listing 4.21: Attribute Receiving An Enumeration Instance

4.2.5 Element Binding

To support repetitive tasks over an element the `Element` and `AbstractElement` classes were modified to support binders. This allows programmers to define, for example, templates for a given element. An example is presented in Listing 4.22 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 }
```

Listing 4.22: 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()`. In regard to the values present in the table instead of having them inserted right away it is possible delay that insertion by indicating behaviour to execute when the information is received. This is achieved by implementing an `ElementVisitor` that supports binding.

In Listing 4.23 we can observe how the `ElementVisitor` would work. It maintains the default behaviour on the elements that are not bound (i.e. else clause). In the case that the element is bound to a function this implementation will clone the element and apply a model (i.e. a `List<String>` object following the example of Listing 4.22) to the clone, effectively executing the function supplied in the previously called binder method (i.e. Listing 4.22 line 8). This function call will generate new children on the cloned table element which 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.

```java
public class CustomVisitor<R> implements ElementVisitor<R> {

    private R model;

    public CustomVisitor(R model){
        this.model = model;
    }

    public <T extends Element> void sharedVisit(Element<T,?> element) {
        // ...
        if(element.isBound()) {
            List<Element> children = element.cloneElem()
                                            .bindTo(model)
                                            .getChildren();
            children.forEach( child -> child.accept(this));
        } else {
            element.getChildren().forEach(item -> item.accept(this));
        }
        // ...
    }
}
```

Listing 4.23: 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[10]. There are multiple `visit` methods that are invoked by the generated classes and the user can define the behaviour that the `ElementVisitor` has when those methods are called. This way the generated code delegates the responsibility to define how the user wants to interact with the generated DSL. The generated `ElementVisitor` class defines four main `visit` methods, Listing 4.24:

- `sharedVisit(Element<T, ?> element)` - This method is called whenever a concrete 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.

```java
public abstract class ElementVisitor<R> {
   <T extends Element> void sharedVisit(Element<T, ?> element);

   void visit(Text text);

   void visit(Comment comment);

   <U> void visit(TextFunction<R, U, ?> textFunction);
}
```

Listing 4.24: ElementVisitor Generated by XsdAsm - The Core Methods

Apart from these four main method we also create specific methods, as shown in Listing 4.25. 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 concrete `ElementVisitor` to have a very simple implementation of only four methods or redefine all the methods for a concrete purpose for the respective DSL.

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

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

Listing 4.25: 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 4.26 and 4.27, respectively.

```java
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 4.26: AbstractElement Class Generated by XsdAsm

```java
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 4.27: Html Class Generated by XsdAsm

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

- Creating the `Element` tree - We need to create the element tree by adding all elements and attributes (Listing 4.28);
- 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 (Listing 4.29).

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

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

Listing 4.28: HTML5 Tree Creation using XsdAsm

```
CustomVisitor customVisitor = new CustomVisitor();

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

Listing 4.29: 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 4.30.

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

   public Html(ElementVisitor visitor) {
      this.visitor = visitor;
      this.parent = null;
      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 4.30: Html Class Generated by XsdAsmFaster

As we can see in the previously Listing we can invoke the `visit` method in the constructor of the concrete element classes, in this case the `Html` class, since the `ElementVisitor` object is passed to all the elements on the tree. Since adding elements results in the creation of new objects, such as `Body` and `Head` in the previous example, it results in the invocation of their respective `visit` method due to the `visit` method being called in each concrete element constructor. The attributes have a very similar behaviour, although they do not have instances created their restrictions are validated, if present, and if all restrictions are validated the respective `visit` method is called and the method ends returning 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 concrete classes since we can directly pass the `name` of the attribute and its `value` as shown in `attrManifest` method in Listing 4.30. 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 4.31.

```java
public abstract class ElementVisitor {
   public abstract void visitElement(Element element);

   public abstract void visitAttribute(String attributeName,
                                       String attributeValue);

   public abstract void visitParent(Element element);

   public abstract <R> void visitText(Text<? extends Element, R> t);

   public abstract <R> void visitComment(Text<? extends Element, R> t);

   public void visitOpenDynamic() {   }

   public void visitCloseDynamic() {   }

   public void visitParentHtml(Html element) {
      this.visitParent(element);
   }

   public void visitElementHtml(Html element) {
      this.visitElement(element);
   }

   // (...)
}
```

