Code Generation of Class Diagrams in AToMPM

Simon Van Mierlo

1. Introduction

Class diagrams[1] are used to model the design of a system visually, containing classes (which have methods and attributes) and their relationships (inheritance, association, …). Commercial tools such as Enterprise Architect[2] allow for the creation of class diagrams and the generation of code in a chosen programming language from these models. This document shows how support for modeling class diagrams and code generation from these models was added to AToMPM. Section 2 explains the ClassDiagram formalism as it was created in AToMPM. Section 3 does the same for the SourceTree formalism, which allows for the modeling of a directory structure. Section 4 explains how the code generation process works. This process consists of two parts: first a SourceTree model is generated from a ClassDiagram model, then code is generated from this SourceTree model. In Section 5, lastly, open issues are discussed.

2. The ClassDiagram Formalism

Firstly, a formalism which allows for the creation of class diagrams has to be created in AToMPM. The metamodel of the ClassDiagram formalism is shown in Figure 1.

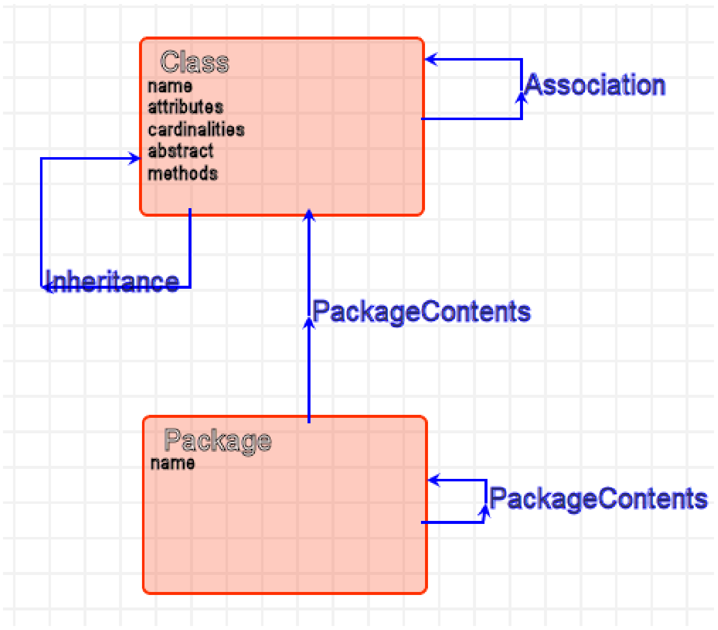


Figure 1: ClassDiagram metamodel.

[1] http://www.agilemodeling.com/artifacts/classDiagram.htm
[2] http://www.sparxsystems.com.au/

The attributes of **Class** are shown below.

```
[
   {
      "name": "name",
      "type": "string",
      "default": "Class_"
   },
   {
      "name": "attributes",
      "type": "list<$ATTRIBUTE>",
      "default": []
   },
   {
      "name": "cardinalities",
      "type": "list<$CARDINALITY>",
      "default": []
   },
   {
      "name": "abstract",
      "type": "boolean",
      "default": false
   },
   {
      "name": "methods",
      "type": "list<$METHOD>",
      "default": []
   }
]
```

Where the type **$METHOD** is defined as follows in **types.js**:

```
'$ARG':'map<[name,type],[string,string]>',

'$ARGS':'list<$ARG>',

'$METHOD':'map<[name,returnType,args,body],[string,string,$ARGS,code]>'
```

Currently, there is support for **Inheritance** and **Association** relationships between classes. A **Package** has a name and can contain zero or more classes and/or packages.

This allows for the creation of class diagram models. An example is shown in Figure 2.

