

## 1. ATL Transformation Example

### 1.1. Example: JavaSource → Table

The JavaSource to Table example describes a transformation from a Java source code to a table which summarizes how many times each method declared in the Java source code is called within the definition of any declared method. A XML-based format, inspired from the JavaML [1], has been developed to encode all interesting information available in the Java source code. We use, for this transformation, a very basic abstract Table model which may be easily mapped to existing table representations (XHTML, ExcelML, etc.).

#### 1.1.1. Transformation overview

The aim of this transformation is to generate, from a Java source code (that is, the declaration of several Java classes), a Table document which summarizes how many times each method is called within the definition of the declared methods. The generated table is organized as follows:

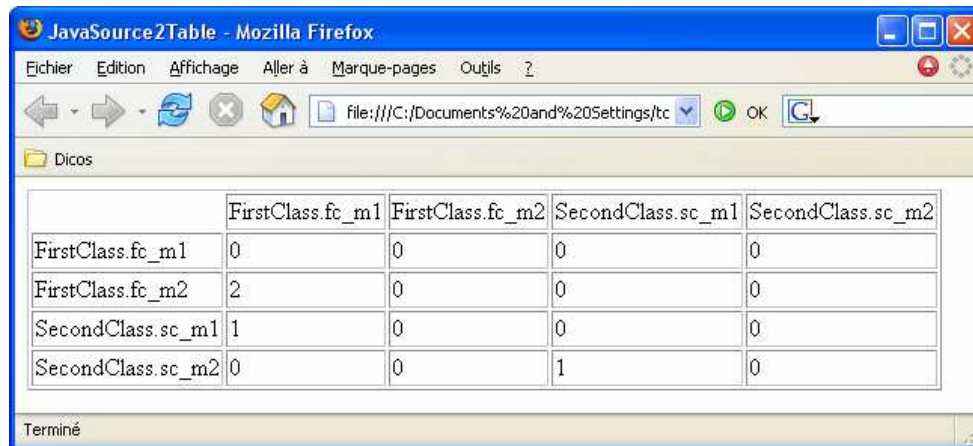
- Except the first cell, which is empty, the first row contains the list of the methods declared in the input Java source code, sorted according to the “class\_name.method\_name” string, where “method\_name” is the name of a method, and “class\_name” the name of the class in which the method is defined.
- The first column is organized in the same way as the first row is.
- Cells of following rows contain the number of calls of the in-column method within the in-row method declaration.

FirstClass.java	SecondClass.java
<pre>public class FirstClass {     public void fc_m1(){     }     public void fc_m2(){         this.fc_m1();         this.fc_m1();     } }</pre>	<pre>public class SecondClass {     public void sc_m1(){         FirstClass a = new FirstClass();         a.fc_m1();     }     public void sc_m2(){         this.sc_m1();     } }</pre>

**Table 1. Java source example**

As an example of the JavaSource to Table transformation, Figure 1 provides the XHTML table corresponding to a Java source code composed of the two classes (“FirstClass” and “SecondClass”) defined in Table 1. All defined methods appear both in rows and columns, sorted by their class, and their name. The table has to be read as follows: the method “FirstClass.fc\_m2” (row 3) is called twice in the definition of “FirstClass.fc\_m1” (column 2), but not in the other method definitions.

Prior to the transformation, the Java source code is injected into an XML model that encodes the information required for the transformation. The developed injector is described in the next section.



	FirstClass.fc_m1	FirstClass.fc_m2	SecondClass.sc_m1	SecondClass.sc_m2
FirstClass.fc_m1	0	0	0	0
FirstClass.fc_m2	2	0	0	0
SecondClass.sc_m1	1	0	0	0
SecondClass.sc_m2	0	0	1	0

Terminé

**Figure 1. Method invocations summary for FirstClass and SecondClass**

### 1.1.2. From Java source code to the JavaSource XML

An injector has been developed to inject Java source code into an XML-based JavaSource model. This model is designed so that it only includes information that is relevant for the JavaSource to Table transformation: the name of declared classes, the name of defined methods (for each class declaration), the name and the class of invoked methods (for each defined method). A simple DTD for this XML model is provided in Appendix III.

