



INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Automatic generation of Java API based on XML Schema

LUÍS CARLOS DA SILVA DUARTE

Licenciado em Engenharia Informática e de Computadores

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

Orientador : Doutor Fernando Miguel Gamboa de Carvalho

Júri:

Presidente: [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]

Junho, 2018



INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Automatic generation of Java API based on XML Schema

LUÍS CARLOS DA SILVA DUARTE

Licenciado em Engenharia Informática e de Computadores

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

Orientador : Doutor Fernando Miguel Gamboa de Carvalho

Júri:

Presidente: [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]

Junho, 2018

Aos meus pais.

Acknowledgments

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

Acronyms and Abbreviations

The list of acronyms and abbreviations are as follow.

API <i>Application Programming Interface</i>	1
DOM <i>Document Object Model</i>	10
HTML <i>HyperText Markup Language</i>	1
IDE <i>Integrated Development Environment</i>	27
JSP <i>JavaServer Pages</i>	6
POM <i>Project Object Model</i>	26
SAX <i>Simple Application Programming Interface for eXtensive Markup Language</i> ..	10
XHTML <i>eXtensive HyperText Markup Language</i>	10
XML <i>eXtensive Markup Language</i>	3
XSD <i>eXtensive Markup Language Schema Definition</i>	1

Abstract

The dissertation must contain two versions of the abstract, one in the same language as the main text, another in a different language. The package assumes the two languages under consideration are always Portuguese and English.

The package will sort the abstracts in the proper order. This means the first abstract will be in the same language as the main text, followed by the abstract in the other language, and then followed by the main text.

The abstract is critical because many researchers will read only that part. Your abstract should provide an accurate and sufficiently detailed summary of your work so that readers will understand what you did, why you did it, what your findings are, and why your findings are useful and important. The abstract must be able to stand alone as an overview of your study that can be understood without reading the entire text. However, your abstract should not be overly detailed. For example, it does not need to include a detailed methods section.

Even though the abstract is one of the first parts of the document, it should be written last. You should write it soon after finishing the other chapters, while the rest of the manuscript is fresh in your mind.

The abstract should not contain bibliography citations, tables, charts or diagrams. Give preference to the use of the verbs in the third person singular. Time and word must not dissociate yourself within the abstract. Abbreviations should be limited. Abbreviations that are defined in the abstract will need to be defined again at first use in the main text.

Finally, you must avoid the use of expressions such as "The present work deals with ... ", "In this thesis are discussed ", "The document concludes that ", "apparently and " etc.

The word limit should be observed, 300 words is the limit.

Abstracts are usually followed by a list of keywords selected by the author. Choosing appropriate keywords is important, because these are used for indexing purposes. Well-chosen keywords enable your manuscript to be more easily identified and cited.

Keywords: Keywords (in English) ...

Resumo

Independentemente da língua em que está escrita a dissertação, é necessário um resumo na língua do texto principal e um resumo noutra língua. Assume-se que as duas línguas em questão serão sempre o Português e o Inglês.

O *template* colocará automaticamente em primeiro lugar o resumo na língua do texto principal e depois o resumo na outra língua. Por exemplo, se a dissertação está escrita em Português, primeiro aparecerá o resumo em Português, depois em Inglês, seguido do texto principal em Português.

Resumo é a versão precisa, sintética e selectiva do texto do documento, destacando os elementos de maior importância. O resumo possibilita a maior divulgação da tese e sua indexação em bases de dados.

A redação deve ser feita com frases curtas e objectivas, organizadas de acordo com a estrutura do trabalho, dando destaque a cada uma das partes abordadas, assim apresentadas: Introdução - Informar, em poucas palavras, o contexto em que o trabalho se insere, sintetizando a problemática estudada. Objectivo - Deve ser explicitado claramente. Métodos - Destacar os procedimentos metodológicos adoptados. Resultados - Destacar os mais relevantes para os objectivos pretendidos. Os trabalhos de natureza quantitativa devem apresentar resultados numéricos, assim como seu significado estatístico. Conclusões - Destacar as conclusões mais relevantes, os estudos adicionais recomendados e os pontos positivos e negativos que poderão influir no conhecimento.

O resumo não deve conter citações bibliográficas, tabelas, quadros, esquemas. Dar preferência ao uso dos verbos na 3ª pessoa do singular. Tempo e verbo não devem dissociar-se dentro do resumo. Deve evitar o uso de abreviaturas e siglas - quando absolutamente necessário, citá-las entre parênteses e precedidas da explicação de seu significado, na primeira vez em que aparecem.

E, deve-se evitar o uso de expressões como "O presente trabalho trata ...", "Nesta tese são discutidos....", "O documento conclui que....", "aparentemente é...."etc.

Existe um limite de palavras, 300 palavras é o limite.

Para indexação da tese nas bases de dados e catálogos de bibliotecas devem ser apontados pelo autor as palavras-chave que identifiquem os assuntos nela tratados. Estes permitirão a recuperação da tese quando da busca da literatura publicada.

Palavras-chave: Palavras-chave (em português) ...

Contents

List of Figures	xvii
List of Listings	xix
1 Introduction	1
1.1 Motivation	1
1.2 Use case	3
1.3 Document Organization	4
2 State of Art	5
2.1 XSD Language	5
2.2 Template Engines	6
2.2.1 Xmllet Similarities	6
2.2.2 J2html	7
2.2.3 Apache Velocity	7
3 Solution	9
3.1 XsdParser	9
3.1.1 Parsing Strategy	10
3.1.2 Reference solving	14
3.2 XsdAsm	16

3.2.1	Supporting Infrastructure	16
3.2.2	Code Generation Strategy	18
3.2.3	Restriction Validation	20
3.2.3.1	Enumerations	23
3.2.4	Element Binding	24
3.3	Client	26
3.3.1	HtmlApi	26
3.3.1.1	Using the HtmlApi	29
4	Deployment	33
4.1	Github Organization	33
4.2	Maven	33
4.3	Sonarcloud	34
4.4	Testing metrics	34
5	Conclusion	37
5.1	Future work	37