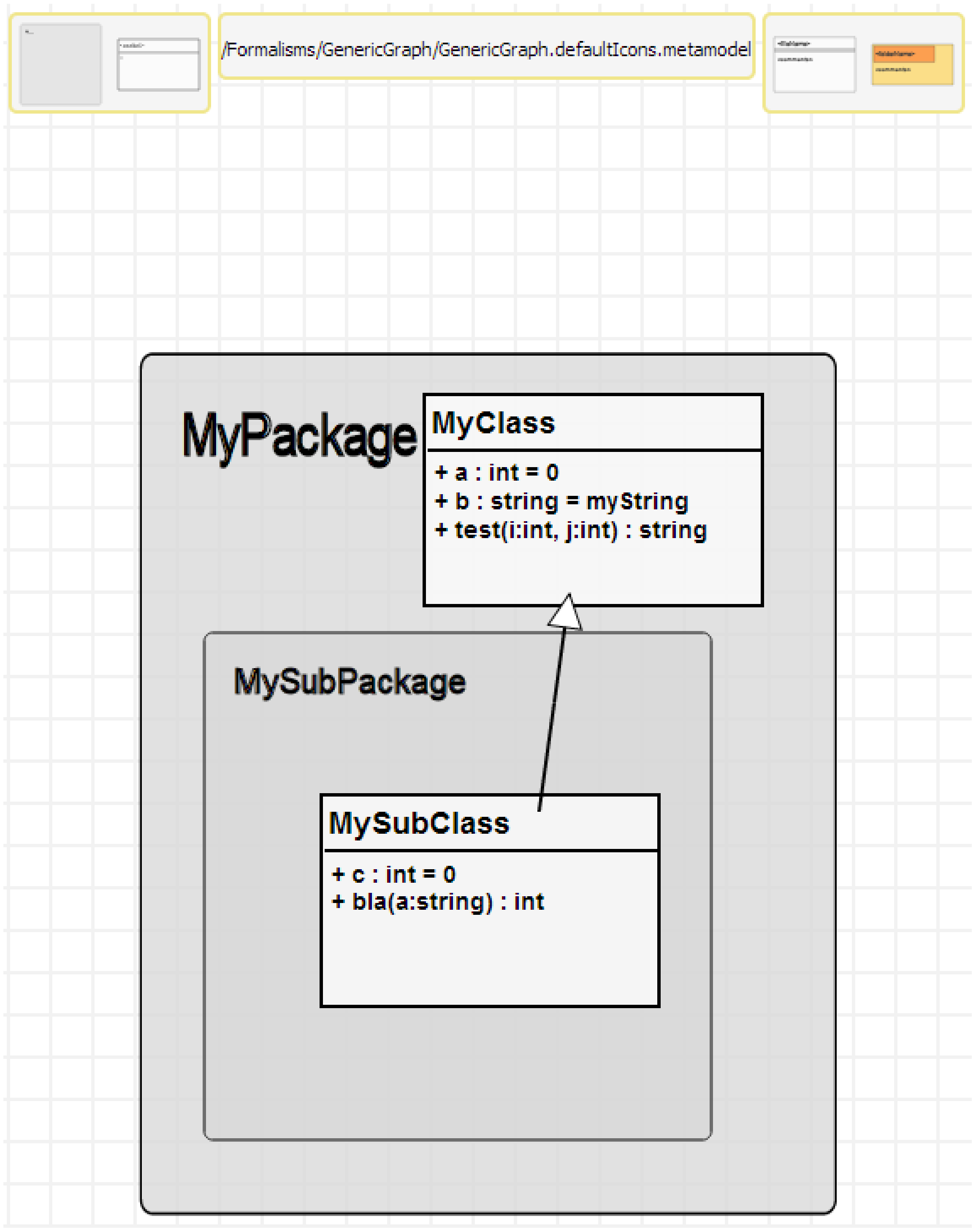


Figure 2: An example of a ClassDiagram model.

It would be possible to create a code generator which generates code in a certain programming language starting from this class diagram. However, this would mean that a generator has to be created for every programming language we might want to generate code for, as each language has its own specific syntax for specifying inheritance relationships, method declarations, etc. For this reason, first an intermediary model will be generated from the class diagram. All language-specific behavior will then be encoded in the transformation which generates this model. This intermediary formalism will be explained in the next section.

3. The SourceTree Formalism

The SourceTree formalism allows for the creation of models which represent a directory structure. Directories contain files and other directories, while files have content. The metamodel of this formalism is shown in Figure 3.