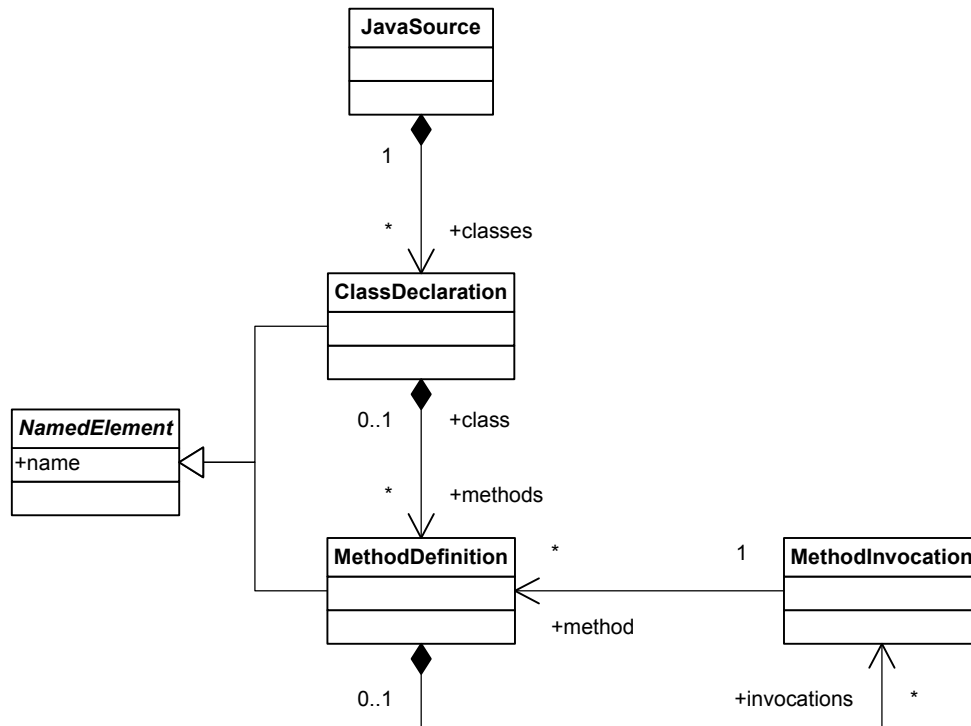
Each Java class is defined in a distinct file. The injector accepts several “.java” files as input, and produces a single file that contains the XML-based representation of the Java source code in input. Table 2 describes the XML file generated from the FirstClass and SecondClass classes described in Table 1.

<pre> &lt;javasource&gt;    &lt;class-declaration&gt;     &lt;name&gt;FirstClass&lt;/name&gt;     &lt;method-definition&gt;       &lt;name&gt;fc_m1&lt;/name&gt;     &lt;/method-definition&gt;     &lt;method-definition&gt;       &lt;name&gt;fc_m2&lt;/name&gt;       &lt;method-invocation&gt;         &lt;class&gt;FirstClass&lt;/class&gt;         &lt;name&gt;fc_m1&lt;/name&gt;       &lt;/method-invocation&gt;       &lt;method-invocation&gt;         &lt;class&gt;FirstClass&lt;/class&gt;         &lt;name&gt;fc_m1&lt;/name&gt;       &lt;/method-invocation&gt;     &lt;/method-definition&gt;   &lt;/class-declaration&gt; </pre>	<pre>     &lt;class-declaration&gt;       &lt;name&gt;SecondClass&lt;/name&gt;       &lt;method-definition&gt;         &lt;name&gt;sc_m1&lt;/name&gt;         &lt;method-invocation&gt;           &lt;class&gt;FirstClass&lt;/class&gt;           &lt;name&gt;fc_m1&lt;/name&gt;         &lt;/method-invocation&gt;       &lt;/method-definition&gt;       &lt;method-definition&gt;         &lt;name&gt;sc_m2&lt;/name&gt;         &lt;method-invocation&gt;           &lt;class&gt;SecondClass&lt;/class&gt;           &lt;name&gt;sc_m1&lt;/name&gt;         &lt;/method-invocation&gt;       &lt;/method-definition&gt;     &lt;/class-declaration&gt;   &lt;/javasource&gt; </pre>
---	---

**Table 2. JavaSource XML representation**

## 1.2. Metamodels

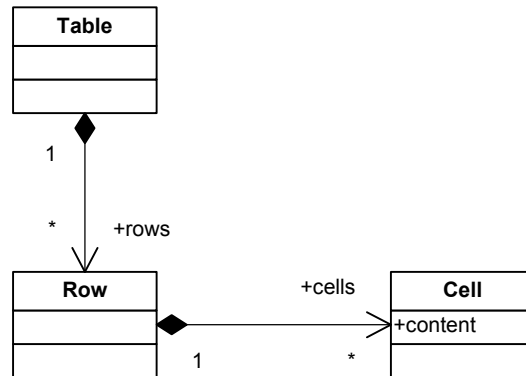
This transformation is based on a basic JavaSource metamodel which only deals with information that is relevant in the scope of this transformation. The considered metamodel is presented in Figure 2, and provided, in km3 format [2], in Appendix I.