Listing 4.31: 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` 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. By using this information the concrete ElementVisitor can implement a cache solution to store all the static components of the template. This generates a huge performance boost since one of the bottlenecks of writing a huge number of small String objects to a StringBuilder object, for example, is the number of append calls performed, by using the cache solution we can generally avoid a very large number of append calls. To indicate that a given block of the template is dynamic the user has to invoke the dynamic(Consumer consumer) method, Listing 4.32. This method will receive a Consumer, which will receive the current object, i.e. the this object. Every action performed in the *fluent interface* in that Consumer accept call will be considered a dynamic part of the template.

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

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

   public Html<Z> dynamic(Consumer<Html<Z>> consumer){
      visitor.visitOpenDynamic();
      consumer.accept(this);
      visitor.visitCloseDynamic();
      return this;
   }
}
```

Listing 4.32: Html Class - The Dynamic method

4.3 Client

To use and test both XsdAsm and XsdParser we need to implement a client for XsdAsm. Two different clients were implemented, one using the HTML5 specification and another using the specification for Android visual layouts. In this section we are going to explore how the HTML5 *fluent interface* is generated using the XsdAsm library and how to use it.

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 (Listing 4.33).

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

Listing 4.33: Fluent Interface Creation

The second option is using the Maven[17] build lifecycle[18] to make that same invocation by adding an extra execution to the *Project Object Model* (POM) file (Listing 4.34) to execute a batch file that invokes the XsdAsm `main` method (Listing 4.35). More information about maven in Section 5.2.

```
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 4.34: Maven - Compile Classes using a Plugin

```
if exist "./src/main/java" rmdir "./src/main/java" /s /q

if not exist "./target/classes/org/xmlet/htmlapi"
    mkdir "./target/classes/org/xmlet/htmlapi"

call
    mvn exec:java -D"exec.mainClass"="org.xmlet.xsdasm.main.XsdAsmMain"
    -D"exec.args"="./src/main/resources/html_5.xsd htmlapi"
```

Listing 4.35: 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 (Listing 4.34, line 8) 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[32] decompiler, the Java decompiler used by

Intellij[12] IDE, to decompile the classes that were automatically generated (Listing 4.36, 4.37).

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

Listing 4.36: Maven - Decompiling Classes using a 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 4.37: 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.

4.3.1.1 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 4.38 which writes the HTML tags based on the name of the element visited and navigates in the element tree by accessing the children of the current element.

```
public class CustomVisitor<R> implements ElementVisitor<R> {

    private PrintStream printStream = System.out;

    public CustomVisitor(){ }

    public <T extends Element> void sharedVisit(Element<T,?> element) {
        printStream.printf("<%s", element.getName());

        element.getAttributes()
                .forEach(attribute ->
                    printStream.printf(" %s=\"%s\"",
                        attribute.getName(), attribute.getValue()));

        printStream.print(">\n");

        element.getChildren().forEach(item -> item.accept(this));

        printStream.printf("</%s>\n", element.getName());
    }
}
```

Listing 4.38: Custom Visitor Example that Implements the ElementVisitor Generated by XsdAsm