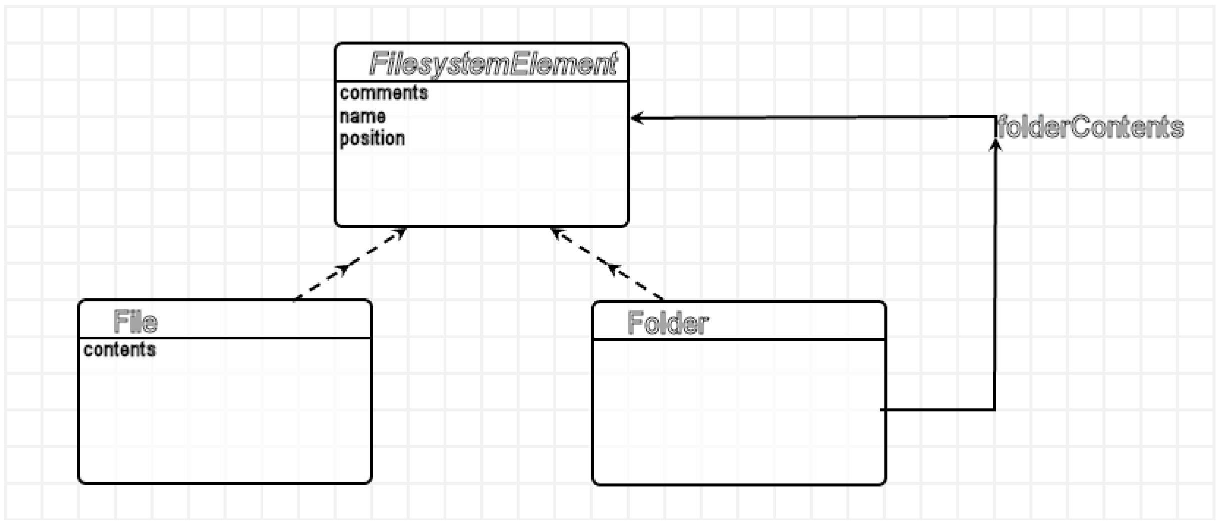
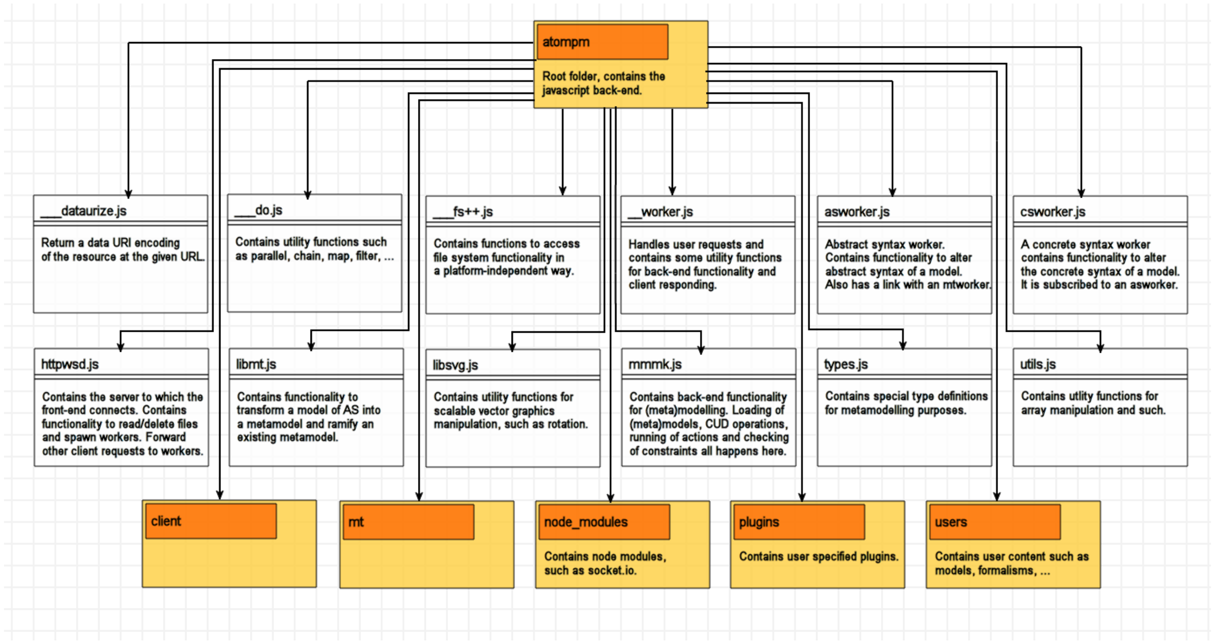


Figure 3: SourceTree metamodel.