**Figure 2. The JavaSource metamodel**

Java sources are modeled by a JavaSource element. This element is composed of ClassDeclarations. Each ClassDeclaration is composed of MethodDefinitions. Both ClassDeclaration and MethodDefinition inherit from the abstract NamedElement class (which provides a name). A MethodDefinition is composed of MethodInvocations (a call to a method). Each MethodInvocation is, in its turn, associated with a one and only MethodDeclaration (the called method).

The transformation also relies on an abstract Table definition. The metamodel considered here is described in Figure 3, and provided in Appendix II in km3 format.



**Figure 3. The Table metamodel**

Within this metamodel, a Table is associated with a Table element. Such an element is composed of several Rows that, in their turn, are composed of several Cells.

### 1.3. Rules Specification

These are the rules to transform a JavaSource model to a Table model:

- For the root JavaSource element, the following elements are created:
  - A Table element which is composed of a sequence of rows;
  - A Row element, linked to the Table element, which corresponds to the first row of the Table. This element is composed of the following sequence of elements
    - An empty Cell, linked to the Row element, which is the first cell of the first row.
    - One Cell, linked to the Row element, for each MethodDefinition. The content of the Cell is equal to the “class\_name.method\_name” string. Within the sequence, Cells are ordered according to their content.
- For each MethodDefinition, the following elements are created:
  - A Row linked to the Table element. This element is composed of the following sequence of elements:
    - A title Cell, linked to the current Row element. Its content is equal to the “class\_name.method\_name” string, where “class\_name” is the name of the class associated with the current MethodDefinition, and “method\_name” is the name of the current MethodDefinition.
    - One Cell, linked to the current Row element, for each MethodDefinition. The content of this Cell corresponds to the number of calls of the in-column method within the definition of the current in-row method.

## 1.4. ATL Code

This ATL code for the JavaSource to Table transformation consists of 2 helpers and 2 rules. Among helpers, allMethodDefs computes the set of all MethodDefinitions, ordered according to: 1) their class name, and 2) their method name.

The computeContent helper returns the number of calls of an in-column MethodDefinition (provided as a parameter) within the current MethodDefinition. For this purpose, it selects, among MethodInvocations within the definition, those that have the same class and method names that the MethodDefinition parameter. The rule then returns the size of the built set.

The rule Main allocates the structure of the Table model: the Table element ("t") and the first Row of this table ("first\_row"). This row is composed of a first empty cell ("first\_col"), and the cells associated with the declared methods ("title\_cols"). This rule makes use of the "thisModule.resolveTemp(e, str)" method (line 50). This method makes it possible to handle the output elements generated in the 'row' output of rule MethodDefinition. The use of the "distinct ... foreach(...)" command (line 62) makes it possible, within a single rule, to generate a distinct Cell for each element of the allMethodDefs sequence.