After creating the `CustomVisitor` presented in Listing 4.38 we can start to create the element tree that we want convert to text using the `CustomVisitor`. 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 4.39 we can see a code example that uses a good amount of the *fluent interface* features, including element creation and how they are added to the element tree, how to add attributes, attributes that receive enumerations as parameters and how to navigate in the element tree using the method °.

```
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 4.39: HtmlApi - The Definition of the Element Tree

With this element tree presented (Listing 4.39) and the previously presented `CustomVisitor` (Listing 4.38) we obtain the following result (Listing 4.40). The indentation was added for readability purposes, since the `CustomVisitor` implementation in Listing 4.38 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 4.40: HtmlApi - The Result of the Element Tree Visit

The `CustomVisitor` of Listing 4.40 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 which deals with how and where the output is written.

5

Deployment

5.1 Github Organization

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.

5.2 Maven

In order to manage the developed projects a tool for project organization and deployment was used, named Maven[17]. 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/`

5.3 Sonarcloud

Code quality and its various implications such as security, low 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 which 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.

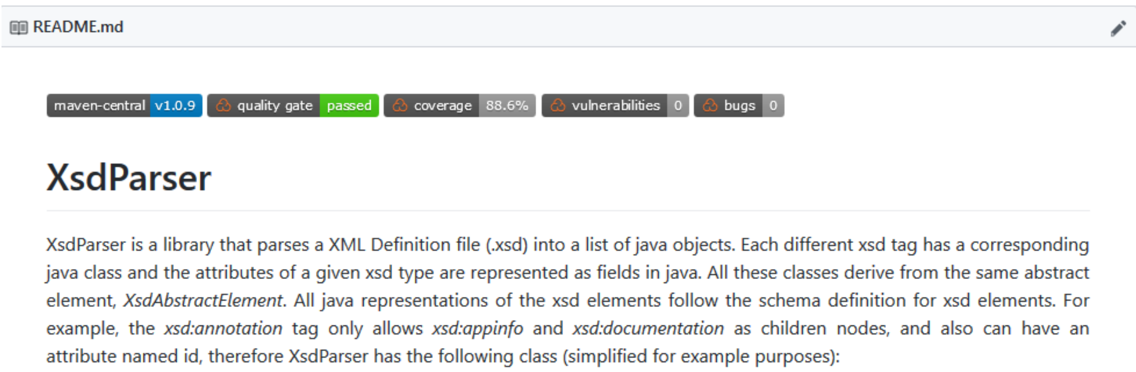

Figure 5.1: XsdParser with the respective Sonarcloud Badges in Github

5.4 Testing metrics

To assert the performance of the `xmlet` solution we used the HTML5 use case to compare it against the multiple solutions. We used all the solutions that were presented in Chapter 3 (J2Html, Rocker, Kotlin). To perform a unbiased comparison instead of creating our own benchmark solution 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:

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

5.4.1 Spring Benchmark

This was the first benchmark solution we found, which is called `spring-comparing-template-engines`[22]. This benchmark uses the Spring[3] framework to host a web application which serves 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 benchmark was promising but was disregarded since it introduced too much *overhead*, e.g. the Spring framework and the additional tool to benchmark the web application, which consumed quite a few resources to perform the requests to the web application. Even though that we did not end up using this specific benchmark we used the template that it used for the benchmark using another benchmark that added less *overhead*.

5.4.2 Template Benchmark

The second benchmark solution was `template-benchmark`[4]. This solution ended up being picked because it introduced less *overhead*. The general idea of the benchmark is the same, it includes many *template engine* solutions which all 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)[20] 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[2], Handlebars[14], Mustache[19], Pebble[21], Thymeleaf[27], Trimou[28], Velocity[29] and Rocker[23]. These *templates engines*, with the exception of Rocker that we already presented in Chapter 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

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

to these we added the solutions presented in the Chapter 3, J2Html and KotlinX.

The `template-benchmark` 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>
            </td>
            <td>
```

```
                <strong>{{value.price}}</strong>
            </td>
            <td{{negativeClass}}>{{value.change}}</td>
            <td{{negativeClass}}>{{value.ratio}}</td>
          </tr>
        {{/stockItems}}
        </tbody>
    </table>
  </body>
</html>
```

Listing 5.1: Stocks Template Defined in the Mustache Idiom

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

```
public class Stock {
    private int index;
    private String name;
    private String url;
    private String symbol;
    private double price;
    private double change;
    private double ratio;
}
```

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` solution. The Presentations template is as follow in Listing 5.3 and the respective `Presentation` object in Listing 5.4.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale
        =1.0">
    <meta http-equiv="content-language" content="IE=Edge">
    <title>
      JFall 2013 Presentations - htmlApi
```

```
    </title>
    <link rel="Stylesheet" href="/webjars/bootstrap/3.3.7-1/css/
       bootstrap.min.css" media="screen">
  </head>
  <body>
    <div class="container">
      <div class="page-header">
        <h1>
          JFall 2013 Presentations - htmlApi
        </h1>
      </div>
      {{#presentationItems}}
      <div class="panel panel-default">
        <div class="panel-heading">
          <h3 class="panel-title">
            {{title}} - {{speakerName}}
          </h3>
        </div>
        <div class="panel-body">
          {{summary}}
        </div>
      </div>
      {{/presentationItems}}
    </div>
    <script src="/webjars/jquery/3.1.1/jquery.min.js">
    </script>
    <script src="/webjars/bootstrap/3.3.7-1/js/bootstrap.min.js">
    </script>
  </body>
</html>
```

Listing 5.3: Presentations Template using the Mustache Idiom

