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

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.

Agradecimentos

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*	1
DOM	*Document Object Model*	8
HTML	*HyperText Markup Language*	1
SAX	*Simple Application Programming Interface for eXtensive Markup Language*	8
UML	*Unified Modeling Language*	13
XHTML	*eXtensive HyperText Markup Language*	8
XML	*eXtensive Markup Language*	3
XSD	*eXtensive Markup Language Schema Definition*	1

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) . . .

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) …

Índice

Lista de Figuras

Lista de Listagens

1 Introduction

The work described in this dissertation is concerned with the implementation of a Java solution, named xmlet, 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 user, eliminating repetitive tasks and human error.

1.1 Motivation

Text has evolved with the advance of technology resulting in the creation of markup languages [1]. 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 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) example there is a violation of HTML rules, a <html> tag containing a <div> tag, which isn't allowed.

```
<html>
  <div>
    (...)
  </div>
</html>
```

Listagem 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. If the user has control over the writing process errors can be induced in the document because even though most text editors present the errors its corrections depend on the user correcting them. The suggestion presented is to pass the writing control to an entity which can enforce the markup languages restrictions. This way it is guaranteed that the user can't produce a document with errors.

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

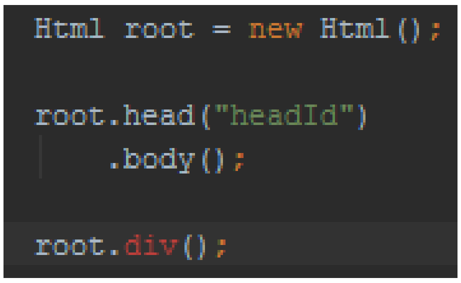
```
Html root = new Html();
root.add(new Div());
```

Listagem 1.2: Error validation in runtime

In order to resolve this problem the entity, Html class, should restrict the children and attributes that accepts based on the existing restrictions on the HTML5 specification. In the 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.

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 and restrictions.

The solution for this new problem is automation. An automated process that converts the definition of elements and restrictions of markups languages in classes that represent those elements and methods which enforce the specification rules.



```
Html root = new Html();

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

root.div();
```

Figura 1.1: Error validation in compile time

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 classes can then be processed in different ways since the Visitor pattern is used. This way each different Visitor implementation can use the generated classes 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. With the help of 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 are similar 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.

2 Existent Tools

A little introduction here

2.1 XSD Language

A little introduction on what is the XSD language, what are its purposes, where is it used, how does it work (elements, attributes).

2.2 J2html

What is it? What are the similarities? What are the improvements introduced by this dissertation that make it better or worse than j2html?

2.3 Apache Velocity

What is it? What are the similarities? What are the improvements introduced by this dissertation that make it better or worse than apache velocity?

3 Solution

In this chapter the suggested solution will be detailed, approaching its different components and how they interact between them. The process of generating an API based on a XSD file contents is complex, so in order to simplify it it was divided into two smaller projects, each one with a given goal and responsibilities. The two main responsibilities in this process are parsing the information from the XSD file and generating the API based on that same parsed information, which means that the code generation project will have a dependency on the parsing project. For writing purposes the projects will be further referenced by their name, the parsing project being named XsdParser and the code generation project named XsdAsm.

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[1]. For example, the xsd:annotation tag only allows xsd:appinfo and xsd:documentation as children nodes, and also can have an attribute named id, therefore XsdParser has the following class:

[1]http://www.datypic.com/sc/xsd/s-xmlschema.xsd.html

```java
public class XsdAnnotation extends XsdIdentifierElements {

   //The id field is inherited from XsdIdentifierElements.
   private List<XsdAppInfo> appInfoList = new ArrayList<>();
   private List<XsdDocumentation> documentations = new ArrayList<>();
}
```

Listagem 3.1: XsdAnnotation class (simplified)

3.1.1 Parsing Strategy

In order to perform the parsing of the file itself two options were researched, *Document Object Model* (DOM) and *Simple Application Programming Interface for eXtensive Markup Language* (SAX). Having balanced the functionalities of both 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 the XsdParser requires in order 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, in order to avoid multiple reads on the file, which is less efficient. The parsing then follows by iterating on this list 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 in order to perform a lookup search to find the corresponding parsing function. From that moment on each element obtains all its attribute information directly from its node object. Regarding the other elements that may be contained in a given node they are all similarly parsed. The skeleton function existing in the shared XsdAbstractElement class 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.

```xml
<xsd:element>
   <xsd:complexType id="complexId">
      (...)
   </xsd:complexType>
</xsd:element>
```

Listagem 3.2: Parsing Concrete Example

Based on the abstract explanation given above a more detailed one will be given detailing how the short snippet of XSD code will be parsed by the XsdParser.

Step 1 - DOM parsing:

The parsing starts with the DOM parsing of the example above, which returns a node list 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.

Step 2 - XsdElement Attribute Parsing:

The XsdElement parse function receives the Node object and extracts all the attribute information, which in this case is empty since the element has no attributes.