The rule MethodDefinition allocates a new Row for each declared MethodDefinition. The rule creates a Row element ("row") which is composed of a first Cell element ("title\_cell") and the data Cells associated with each declared method ("cells"). The content of the sequence of data cells is computed by calling the computeContent helper for each element of the allMethodDefs ordered set.

```
1  module JavaSource2Table;
2  create OUT : Table from IN : JavaSource;
3
4
5  -----
6  -- HELPERS -----
7  -----
8
9  -- This helper builds the sequence of all method definitions in all existing
10 -- classes.
11 -- Built sequence is ordered according to the couple (class_name, method_name).
12 -- RETURN: Sequence(JavaSource!MethodDefinition)
13 helper def: allMethodDefs : Sequence(JavaSource!MethodDefinition) =
14   JavaSource!MethodDefinition.allInstances()
15   ->sortedBy(e | e.class.name + '_' + e.name);
16
17 -- This helper builds the content of the table cell associated with the context
18 -- MethodDefinition (row) and the input MethodDefinition (column).
19 -- The computed value corresponds to the number of calls of the second method
20 -- within the first method definition.
21 -- CONTEXT: JavaSource!MethodDefinition
22 -- IN:      JavaSource!MethodDefinition
23 -- RETURN: String
24 helper context JavaSource!MethodDefinition
25   def : computeContent(col : JavaSource!MethodDefinition) : String =
26     self.invocations
27     ->select(i | i.method.name = col.name and
28             i.method.class.name = col.class.name)
29     ->size();
30
31
32 -----
33 -- RULES -----
34 -----
35
```

```
36 -- Rule 'Main'
37 -- This rule generates the Table as well as its first row.
38 -- First row cells contain the different defined method, in the format
39 -- 'class_name.method_name', ordered by the value of the build string.
40 rule Main {
41   from
42     s : JavaSource!JavaSource
43
44   to
45     -- Table is composed of the first row + data rows
46     t : Table!Table (
47       rows <-
48         Sequence{first_row,
49                 thisModule.allMethodDefs
50                 ->collect(e | thisModule.resolveTemp(e, 'row'))
51               }
52     ),
53     -- First row is composed of the first column + title columns
54     first_row : Table!Row (
55       cells <- Sequence{first_col, title_cols}
56     ),
57     -- First column empty
58     first_col : Table!Cell (
59       content <- ''
60     ),
61     -- Title cols = 'class_name.method_name'
62     title_cols : distinct Table!Cellforeach(mDef in thisModule.allMethodDefs)(
63       content <- mDef.class.name + '.' + mDef.name
64     )
65 }
66
67 -- Rule 'MethodDefinition'
68 -- This rule generates the content of the table, including the first cell of
69 -- each row, which identifies a method (format 'class_name.method_name').
70 rule MethodDefinition {
71   from
72     m : JavaSource!MethodDefinition
73
74   to
75     -- Rows are composed of the first (title) cell + data cells
76     row : Table!Row (
77       cells <- Sequence{title_cel, cels}
78     ),
79     -- Title cell = 'class_name.method_name'
80     title_cel : Table!Cell (
81       content <- m.class.name + '.' + m.name
82     ),
83     -- Data cells = nb of calls of each method within a method definition
84     cels : distinct Table!Cell foreach(mDef in thisModule.allMethodDefs)(
85       content <- m.computeContent(mDef).toString()
86     )
87 }
```

## I. JavaSource metamodel in km3 format


```
package JavaSource {  
    class JavaSource {  
        reference classes[1-*] container : ClassDeclaration;  
    }  
  
    abstract class NamedElement {  
        attribute name : String;  
    }  
  
    class ClassDeclaration extends NamedElement {  
        reference superclass[0-1] : ClassDeclaration;  
        reference methods[*] container : MethodDefinition oppositeOf "class";  
    }  
  
    class MethodDefinition extends NamedElement {  
        reference "class"[1-1] : ClassDeclaration oppositeOf methods;  
        reference invocations[*] container : MethodInvocation;  
    }  
  
    class MethodInvocation {  
        reference method[1-1] : MethodDefinition;  
    }  
}  
  
package PrimitiveTypes {  
    datatype String;  
}
```

	<b>ATL TRANSFORMATION EXAMPLE</b>	
	<b>JavaSource to Table</b>	Date 11/03/2005

## II. Table metamodel in km3 format


```
package Table {  
    class Table {  
        reference rows[1-]* ordered container : Row;  
    }  
  
    class Row {  
        reference cells[1-]* ordered container : Cell;  
    }  
  
    class Cell {  
        attribute content : String;  
    }  
}  
  
package PrimitiveTypes {  
    datatype String;  
}
```



	<b>ATL TRANSFORMATION EXAMPLE</b>	
	<b>JavaSource to Table</b>	Date 11/03/2005

### III. XML-based Javasource DTD

```
<!DOCTYPE JavaSource [  
  
  <ELEMENT java-source (class-declaration+)>  
  <ELEMENT class-declaration (name, method-definition*)>  
  <ELEMENT method-definition (name, method-invocation*)>  
  <ELEMENT method-invocation (class, name)>  
  <ELEMENT name (#PCDATA)>  
  <ELEMENT class (#PCDATA)>  
  
>
```

	<b>ATL TRANSFORMATION EXAMPLE</b>	
	<b>JavaSource to Table</b>	Date 11/03/2005

---

## References

- [1] JavaML: A Markup Language for Java Source, Greg J. Baros, <http://www.cs.washington.edu/research/constraints/web/badros-javaml-www9/>.
- [2] KM3: Kernel MetaMetaModel. Available at <http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/doc/atl/index.html>.