```
public class Presentation {
    private String title;
    private String speakerName;
    private String summary;
}
```

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, one for the Stocks template and other Presentations template, in each *template engine* solution that will obtain/generate the template and insert a default set of elements for each template. 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.

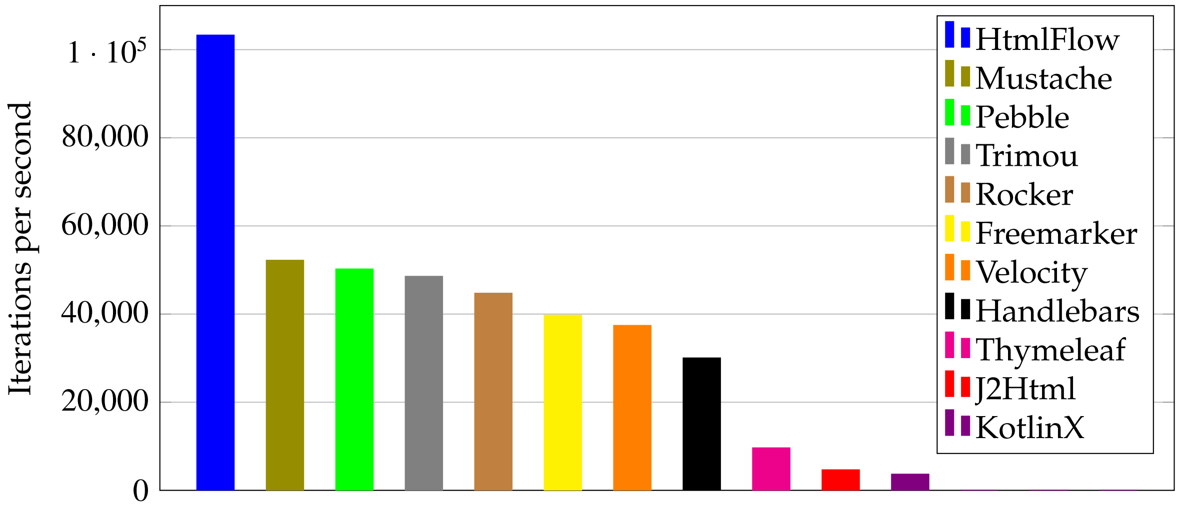

Figure 5.2: Benchmark Presentation - Results Gathered using one Thread

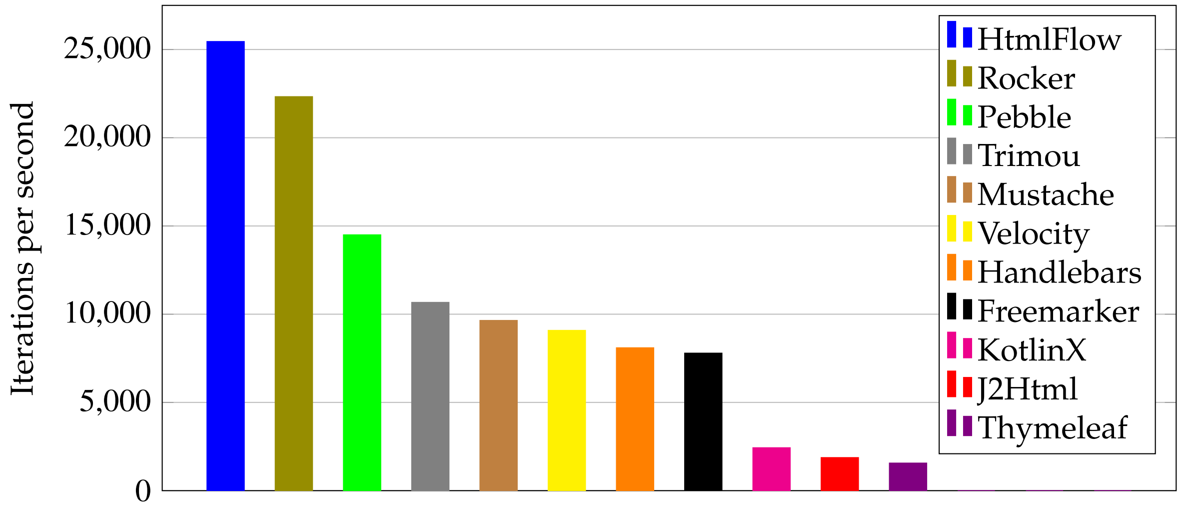

Figure 5.3: Benchmark Stocks - Results Gathered using one Thread

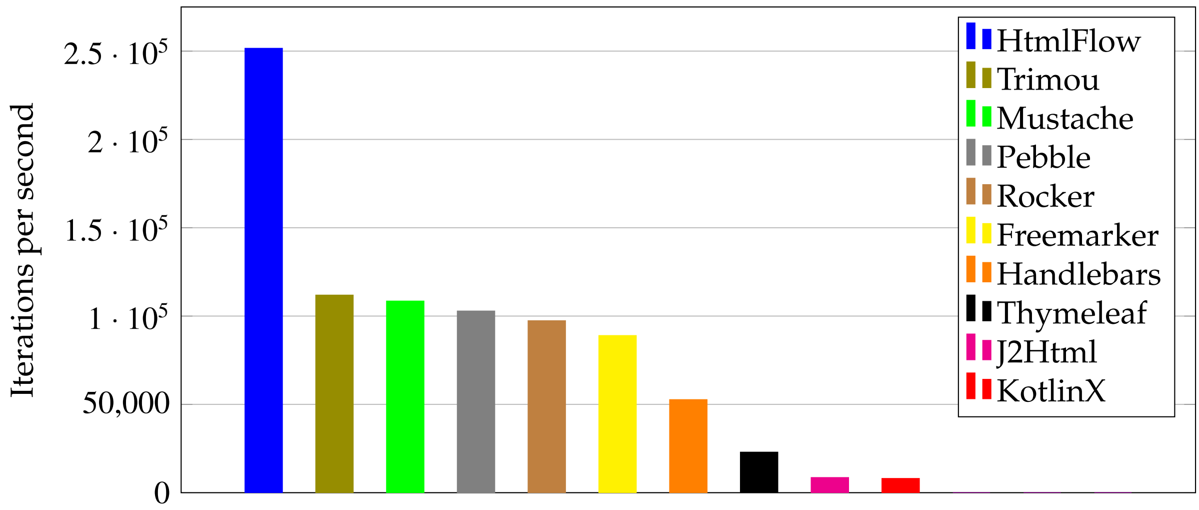

Figure 5.4: Benchmark Presentations - Results Gathered using four Threads

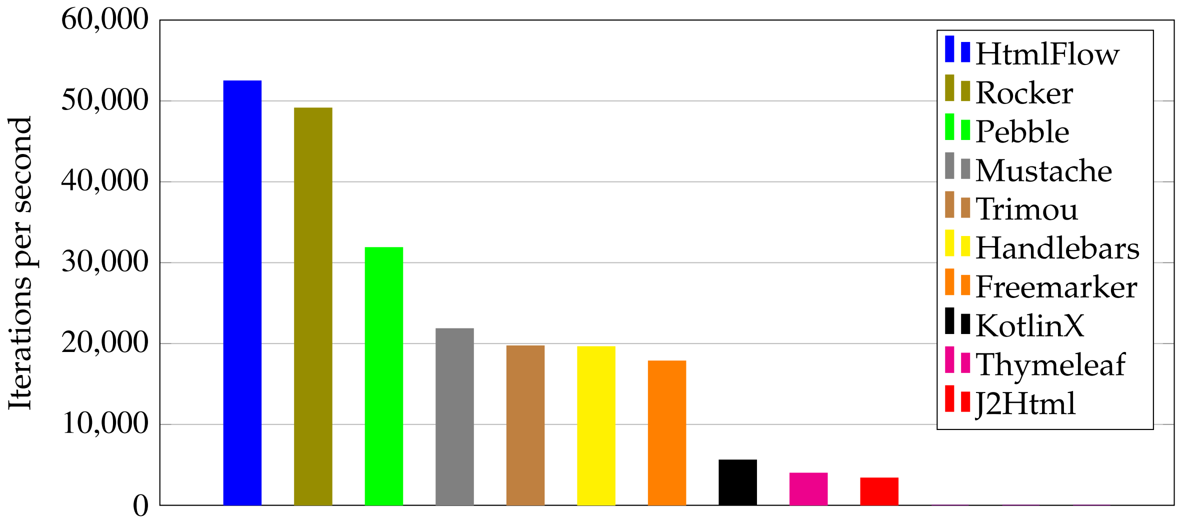

Figure 5.5: Benchmark Stocks - Results Gathered using four Threads

To analyze these results we have to approach two different instances, the more classical *template engines* 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, 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 engines solutions. On the other end of the spectrum we have J2Html and KotlinX. 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, 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 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 achieve.

Lastly, the HtmlFlow solution. 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 and thirty one times faster on the Presentations benchmark with four threads. 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 proving a solution with a fluent language, which should be intuitive even for people that never programmed in Java before, since the flow of the written code ends up being similar to writing XML.