List of Figures

1.1	Error validation in compile time	3
3.1	API - Supporting Infrastructure	17

List of Listings

1.1	Failed HTML rule validation	2
1.2	Lack of rule validation	2
3.1	XsdAnnotation class (Simplified)	10
3.2	DOM Document Parsing	10
3.3	XsdParser Node Parsing Process	11
3.4	XsdAttribute Information Extraction (Simplified)	11
3.5	XsdParseSkeleton Parsing Children From a Node	12
3.6	Parsing Concrete Example	13
3.7	Reference Solving Example	15
3.8	Code Generation XSD Example	18
3.9	Html Element Class	19
3.10	Body Element Class	19
3.11	Head Element Class	19
3.12	Manifest Attribute Class	19
3.13	CommonAttributeGroup Interface	20
3.14	Restrictions Example	21
3.15	Attribute Class Example	21
3.16	Attribute Static Constructor Restrictions	21
3.17	BaseAttribute Rule Validation Restrictions	22
3.18	Restriction Validator Class (Simplified)	23

3.19 Enumeration XSD Definition	23
3.20 Enumeration Class	24
3.21 Attribute Receiving An Enumeration	24
3.22 Binder Usage Example	24
3.23 Visitor with binding support	25
3.24 API creation	26
3.25 Maven API compile classes plugin	27
3.26 Maven API creation batch file (create_class_binaries.bat)	27
3.27 Maven API decompile classes plugin	28
3.28 Maven API decompile batch file (decompile_class_binaries.bat)	28
3.29 Custom Visitor	29
3.30 HtmlApi Element Tree	30
3.31 HtmlApi Visitor Result	31

1

Introduction

The work described in this dissertation is concerned with the implementation of a Java solution, named `xmllet`, that allows the automatic generation of a fluent *Application Programming Interface* (API) based on a *eXtensive Markup Language Schema Definition* (XSD) file. The generated classes are very similar most of the time and such solution may save time to the programmer, eliminating repetitive tasks and human error.

1.1 Motivation

Text has evolved with the advance of technology resulting in the creation of markup languages [?]. 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 IntelliSense¹ 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 don't actually enforce the language rules. In the following *HyperText Markup Language* (HTML)

¹[Intellisense Definition](#)

example there is a violation of HTML rules, a `<html>` tag containing a `<div>` tag, which is not allowed (Listing 1.1).

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

Listing 1.1: Failed HTML rule validation

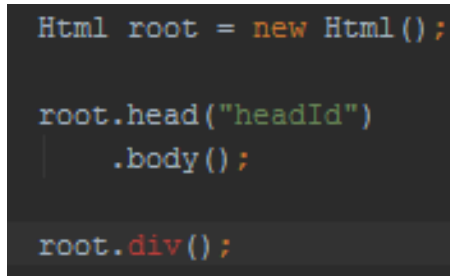
The solution to having documents that respect the markup language rules is changing the way the document writing works. As long as the writing process is fully controlled by the programmer errors will always happen because even though most text editors present the errors its corrections depend on the programmer correcting them. The suggestion presented is to move the writing control to an entity which can enforce the markup languages restrictions. This way it is guaranteed that the programmer won't be able to produce a document with errors.

Moving from the HTML representation to a Java representation of the issue there are multiple solutions to solve it. In this first code sample the programmer is allowed to add any child to the `Html` element (Listing 1.2), resulting in a violation of the language restrictions, which is not detected unless the programmer verifies the result manually.

```
1  Html root = new Html();
2  root.add(new Div());
```

Listing 1.2: Lack of rule validation

To solve this problem the entity, `Html` class, should restrict the children and attributes that accepts based on the existing restrictions on the HTML5 specification. As shown in the Figure 1.1 the generated API enforces the language restrictions in compile time, this way guaranteeing that any document generated will be compliant with the respective specification.



```
Html root = new Html();

root.head("headId")
    .body();

root.div();
```

Figure 1.1: Error validation in compile time

This solution even though apparently solving the main problem introduces a new one. The new problem resides on the fact that manually recreating all the rules of a given markup language is normally a very long process since most markup languages have a vast number of elements, attributes and restrictions.

The solution for this new problem is automation. An automated process that converts the markup language definition of elements, its attributes and restrictions to classes that represent those elements and methods which enforce the specification rules. With this automated process the application can generate a fluent API that allow the users to write their texts in a fluent way without errors and respecting the markup language specification. This is the main objective of this work, creating an infrastructure that reads a markup language definition, from a XSD file, and generates an API that allows the users to write well formed documents.

1.2 Use case

The use case that will be used to test and evaluate the solution will be the HTML5 XSD. In this case there are multiple elements that share behavior and/or attributes that can be generated automatically. The generated classes allow to create a tree of elements that represent a *eXtensive Markup Language* (XML) document. The resulting tree of elements can then be processed in different ways by using the Visitor pattern. This way each different Visitor implementation can use the generated tree of elements to write HTML documents to a stream, a socket or a file.

The generated HTML5 elements API will then be used in the HtmlFlow API, which is also a fluent Java API that is used to write well formed HTML files. With the Visitor pattern the HtmlFlow will only need to implement its own Visitor to achieve its goal. At the moment the HtmlFlow library only supports a set of the HTML elements which were created manually and the rest of the library

interacts with those elements in order to write the HTML files. By using the solution which will be developed in this work the HtmlFlow will support the whole HTML syntax.

1.3 Document Organization

This document will be separated in five distinct chapters. The first chapter, this one, introduces the problem that was presented. The second chapter presents existent technology that is relevant to this solution. The third chapter explains in detail the different components of the suggested solution. The fourth chapter approaches the deployment and testing of the suggested solution. The fifth and last chapter of this document contains some final remarks and description of future work.



State of Art

In this chapter we are going to introduce the XSD language in order to provide a better understanding of the next chapters and also introduce some tools that were discovered during the development of this dissertation.

2.1 XSD Language

The XSD language is a description of a type of XML document. Its purpose is to create a set of rules and constraints that a given type of XML document must follow in order to be considered valid. These rules are meant to create a contract on the type of information contained in the XML documents, apart from having well formed XML. To describe the rules and restrictions for a given XML document the XSD language relies on two 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 language. There are multiple elements and attributes present in the XSD language, which are specified at [XSD Schema](#). In this dissertation we will use the set of rules and restrictions of the XSD files provided to build a fluent API that will enforce the rules and restrictions specified by the given file.

2.2 Template Engines

Templates engines are systems based on template files. A template file contains two types of information:

1. Static - This information doesn't change and should always be present.
2. Dynamic - This information depends on external input.

