



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

Relatório preliminar para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador : Doutor Fernando Miguel Gamboa de Carvalho

Janeiro, 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

Relatório preliminar para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador : Doutor Fernando Miguel Gamboa de Carvalho

Janeiro, 2018



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 API based on a XML schema. 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]. This markup languages add annotations to text, 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 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. In the following HTML example there is a violation of HTML rules, a `<html>` tag containing a `<div>` tag, which isn't allowed.

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

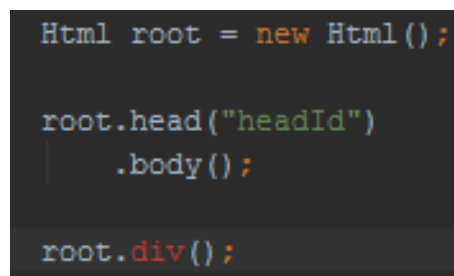
Listing 1.1: Failed HTML rule validation

The solution to having documents that respect the markup language rules is changing the way the document writing works. If the user has control over the writing process errors can be induced in the document but if the writing control is given to an entity which can enforce the markup languages restrictions then it is guaranteed that the user can't produce a document with errors. In the following code sample the user is allowed to add any child to the Html element, resulting in a violation of the language restrictions, which is only detected during execution.

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

Listing 1.2: Error validation in runtime

In order to create this entity all the restrictions of the given markup language could be recreated in a API and therefore allow the user to generate the document. In the 1.1, the API enforces the language restrictions in compile time, this way any document generated by this API will be valid.



```
Html root = new Html();

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

root.div();
```

Figure 1.1: Error validation in compile time

The only problem with this solution is recreating all the rules of a given markup language, which can be a very long process since most markup languages have a vast number of elements and restrictions.

The solution for this 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 that enforce those restrictions. 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 semantics. This is the main objective of this work, creating an infrastructure that reads a markup language definition, in XML Schema Definition, 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 XML schema. In this case we have 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 XML document. This approach is very similar to DOM, the difference being that in this case each API will be specific to a given XSD file, in this case the HTML5 specification. 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 Organization

This project will be divided in separate components, in order to keep a logic organization. The different components are explained in the following subsections. The interaction between the existing components is described in 1.2.

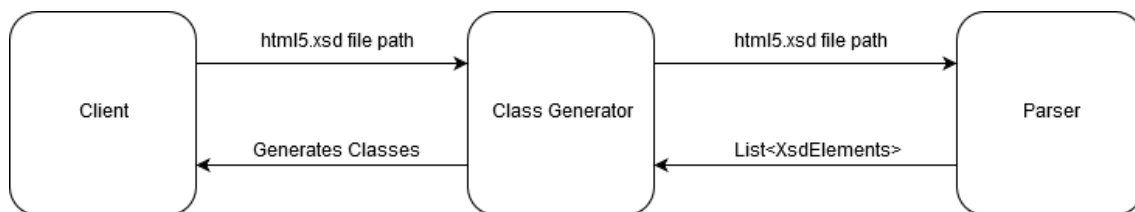


Figure 1.2: Components interaction

1.3.1 Parser

The parser will have the responsibility to read the XML schema element tree and extract the needed information in order to generate the respective classes. The result of the execution of this component should be a list of elements, each element representing a class that should be created and containing all the information needed for the creation.

1.3.2 Class Generator

The class generator will have the responsibility of generating classes based on the information received from the Parser. The class generator should request the parsing of a XML schema file and based on the Parser result, create the classes accordingly. Apart from that the generator should also create an infrastructure that will help the usage of the the resulting API.

To achieve the generation of the classes a tool named ASM will be used. This tool allows the manipulation of byte codes, allowing the generation of classes, methods and fields.

1.3.2.1 Supporting Infrastructure

The generated code will be supported by an infrastructure that mimics the syntax of XML schema files. The supporting infrastructure is shown in 1.3.

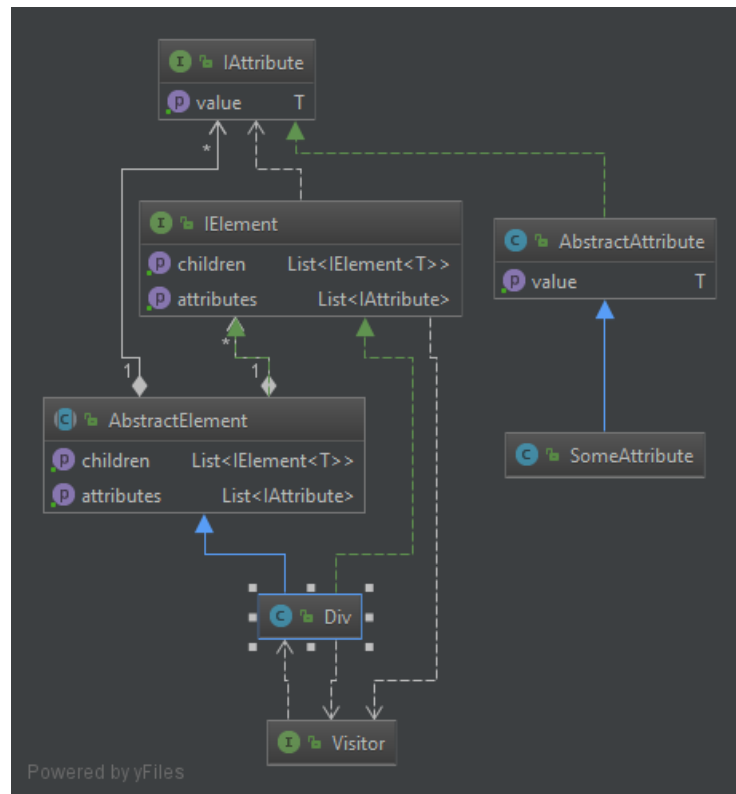


Figure 1.3: Supporting Infrastructure

All the generated APIs will have this classes which are independent of the contents of the parsed file. This infrastructure will then be extended by different type of classes, divided in four groups.

Elements

The elements are a group of classes that will be generated based in the existing xsd:elements. All these classes will extend AbstractElement. Each element will also contain the specific element code, which can include addition of elements, attributes or implementing group and element interfaces.

Attributes

The attributes are a group of classes that will be generated based in the existing xsd:attributes. All these classes will extend AbstractAttribute. Each attribute will have a type, that indicates the type of the value of the said attribute. Attributes will also enforce restrictions to their value if there are any explicitly described in the XML schema file.

Group Interface

The group interfaces are an addition that represent the `xsd:attributegroups`. In the XML schema these attributegroups indicate that a given element is allowed to have the said attributes, in the generated code the respective interfaces allow the addition of all the attributes present in the said group to the element attributes.

Element Interface

The element interfaces are similar to the group interfaces, the difference being that element interfaces allow the addition of other elements as children of the current element.

Visitors

In order to the generated API allow manipulation by the client all the generated elements implement the Visitor pattern, therefore the client of the API can implement its own Visitor class and specify the behavior of the visit methods.

1.3.3 Client

A separate component that will execute the class generator specifying the file name and the API name to be generated. This component will then contain the resulting API and can be used directly. This component can also implement tests to verify if the generation of the API implemented all the expected features.

Bibliography

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