The main language definition used in order to test and develop this solution was the HTML5 syntax, which generated the HtmlApi project, containing a set of classes reflecting all the elements and attributes present in the HTML language. This HtmlApi project was then used by the HtmlFlow library in order to provide an library that 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.

6.1 Future work

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. The objective 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.

Bibliography

[1] Android. Android. URL https://www.android.com/. (p. 9)

[2] Apache. Apache freemarker, February 2015. URL https://freemarker.apache.org/. (pp. 12 and 61)

[3] ASM. Asm. URL https://asm.ow2.io/. (p. 35)

[4] Mitchell Bösecke. Template benchmark. URL https://github.com/mbosecke/template-benchmark. (pp. 1 and 61)

[5] Fernando Miguel Carvalho. Htmlflow. URL https://github.com/xmlet/HtmlFlow. (pp. 1 and 22)

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

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

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

[9] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010. ISBN 0321712943, 9780321712943. (p. 2)

[10] 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. 30 and 46)

[11] J2Html. J2html. URL https://j2html.com/. (p. 22)

[12] Jetbrains. Intellij integrated development environment. URL https://www.jetbrains.com/idea/. (p. 55)

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

[14] Yehuda Katz. Handlebars. URL http://handlebarsjs.com/. (pp. 12 and 61)

[15] Kotlin. Kotlin, . URL https://kotlinlang.org/. (p. 23)

[16] Kotlin. Kotlinx, . URL https://github.com/Kotlin/kotlinx.html. (p. 24)

[17] Apache Maven. Maven, . URL https://maven.apache.org/. (pp. 53 and 59)

[18] Apache Maven. Introduction to the build lifecycle, . URL http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html. (p. 53)

[19] Mustache. Mustache. URL https://mustache.github.io/. (pp. 1, 7, 12, and 61)

[20] Oracle. Java microbenchmark harness. URL http://openjdk.java.net/projects/code-tools/jmh/. (p. 61)

[21] Pebble. Pebble. URL https://github.com/PebbleTemplates/pebble. (pp. 1, 12, and 61)

[22] Jeroen Reijn. spring-comparing-template-engines. URL https://github.com/jreijn/spring-comparing-template-engines. (pp. 1 and 61)

[23] Rocker. Rocker. URL https://github.com/fizzed/rocker. (pp. 1, 12, 23, and 61)

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

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

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

[27] Thymeleaf. Thymeleaf. URL https://www.thymeleaf.org/. (pp. 12 and 61)

[28] Trimou. Trimou. URL http://trimou.org/. (pp. 1, 12, and 61)

[29] Velocity. Velocity. URL http://velocity.apache.org/. (pp. 12 and 61)

[30] Priscilla Walmsley. Xml schema, January 2004. URL http://www.datypic.com/sc/xsd/s-xmlschema.xsd.html. (p. 22)

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

[32] Windup. Fernflower decompiler github. URL https://github.com/windup/windup/tree/master/decompiler/impl-fernflower. (p. 54)