By using template files, the template engines are able to separate the presentation aspect of a given project from its business logic. This is usually important since by having these aspects separated posterior changes to the visual presentation will be implemented faster since there are very few dependencies between the project different layers. By using this approach bring other advantages, such as:

1. Performance - By reusing the template files the engine avoids repetitive tasks.
2. Deployment Speed - It becomes easier to create new outputs, such as new web pages, fill a document, etc.

Template engines have been around for a long time, such as *JavaServer Pages* (JSP)¹ used with the Java language back in 1999, but gained a new significance with its application on the world of Web technologies. Multiple solutions have appeared that run the substitution process either on the client side of the web sites or all the way on the server side. Apart from this kind of usage there are also templates for full websites that can be used by people that don't have a real understanding of programming and website design. This kind of pre-made websites can either be used freely but more complex solutions are sold by companies that specialize on this kind of product.

In the Sections 2.2.2 and 2.2.3 we introduce two solutions, J2html and Apache Velocity, which may be used in a similar way to `xmlet`.

2.2.1 Xmlet Similarities

Even though `xmlet` doesn't aim to be a template engine, both technologies share a few similarities. What similarities does `xmlet` share with most template engines?

¹[JavaServer Pages](#)

- Template definition - `xmlet` can define a template within the Java language, removing the need to use external files which usually implies using an extra "language" or syntax in order to define the template. This template definition has the disadvantage of being tied to a concrete language such as Java, which shouldn't be problematic if a project isn't supposed to change language.
- Template substitution - `xmlet` can also define a template to receive a certain type of data in order to fill the dynamic information of any given template. This should be faster than the regular behaviour of template engines since `xmlet` doesn't need to read its template from a file on the file system.

2.2.2 J2html

J2html² is a Java library used to write HTML, a very similar solution to the used case presented in Section 1.2. The main difference between the two solutions are that the J2html does not verify the specification rules of the HTML language either at compile time or at runtime. This library also shows that the issue we are trying to solve with this dissertation is relevant since this library has quite a few forks and watchers on their github page³. In Chapter 4 we will present some performance tests to verify if our solution is more efficient at writing HTML.

2.2.3 Apache Velocity

Apache Velocity⁴ is a template engine that we discovered through J2html. Even though the `xmlet` solution doesn't define itself as a template engine it can also be used to such extent. Since the template engines are based on a template file, with some sort of code embedded in the language (HTML for example) the same result can be obtained by using the solution presented in this project. This solution improves the template engine solutions by allowing the users to define the exact same aspects defined in the template files directly into code, allowing the verification of the "template" at compile time by the language compiler. This reduces the overall complexity by removing possible errors in the template files and removing the necessity to separate template files and actual application code, while

²J2html

³J2html Github Page

⁴Apache Velocity

enforcing the language specification. In Chapter 4 we will also compare this solution with the J2html and the `xmlet` solution.

3

Solution

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

1. Parsing the information from the XSD file;
2. Generating the API 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`.

3.1 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 2.1. 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 3.1.

```
1 public class XsdAnnotation extends XsdIdentifierElements {  
2  
3     //The id field is inherited from XsdIdentifierElements.  
4     private List<XsdAppInfo> appInfoList = new ArrayList<>();  
5     private List<XsdDocumentation> documentations = new ArrayList<>();
```

Listing 3.1: XsdAnnotation class (Simplified)

3.1.1 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. 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 into a node list, performing a single read on the XSD file, avoiding multiple reads which is less efficient (Listing 3.2).

```
1 DocumentBuilderFactory dbFactory= DocumentBuilderFactory.newInstance();  
2 DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();  
3 //Parses the XSD file  
4 Document doc = dBuilder.parse(xsdFile);  
5 //Obtains the XSD file node list  
6 NodeList nodes = doc.getFirstChild().getChildNodes();
```

Listing 3.2: DOM Document Parsing

Then parsing the continues by iterating this list (i.e. nodes) and obtaining the name of the element represented by that node, e.g. `xsd:element` or `xsd:complexType`. The name of the element will be needed to perform a lookup search to find the corresponding parsing function (Listing 3.3).

```

1 stream(nodes)
2   .filter(node ->
3       node.getNodeType() == Node.ELEMENT_NODE &&
4       parseMappers.get(node.getNodeName()) != null)
5   .map(node ->
6       parseMappers.get(node.getNodeName())
7           .apply(node))
8   .forEach(elements::add);

```

Listing 3.3: XsdParser Node Parsing Process

From that moment on each element obtains all its attribute information directly from its node object (Listing 3.4).

```

1 public class XsdAttribute extends XsdReferenceElement {
2     private XsdAttribute(Map<String, String> elementFieldsMapParam) {
3         setFields(elementFieldsMapParam);
4     }
5
6     @Override
7     public void setFields(Map<String, String> elementFieldsMapParam) {
8         super.setFields(elementFieldsMapParam);
9
10        this.fixed = elementFieldsMap.getOrDefault(FIXED_TAG, fixed);
11        this.type = elementFieldsMap.getOrDefault(TYPE_TAG, type);
12        this.form = elementFieldsMap.getOrDefault(FORM_TAG, form);
13        this.use = elementFieldsMap.getOrDefault(USE_TAG, "optional");
14        this.defaultElement = elementFieldsMap.getOrDefault(
15            DEFAULT_ELEMENT_TAG, defaultElement);
16    }
17
18    public static ReferenceBase parse(Node node) {
19        NamedNodeMap nodeAttributes = node.getAttributes();
20        Map<String, String> attrMap = convertNodeMap(nodeAttributes);
21        XsdAttribute attribute = new XsdAttribute(attrMap);
22
23        return xsdParseSkeleton(node, attribute);
24    }

```

Listing 3.4: XsdAttribute Information Extraction (Simplified)