Step 3 - XsdElement Children:

In order to parse the XsdElement children the XsdAbstractElement xsdParseSkeleton function is called and starts to iterate in 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 is similar to the xsd:element, it extracts the attribute information from its respective node, in this case meaning that it will obtain a value from the node attribute named id and assigning it to the id field of the XsdCompleType object.

Step 5 - XsdElement Visitor Notification:

After the parsing of the xsd:compleType node is complete the previously created XsdElement is notified that it contains the newly created XsdCompleType 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 also be used to not perform any change if any given element receives other element as children that it shouldn't receive, effectively ignoring the elements that violate 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 performing the parsing process described previously, there is still an issue to solve regarding the references that exist in the XSD schema definition. In XSD files the usage of the ref attribute is frequent in order 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 in the file and the other being the replacement of the reference object by the referenced object when it is present. In order to effectively help resolve the referencing problem there were added some wrapper classes. These wrapper classes contain the wrapped element and serve as a classifier for the wrapped element. The existing wrapper classes are as follow:

- UnsolvedElement - A wrapper class to each element that was not found in the file.
- ConcreteElement - A wrapper class to each element that is present in the file.
- NamedConcreteElement - A 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 allows for a detailed filtering of the parsed elements 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 UnsolvedReference object. The second step is to obtain all the UnsolvedReference objects and iterate them in order to obtain the value of each element ref attribute and 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. Below it is presented a concrete example of how this process works in order to provide a better understanding of this process.

```
<?xml version='1.0' encoding='utf-8' ?>
<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>
```

Listagem 3.3: 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 user can query the parser to discover which elements are missing and where were they used. The user 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 in order to obtain the information needed to generate the correspondent classes. In order to generate classes this library also uses the ASM[2] library, which is a library that provides a Java interface to bytecode manipulation, which allows for the creation of classes, methods, etc. This was the suggested library to implement the code generation part of the project. It is a library that is still regularly maintained, the most recent version, 6.1, was release in 11 march of 2018. Also it has some tools to help the new users to understand how the library works, which are very helpful in order to start generating code and to increase the complexity of the generated code.

3.2.1 Supporting Infrastructure

Based on the XSD language it was created a infrastructure that served as a common set of classes that will belong to every API generated by this project. 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 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.

Visitor classes:

[2] `http://asm.ow2.org/`

- ElementVisitor - An interface to define methods for all the generated elements that can be visited with the Visitor pattern.

- AbstractElementVisitor - An abstract class that implements all the methods from ElementVisitor, pointing every method implementation for a single method. It helps the end user to create a custom Visitor with less code in situations where only few elements have different implementations.