An example of such a model, modelling the root AToMPM source folder, is shown in Figure 4.



atompm
Root folder, contains the javascript back-end.
___dataurize.js
Return a data URI encoding of the resource at the given URL.
___do.js
Contains utility functions such as parallel, chain, map, filter, ...
___fs++.js
Contains functions to access file system functionality in a platform-independent way.
__worker.js
Handles user requests and contains some utility functions for back-end functionality and client responding.
asworker.js
Abstract syntax worker. Contains functionality to alter abstract syntax of a model. Also has a link with an mtworker.
csworker.js
A concrete syntax worker contains functionality to alter the concrete syntax of a model. It is subscribed to an asworker.
httpwsd.js
Contains the server to which the front-end connects. Contains functionality to read/delete files and spawn workers. Forward other client requests to workers.
libmt.js
Contains functionality to transform a model of AS into a metamodel and ramify an existing metamodel.
libsvg.js
Contains utility functions for scalable vector graphics manipulation, such as rotation.
mmmk.js
Contains back-end functionality for (meta)modelling. Loading of (meta)models, CUD operations, running of actions and checking of constraints all happens here.
types.js
Contains special type definitions for metamodelling purposes.
utils.js
Contains utlity functions for array manipulation and such.
client
mt
node_modules
Contains node modules, such as socket.io.
plugins
Contains user specified plugins.
users
Contains user content such as models, formalisms, ...

Figure 4: Model of the root AToMPM source folder.

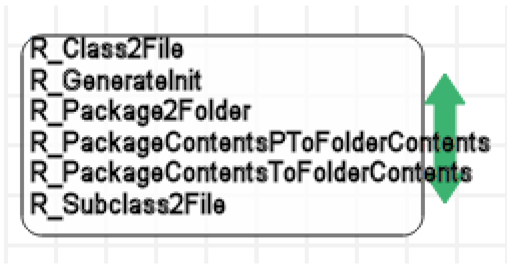
The next section explains how the ClassDiagram and SourceTree formalisms are used in tandem to generate source code from a ClassDiagram model.

4. Code Generation

This section will explain how, starting from a ClassDiagram model, source code is generated. The language of choice is Python, but transformations can be developed for every (object-oriented) programming language. First, the ClassDiagram model is transformed to a SourceTree model, as explained in Section 4.1. Then, the code is generated by an AToMPM plugin.

4.1. Transforming ClassDiagram Models to SourceTree Models

To transform ClassDiagram models to SourceTree models, the transformation shown in Figure 5 was created.



R_Class2File
R_GenerateInit
R_Package2Folder
R_PackageContentsPToFolderContents
R_PackageContentsToFolderContents
R_Subclass2File

Figure 5: CDtoST transformation.

The transformation generates a SourceTree structure containing Python files. It generates a file for each class and a folder for each package. It supports subclassing across packages, but does not do anything with association relationships.

4.2. Generating Code from FileSystem Models

AToMPM allows for the creation of plug-ins, which intercept HTTP requests and perform an action in response. A code generator plug-in was created, intercepting POST HTTP requests to the **/generatecode** URL. It allows for the specification of a ‘root’ directory, in which all generated code will be placed. The root will itself be a subfolder of the ‘generated_code’ folder, which is located in the root AToMPM folder. A toolbar was created with one button, performing the following action:

```
_openDialog(
    _CUSTOM,
    {'title':'Insert the name of the root folder.',
     'widgets':[{'id':'folder_name',
                 'type':'input',
                 'label':'Folder Name',
                 'default':''}]
    },
    function(data) {
        _httpReq('POST',
                 '/plugins/codegenerator/generatecode?wid='+_context.wid,
                 {'root':data['folder_name']});
    }
);
```

5. Issues

Currently, the body of a method has to be specified as a string. This means that each line has to be ended explicitly by a ‘\’, and each tab has to be encoded by a ‘\t’. During the transformation, each line is stripped of leading whitespaces and tabs, because this would otherwise lead to syntax errors in the Python files.