Regarding the other elements that may be contained in a given node they are all similarly parsed. The `xsdParseSkeleton` function existing in the `XsdAbstractElement` class (Listing 3.5) will iterate in all the children of a given node, invoke the respective parse function of each one and then notify the parent element, using the Visitor pattern, so that the parent element can perform the changes needed based on the element received.

```

1 static ReferenceBase xsdParseSkeleton(Node node, XsdAbstractElement
   element){
2     Node child = node.getFirstChild();
3
4     //Iterates in all children from the received Node object, node.
5     while (child != null) {
6         //Only parses element nodes, ignoring comments and text nodes.
7         if (child.getNodeType() == Node.ELEMENT_NODE) {
8             //Obtains the name of the node, e.g. xsd:element.
9             String nodeName = child.getNodeName();
10
11             //Searches on a mapper for a parsing functions
12             //for the respective type.
13             Function<Node, ReferenceBase> parserFunction = XsdParser.
                getParseMappers().get(nodeName);
14
15             //Applies the parsing functions, if any, and notifies
16             //the parent objects Visitor to the newly created object.
17             if (parserFunction != null){
18                 parserFunction.apply(child)
19                     .getElement()
20                     .accept(element.getVisitor());
21             }
22         }
23
24         //Moves on to the next sibling.
25         child = child.getNextSibling();
26     }
27
28     //Wraps the element in a ReferenceBase object,
29     //which will be explained further ahead.
30     return ReferenceBase.createFromXsd(element);
31 }

```

Listing 3.5: XsdParseSkeleton 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 concrete example with the XSD code present in Listing 3.6.

```
1 <xsd:element>
2   <xsd:complexType id="complexId">
3     <!-- (...) -->
4   </xsd:complexType>
5 </xsd:element>
```

Listing 3.6: Parsing Concrete Example

Step 1 - DOM parsing:

The parsing starts with the DOM library parsing the code (Listing 3.6), which returns a node list (i.e. nodes) containing only one node, the `xsd:element` node. Using the `xsd:element` string the `XsdElement` parse function will be obtained from an existent string to function mapper (i.e. `parseMappers`).

Step 2 - XsdElement Attribute Parsing:

The `XsdElement` parse function receives the `Node` object and extracts all the attribute information (similar to the example presented in Listing 3.4), which in this case is empty since the element has no attributes.

Step 3 - XsdElement Children:

To parse the `XsdElement` children the `XsdAbstractElement` `xsdParseSkeleton` function is called (Listing 3.5) and starts to iterate the `xsd:element` node children, which is a node list containing a single element, the `xsd:complexType` node.

Step 4 - XsdComplexType Attribute Parsing:

The parsing of the `xsd:complexType` node is similar to `xsd:element`, it extracts the attribute information from its respective node, in this case it will obtain a value from the node attribute named `id` and assigning it to the `id` field of the `XsdComplexType` object (similar to the example presented in Listing 3.4).

Step 5 - XsdElement Visitor Notification:

After parsing the `xsd:complexType` node the previously created `XsdElement` object is notified. This notification informs the `XsdElement` object that it contains the newly created `XsdComplexType` object using the Visitor pattern. The `XsdElement` should then act accordingly based on the type of the object received as his children, since different types of objects should be treated differently.

This whole behaviour is shared by all the classes that represent a XSD element. The Visitor pattern is a very important tool in the parsing process since it allows each element to define a different behaviour for each element received as children. This is also useful to implement the schema rules, since it can be used to define empty methods to reject any children that a given element type shouldn't contain as per definition on the schema specification. The same happens to the attributes present in the parsed DOM nodes, each XsdParser object only extracts the attributes defined in the XSD schema specification, ignoring other attributes present.

3.1.2 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 was not found in the file.
- `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 3.7.

```

1 <?xml version='1.0' encoding='utf-8' ?>
2 <xsd:schema xmlns='http://schemas.microsoft.com/intellisense/html-5'
   xmlns:xsd='http://www.w3.org/2001/XMLSchema'>
3
4   <!-- NamedConcreteType wrapping a XsdGroup -->
5   <xsd:group id="replacement" name="flowContent">
6     <!-- (...) -->
7   </xsd:group>
8
9   <!-- ConcreteElement wrapping a XsdChoice -->
10  <xsd:choice>
11    <!-- UnsolvedReference wrapping a XsdGroup -->
12    <xsd:group id="toBeReplaced" ref="flowContent"/>
13  </xsd:choice>
14 </xsd:schema>

```

Listing 3.7: Reference Solving Example

In this short example we have a XsdChoice element that contains a XsdGroup element with a reference 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 getUnsolvedReferencesForFile(String filePath), 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.

3.2 XsdAsm

XsdAsm is a library dedicated to generate a fluent Java API 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 ASM¹ library, which is a library that provides a Java interface to bytecode manipulation, which provide method 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, 6.1, was release in 11 march 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.

3.2.1 Supporting Infrastructure

To support the foundations of the XSD language an infrastructure is created in every API 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 and adds restriction verification to all the deriving classes. All the attributes that have restrictions should derive from this class.

¹[ASM Website](#)

Visitor classes:

- **ElementVisitor** - An interface that defines methods for all the generated elements that can be visited with the Visitor pattern. All the methods have a default implementation that point to a single method. This behaviour aims to reduce the amount of code needed to create concrete implementations of Visitors.

Taking in consideration those classes, a very simplistic API could be represented with the class diagram (Figure 3.1). In this example we have an element, `Html`, that extends `AbstractElement` and an attribute, `AttrManifestString`, that extends `BaseAttribute`.