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

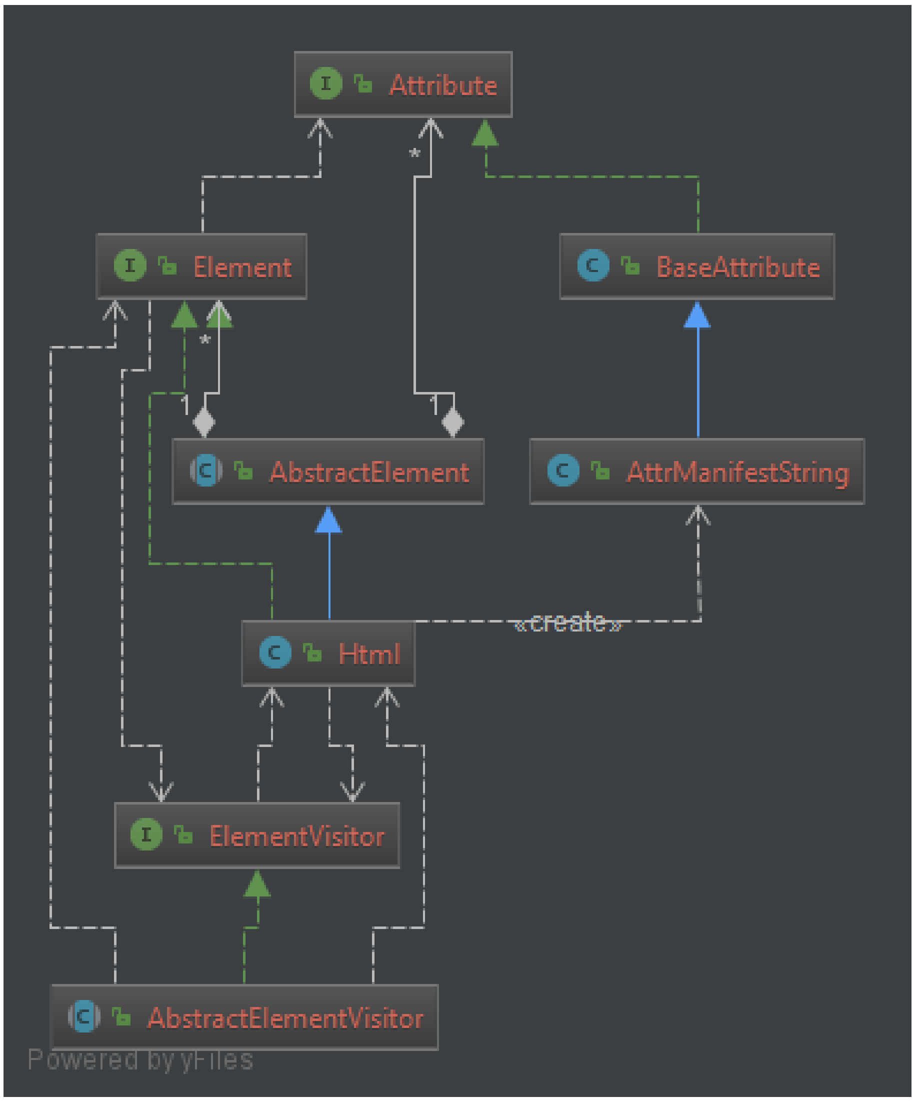


Figura 3.1: API - Supporting Infrastructure

3.2.2 Code Generation Strategy

In order to understand how most of this project works an example is provided and an detailed explanation will be provided. In this example there will be some simplifications in order to make it easier to understand.

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

   <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>
```

Listagem 3.4: Code Generation XSD Example

With this example there are a multitude of classes that will need to be created, apart from the always present supporting infrastructure as presented above.

- Html Element - A class that represents the Html element, deriving from AbstractElement.
- Body and Head Methods - Both methods present in the Html class that add Body and Head instances to Html children.
- Manifest Method - A method present in Html class that adds an instance of the Manifest attribute to the Html attribute list.

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

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

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

```java

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

Listagem 3.5: Html Element Class

- Body and Head classes - Classes for both Body and Head elements, similar to the generated Html class. The class contents will be dependent on the contents present in the concrete xsd:elements.

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

Listagem 3.6: Body Element Class

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

Listagem 3.7: Head Element Class

- Manifest Attribute - A class that represents the Manifest attribute, deriving from BaseAttribute.

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

Listagem 3.8: Manifest Attribute Class

- CommonAttributeGroup Interface - An interface with default methods that add the group attributes to the concrete element.

```java
public interface CommonAttributeGroup extends Element {

  default Html attrSomeAttribute(String attributeValue) {
    this.addAttr(new SomeAttribute(attributeValue));
    return this;
  }
}
```

Listagem 3.9: CommonAttributeGroup Interface

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. Reflecting those same restrictions to the Java language we have two ways of ensure those same restrictions, either at runtime or in compile time. This library tries to validate most of the restrictions in compile time, as shown in the example above. But in some restrictions it isn't possible to validate in compile time, an example of this is the following restriction:

```xml
<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>
```

Listagem 3.10: Restrictions Example

In this example we have an element that has an attribute called valueList. This attribute has some restrictions, it is represented by a xs:list and its element count should be between 1 and 5. Transporting this example to the Java language it will result in the following class:

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

Listagem 3.11: Attribute Class Example

But with this solution the xs:maxLength and xs:minLength are ignored. To solve this problem the existing restrictions existing in any given attribute are hardcoded in the class static constructor, which stores the restrictions in a Map object. This way, whenever an instance is created a validation function is called in the BaseAttribute constructor and will throw an exception if any restriction present in the Map is violated. This way the generated API ensures that any successful usage follows the rules previously defined.

3.2.3.1 Enumerations

In regard to the restrictions there is a special restriction that can be enforced at compile time, the xs:enumeration. In order 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 we have an attribute with three possible values, command, checkbox and radio.

```xml
<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>
```

Listagem 3.12: Enumeration Definition

This results in the creation of an Enum, EnumTypeCommand, as shown and the attribute will then receive an instance of EnumTypeCommand, ensuring only allowed values are used.

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

Listagem 3.13: Enumeration Class

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

Listagem 3.14: Attribute Receiving An Enumeration

3.2.4 Element Binding

In order to support repetitive tasks over an element binders were implemented. This allows for users to define, for example, templates for a given element. An example is presented below, it uses the HTML5 API as example.

```java
public class BinderExample{
   public void bindExample(){
      Html<Html> root = new Html<>();
      Body<Html<Html>> body = root.body();

      Table<Body<Html<Html>>> table = body.table();
      table.tr().th().text("Title");
      table.<List<String>>binder((elem, list) ->
                  list.forEach(tdValue ->
                     elem.tr().td().text(tdValue)
                  )
               );

      //Keep adding elements to the body of the document.
   }
}
```

Listagem 3.15: Attribute Receiving An Enumeration

In this example a table is created, and a title is added in the first row as a title header. 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 way a template can be defined and reused with different values. A full example of how this works is available at the method testBinderUsage.

3.3 Client

Some stuff here.

3.4 HtmlFlow

More stuff here.

4 Deployment

4.1 Github Organization

This project and all its components belong to a github organization called xmlet. 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, managing project dependencies. Maven was also used to generate documentation and deploying the projects to a central code repository, 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 present in this dissertation were evaluated in various metrics and the results made public for consultation. This way, either future users of those projects or just for the developers, this metrics can be used to serve 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.

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.

5 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.

Referências

[1] Per. Christensson. Markup language definition., June 2011. URL https://techterms.com/definition/markup_language. (p. 1)