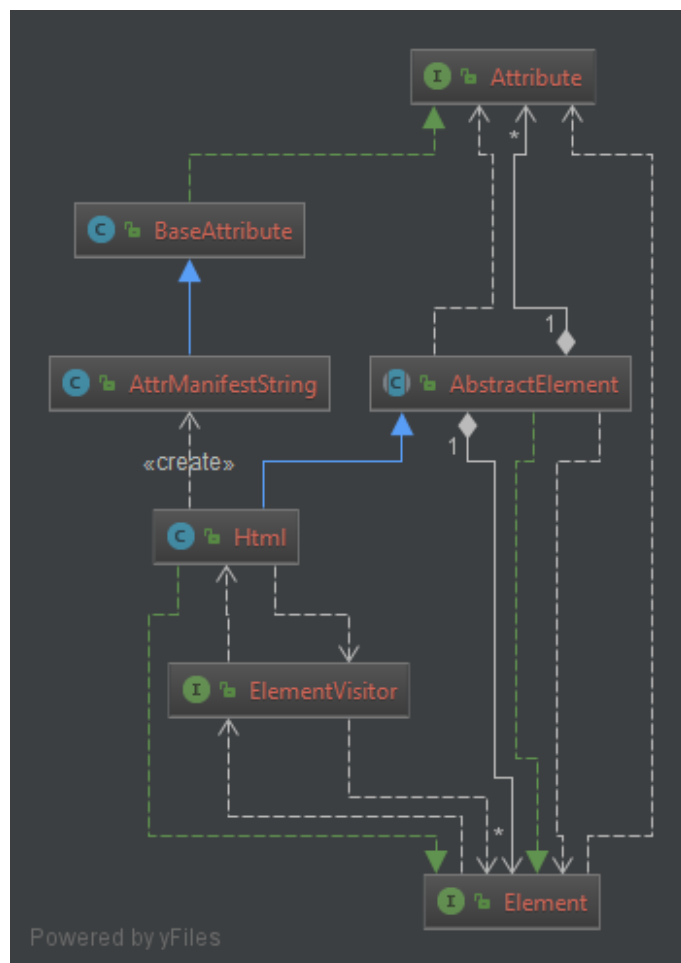


Figure 3.1: API - Supporting Infrastructure

3.2.2 Code Generation Strategy

To understand how most of this project works a XSD example (Listing 3.8) and a detailed explanation is provided. In this example there will be some simplifications making it easier to understand how the library works internally.

```
1 <xs:element name="html">
2
3   <xs:attributeGroup name="commonAttributeGroup">
4     <xs:attribute name="someAttribute" type="xs:string">
5   </xs:attributeGroup>
6
7   <xs:complexType>
8     <xs:choice>
9       <xs:element ref="body"/>
10      <xs:element ref="head"/>
11    </xs:choice>
12    <xs:attributeGroup ref="commonAttributeGroup" />
13    <xs:attribute name="manifest" type="xs:string" />
14  </xs:complexType>
15
16 </xs:element>
```

Listing 3.8: Code Generation XSD Example

With this example there is a multitude of classes that need to be created, apart from the always present supporting infrastructure presented in Section 3.2.1.

- **Html Element** - A class that represents the `Html` element (Listing 3.9), deriving from `AbstractElement`.
- **Body and Head Methods** - Both methods present in the `Html` class (Listing 3.9) that add `Body` (Listing 3.10) and `Head` (Listing 3.11) instances to `Html` children list.
- **Manifest Method** - A method present in `Html` class (Listing 3.9) that adds an instance of the `Manifest` attribute (Listing 3.12) to the `Html` attribute list.

```

1 public class Html extends AbstractElement implements
    CommonAttributeGroup {
2     public Html() { }
3
4     public Html attrManifest(String attrManifest) {
5         this.addAttr(new AttrManifest(attrManifest));
6     }
7
8     public Body body() { this.addChild(new Body()); }
9
10    public Head head() { this.addChild(new Head()); }
11 }

```

Listing 3.9: Html Element Class

- Body and Head classes - Classes for both Body (Listing 3.10) and Head (Listing 3.11) elements, similar to the generated Html class (Listing 3.9). The class contents will be dependent on the contents present in the concrete `xsd:element` nodes.

```

1 public class Body extends AbstractElement {
2     //Similar to Html, based on the contents of the respective
3     //xsd:element node.
4 }

```

Listing 3.10: Body Element Class

```

1 public class Head extends AbstractElement {
2     //Similar to Html, based on the contents of the respective
3     //xsd:element node.
4 }

```

Listing 3.11: Head Element Class

- Manifest Attribute - A class that represents the Manifest attribute (Listing 3.12), deriving from BaseAttribute.

```

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

```

Listing 3.12: Manifest Attribute Class

- **CommonAttributeGroup Interface** - An interface with default methods that add the group attributes to the concrete element (Listing 3.13).

```
1 public interface CommonAttributeGroup extends Element {  
2     default Html attrSomeAttribute(String attributeValue) {  
3         this.addAttr(new SomeAttribute(attributeValue));  
4         return this;  
5     }  
6 }
```

Listing 3.13: CommonAttributeGroup Interface

As we can see from the previous example 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 API to navigate in the element tree while maintaining type information which is essential to guarantee the specific language restrictions.

3.2.3 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 can't be validated in compile time, an example of this is the following restriction (Listing 3.14):

```

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

```

Listing 3.14: Restrictions Example

In this example (Listing 3.14) 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 3.15):

```

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

```

Listing 3.15: Attribute Class Example

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 static constructor, which stores the restrictions in a static Map object as showed in Listing 3.16.

```

1 static {
2   restrictions = new ArrayList<Map<String, Object>>();
3   HashMap<String, Object> restriction = new HashMap<>();
4   restriction.put("MaxLength", Integer.valueOf(5));
5   restriction.put("MinLength", Integer.valueOf(1));
6   restrictions.add(restriction);
7 }

```

Listing 3.16: Attribute Static Constructor Restrictions

By using this strategy the restrictions can be validated whenever an instance of a concrete attribute is created. To enforce the restrictions present in the Map object the `BaseAttribute` constructor (Listing 3.17) will pass those values to a class, `RestrictionValidator`, which validates all the different types of restrictions, in this case `xs:maxlength` and `xs:minlength`. Each different restriction has its validation method (i.e. `validateMaxLength` and `validateMinLength` at Listing 3.18 lines 6 and 11, respectively) which will throw an exception if the value (i.e. `val`) to validate does not match the restriction. By using this strategy the API ensures that any successful usage follows the rules previously defined by the schema.

```
1 public class BaseAttribute<T> implements Attribute<T> {
2     static List<Map<String, Object>> restrictions = new ArrayList();
3
4     public BaseAttribute(T var1, String var2) {
5         // ...
6         restrictions.forEach(this::validateRestrictions);
7     }
8
9     private void validateRestrictions(Map<String, Object> restriction){
10         Object value = this.getValue();
11
12         if (value instanceof String) {
13             RestrictionValidator.validate(restriction, (String)value);
14         }
15
16         if (value instanceof Integer || value instanceof Short ||
17             value instanceof Float || value instanceof Double) {
18             RestrictionValidator.validate(restriction, (Double)value);
19         }
20
21         if (value instanceof List) {
22             RestrictionValidator.validate(restriction, (List)value);
23         }
24     }
25 }
```

Listing 3.17: BaseAttribute Rule Validation Restrictions

```

1 public class RestrictionValidator {
2     static void validate(Map<String, Object> restMap, List val) {
3         validateMinLength(restMap.getDefault("MinLength", -1), val);
4         validateMaxLength(restMap.getDefault("MaxLength", -1), val);
5     }
6     private static void validateMaxLength(int maxLength, List list) {
7         if (maxLength != -1 && list.size() > maxLength) {
8             throw new RestrictionViolationException("Violation of
9                 maxLength restriction.");
10        }
11    }
12    private static void validateMinLength(int minLength, List list) {
13        if (minLength != -1 && list.size() < minLength) {
14            throw new RestrictionViolationException("Violation of
15                minLength restriction.");
16        }
17    }
18 }

```

Listing 3.18: Restriction Validator Class (Simplified)

3.2.3.1 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` tags. In the following example (Listing 3.19) we have an attribute with three possible values, command, checkbox and radio.

```

1 <xs:attribute name="type">
2     <xs:simpleType>
3         <xs:restriction base="xsd:string">
4             <xs:enumeration value="command" />
5             <xs:enumeration value="checkbox" />
6             <xs:enumeration value="radio" />
7         </xs:restriction>
8     </xs:simpleType>
9 </xs:attribute>

```

Listing 3.19: Enumeration XSD Definition

This results in the creation of an Enum, `EnumTypeCommand` (Listing 3.20). The attribute class will then receive an instance of `EnumTypeCommand`, ensuring that only allowed values are used (Listing 3.21).

```

1 public enum EnumTypeCommand {
2     COMMAND (String.valueOf("command")),
3     CHECKBOX (String.valueOf("checkbox")),
4     RADIO (String.valueOf("radio"))
5 }

```

Listing 3.20: Enumeration Class

```

1 public class AttrTypeEnumTypeCommand extends BaseAttribute<String> {
2     public AttrTypeEnumTypeCommand(EnumTypeCommand attrValue) {
3         super(attrValue.getValue());
4     }
5 }

```

Listing 3.21: Attribute Receiving An Enumeration

3.2.4 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 3.22 using the HTML5 API.

```

1 public class BinderExample{
2     public void bindExample(){
3         Html<Html> root = new Html<>();
4         Body<Html<Html>> body = root.body();
5
6         Table<Body<Html<Html>>> table = body.table();
7         table.tr().th().text("Title");
8         table.<List<String>>binder((elem, list) ->
9             list.forEach(tdValue ->
10                 elem.tr().td().text(tdValue)
11             )
12         );
13         //Keep adding elements to the body of the document.
14     }
15 }

```

Listing 3.22: Binder Usage Example

In this example we create a table and add 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 what will the element do when the information is received. This is achieved by implementing a Visitor that supports binding.

In Listing 3.23 we can observe how the Visitor would work. It maintains the default behaviour on the elements that aren't 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 3.22) to the clone, effectively executing the function supplied in the previously called binder method (i.e. Listing 3.22 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 isn't affected since all these changes are performed in a clone of the bound element, meaning that the template can be reused.

```
1 public <T extends Element> void sharedVisit(Element<T, ?> element) {
2     // ...
3
4     if(element.isBound()) {
5         List<Element> children = element.cloneElem()
6                                     .bindTo(model)
7                                     .getChildren();
8         children.forEach( child -> child.accept(this));
9     } else {
10        element.getChildren().forEach(item -> item.accept(this));
11    }
12
13    // ...
14 }
```

Listing 3.23: Visitor with binding support

3.3 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 API is generated using the XsdAsm library and how to use the resulting API.

3.3.1 HtmlApi

To generate the HTML5 API 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 API name (Listing 3.24).

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

Listing 3.24: API creation

The second option is using the Maven² build lifecycle³ to make that same invocation by adding an extra execution to the *Project Object Model* (POM) file (Listing 3.25) to execute a batch file that invokes the XsdAsm main method (Listing 3.26).

²Maven

³Maven Build Lifecycle

```

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 3.25: Maven API compile classes plugin

```

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

```

Listing 3.26: Maven API creation batch file (create_class_binaries.bat)

This client uses the Maven lifecycle option by adding an execution at the validate phase (Listing 3.25, line 8) which invokes XsdAsm main method to create the API. 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 Fernflower⁴ decompiler, the Java decompiler used by IntelliJ⁵ *Integrated*

⁴Fernflower Decompiler

⁵IntelliJ IDE

Development Environment (IDE), to decompile the classes that were automatically generated (Listing 3.27, 3.28).

```

1 <!-- Plugin Information -->
2 <execution>
3   <id>decompile_classes</id>
4   <phase>validate</phase>
5   <goals>
6     <goal>
7       exec
8     </goal>
9   </goals>
10  <configuration>
11    <executable>
12      ${basedir}/decompile_class_binaries.bat
13    </executable>
14  </configuration>
15 </execution>

```

Listing 3.27: Maven API decompile classes plugin

```

1 if not exist ".\src\main\java\org\xmllet\htmlapi" mkdir ".\src\main\java
  \org\xmllet\htmlapi"
2
3 call
4   mvn exec:java
5     -D"exec.mainClass"="org.jetbrains.java.decompiler.main.decompiler.
      ConsoleDecompiler"
6     -D"exec.args"="-dgs=true .\target\classes\org\xmllet\htmlapi .\src\
      main\java\org\xmllet\htmlapi"
7
8 if exist ".\target\classes\org" rmdir ".\target\classes\org" /s /q

```

Listing 3.28: Maven API decompile batch file (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 API in the future, also allows to find any problem with the generated code since it forces the compilation of all the classes previously generated.

3.3.1.1 Using the HtmlApi

After the previously described compilation process of the HtmlApi project we are ready to use the generated API. To start using it the first step is to implement a custom Visitor class, which defines what to do when the created element tree is visited. A very simple example is presented in Listing 3.29 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.

```

1 public class CustomVisitor<R> implements ElementVisitor<R> {
2
3     private PrintStream printStream = System.out;
4
5     public CustomVisitor() { }
6
7     public <T extends Element> void sharedVisit(Element<T, ?> element)
8     {
9         printStream.printf("<%s", element.getName());
10
11         element.getAttributes()
12             .forEach(attribute ->
13                 printStream.printf(" %s=\"%s\"",
14                     attribute.getName(), attribute.getValue()));
15
16         printStream.print(">\n");
17
18         element.getChildren().forEach(item -> item.accept(this));
19
20         printStream.printf("</%s>\n", element.getName());
21     }
22 }
```

Listing 3.29: Custom Visitor

After creating the Visitor presented in Listing 3.29 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 `parent`, a short method name to keep the code as clean as possible. In Listing 3.30 we can see a code example that uses a good amount of the API 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 `°`.

```

1 Html<Html> root = new Html<>();
2
3 root.head()
4     .meta() .attrCharset("UTF-8") .°()
5     .title()
6         .text("Title") .°()
7     .link() .attrType(EnumTypeContentType.TEXT_CSS)
8         .attrHref("/assets/images/favicon.png") .°()
9     .link() .attrType(EnumTypeContentType.TEXT_CSS)
10        .attrHref("/assets/styles/main.css") .°() .°()
11 .body() .attrClass("clear")
12     .div()
13         .header()
14             .section()
15                 .div()
16                     .img() .attrId("brand")
17                         .attrSrc("./assets/images/logo.png") .°
18                             ()
19                     .aside()
20                         .em()
21                             .text("Advertisement")
22                             .span()
23                                 .text("HtmlApi is great!");
24 CustomVisitor customVisitor = new CustomVisitor();
25
26 customVisitor.visit(root);

```

Listing 3.30: HtmlApi Element Tree

With this element tree presented (Listing 3.30) and the previously presented `CustomVisitor` (Listing 3.29) we obtain the following result (Listing 3.31). The indentation was added for readability purposes, since the `CustomVisitor` implementation in Listing 3.29 does not indent the resulting HTML.

```
1 <html>
2   <head>
3     <meta charset="UTF-8">
4   </meta>
5   <title>
6     Title
7   </title>
8   <link type="text/css" href="/assets/images/favicon.png">
9   </link>
10  <link type="text/css" href="/assets/styles/main.css">
11  </link>
12 </head>
13 <body class="clear">
14   <div>
15     <header>
16       <section>
17         <div>
18           
19         </img>
20         <aside>
21           <em>
22             Advertisement
23           <span>
24             HtmlApi is great!
25           </span>
26         </em>
27       </aside>
28     </div>
29   </section>
30 </header>
31 </div>
32 </body>
33 </html>
```

Listing 3.31: HtmlApi Visitor Result

The `CustomVisitor` of Listing 3.31 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 come in, it implements the particular aspects of the HTML syntax in its `Visitor` implementation which deals with how and where the output is written.

4

Deployment

4.1 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 APIs are also created as if they belong to this organization.

4.2 Maven

In order to manage the developed projects a tool for project organization and deployment was used, named Maven. 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 Repository². All the releases of projects belonging to the `xmlet` Github organization can be found under using the artifactId, [com.github.xmlet](https://search.maven.org/artifact/com.github.xmlet).

¹[xmlet Github](#)

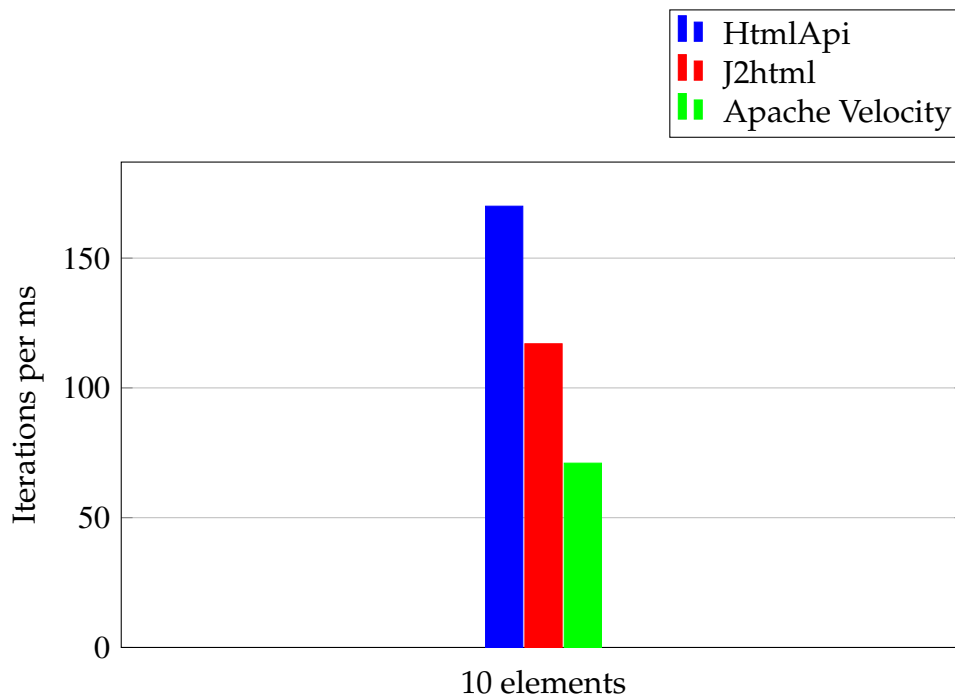
²[Maven Central Repository](#)

4.3 Sonarcloud

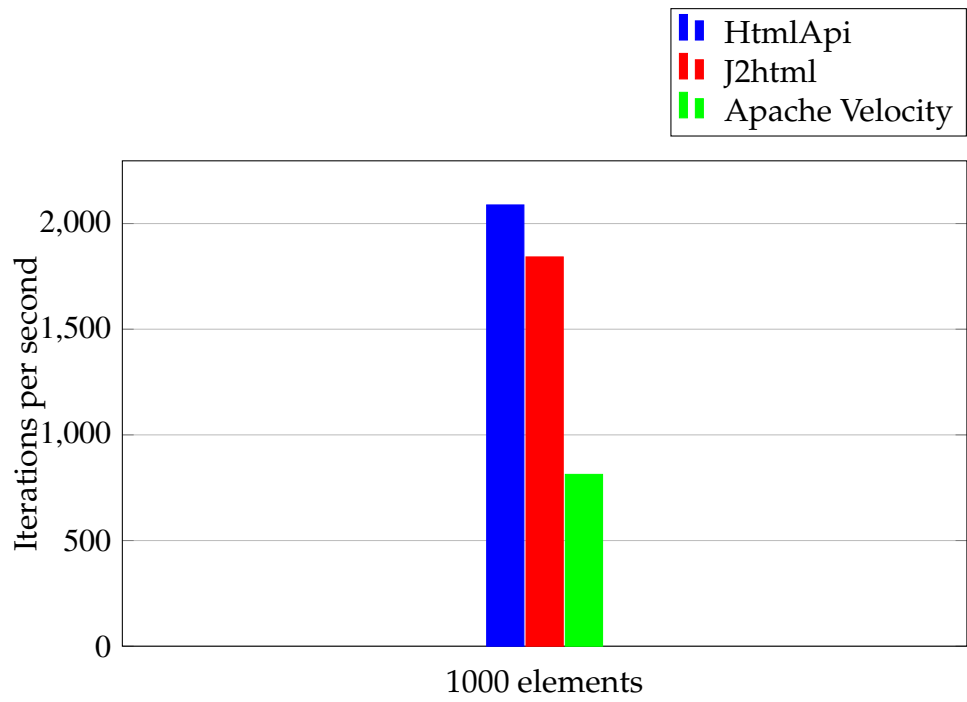
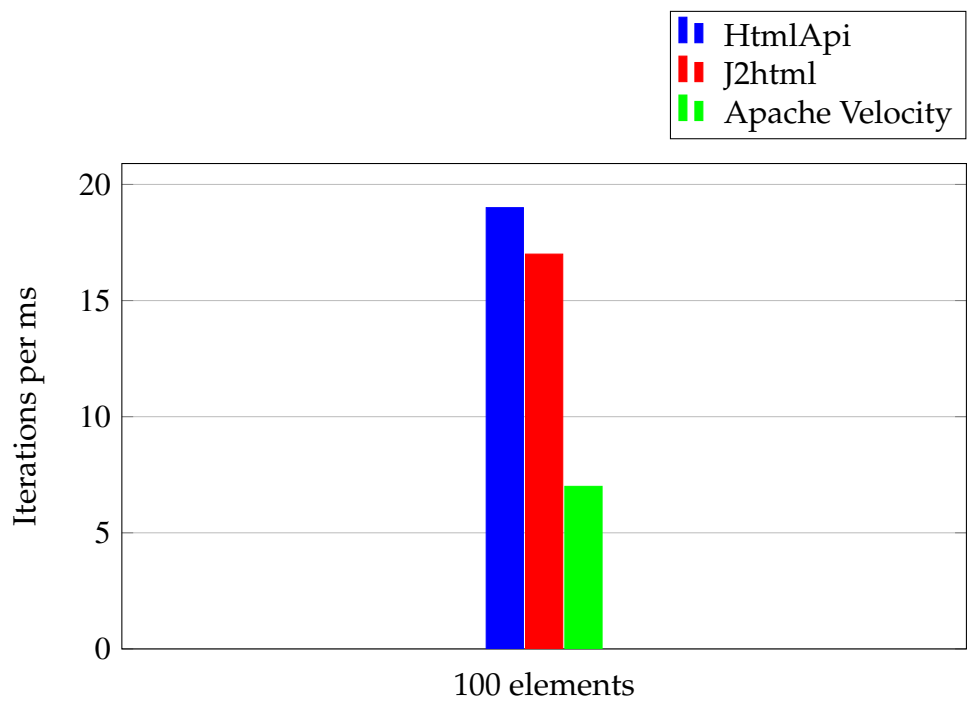
Code quality and its various implications such as security, low performance and bugs should always be an important issue to a programmer. With that in mind all the projects contained in the `xmlet` solution were evaluated in various metrics and the results made public for consultation. This way, either 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 Sonarcloud³, which provides free of charge evaluations and stores the results which are available for everyone.

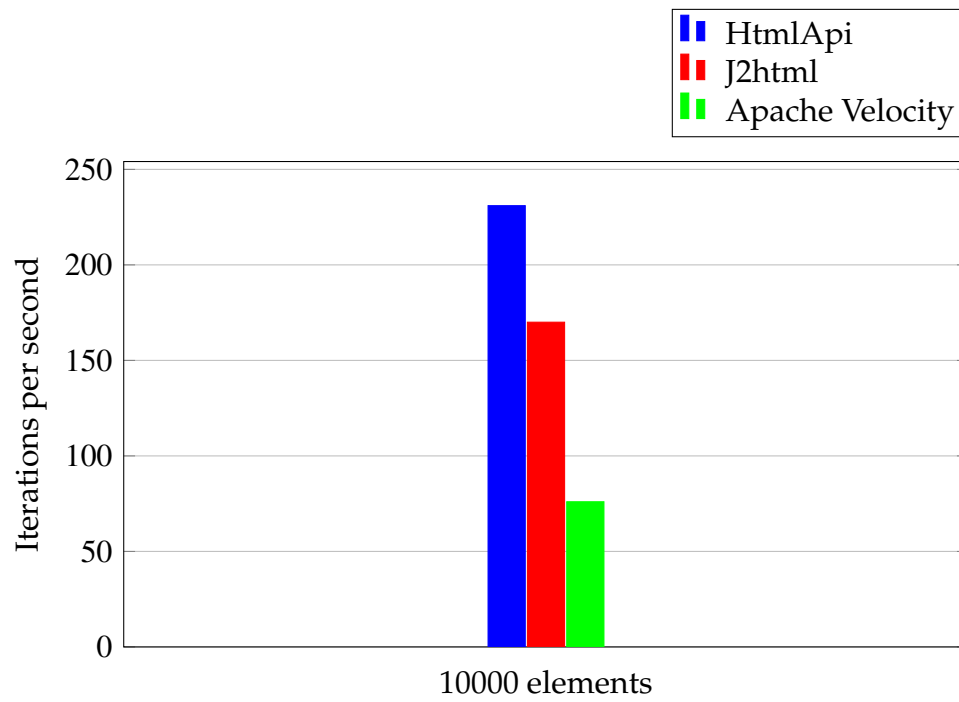
4.4 Testing metrics

Perform efficiency tests comparing the `HtmlApi`, `j2html` and `Apache Velocity`. The test should be based on a `html` page with multiple elements and attributes, probably the test should be performed with different number of `html` elements, like 10, 50, 100, 1000.



³[Sonarcloud xmlet page](#)







Conclusion

Here be conclusion.

5.1 Future work

Talk about adding support for `xsd:import` tag, moving the visitor calls to the insertion of elements and attributes in order to improve the efficiency of the resulting API.

