

# Design of an UML tool with round-trip engineering for Java and implementation as a web application in Adobe Air

Christoph Grundmann

10.08.2011

## Contents

<b>1</b>	<b>Round-trip engineering</b>	<b>1</b>
1.1	Reverse engineering . . . . .	2
1.1.1	Parser . . . . .	2
1.1.2	Parser grammar rules . . . . .	2
1.1.3	Parser grammar example . . . . .	3
1.1.4	Parser output . . . . .	4
1.1.5	Class diagram generation . . . . .	7
1.2	Forward engineering . . . . .	8
1.2.1	Generalization . . . . .	8
1.2.2	Binary association . . . . .	9
1.2.3	Provided interfaces . . . . .	10
1.2.4	Required interfaces . . . . .	11
1.2.5	Extended interfaces . . . . .	12

## 1 Round-trip engineering

Today's complexity of software requires the use of diagrams that will describe the structure of programs and how the internal processes communicate, during and primary at the beginning of the development. Moreover for object oriented languages like Java, the UML standard is the way to go, because of its focus for the object oriented developing of software. Structure diagrams, like the class diagram, have the capability to generate source code by analyzing it. In addition it's possible to create a class diagram with the

help of source code. The creation of Java source code from a class diagram is called forward engineering, whereas the creation of a class diagram from the Java source code is called reverse engineering. The whole technique is known as round trip engineering. So round trip engineering allows keeping the consistency between diagrams and source code.

## 1.1 Reverse engineering

To generate a class diagram with the help of Java source code, a parser is needed to interpret the grammatical structure of it. Usually a program doesn't understand the grammatical structure of an input like a string before it is interpreted. The first step in interpreting the source code a parser converts it into a specific data format, in this case into a JSON object, which can be used to interpret the grammatical structure. After that step the new output can be used to generate the class diagram. Classes can easily be instantiated by reading the JavaScript friendly output, where the relations need to be interpreted.

### 1.1.1 Parser

AlphaUML uses PEG.js, a Java parser that is completely written in JavaScript, to parse Java source code into a JSON object. It allows converting a string, by using a specific language grammar, into any output format. The grammar describes the Java language and defines the output. It follows specific rules similar to regular expressions to match the input with the Java grammar. Furthermore it allows manipulating the output of each expression by using JavaScript as a so called action. In this case an action is used to convert the expression result to a JSON object. But in some cases an empty string returns if an expression is optional and it doesn't match. Then the empty string will be replaced with null to catch missing matches in the processing to a class diagram.

### 1.1.2 Parser grammar rules

PEG.js provides some rules which are necessary to write an own grammar. A collection of the most common rules will be described below. In addition a small grammar example with explanation after the rules shows how they are used.

#### **"literal" or 'literal'**

Return the literal string on matching.

.

Return one character as a string.

#### **[characters]**

Return one character of the bracket contained set.

**rule**

It describes a fully expression that can use in other rules or expressions

**(expression)**

It allows the declaration of a subexpression to encapsulate it. Its similar to a rule and is necessary to set an expression optional.

**expression \***

Zero or more match results allowed. Returns the result as an array.

**expression +**

One or more match results necessary. If the input doesn't match it returns a parsing error.

**expression ?**

It allows optional expressions. On succeeding the match result returns, otherwise it returns an empty string.

**! expression**

It forbids a specific expression.

**label : expression**

A label stores the match result into a specific variable that has the same name. It can be used in an action to get access of the stored result.

**expression { action }**

An action is a JavaScript snippet that will be executed 'cause the match is successful. It has the access to each labeled expression of its expression and allows the manipulation of the match result.

**expression\_1 expression\_2 ... expression\_n**

A sequence of expression is allowed and returns their results as an array.

**expression\_1 / expression\_2 / ... / expression\_n**

It allows to create an OR function. It tries to match with one of the given expressions.

**1.1.3 Parser grammar example**

The following example shows a rule called DataType which is used for parsing a data type. It matches primitive data types, generic types or array types. Each information will return as a JSON object, formatted by the action. The DataTypeKeyword rule provides a collection of data types and allows each possible java identifier (Identifier) as

data type, so user-defined objects are possible. The Generic and Array rules are optional and return an empty string on miss.

Listing 1: list of data type keywords

```
1 DataTypeKeyword = (  
2     "boolean"  
3     /    "byte"  
4     /    "char"  
5     /    "double"  
6     /    "enum"  
7     /    "float"  
8     /    "int"  
9     /    "long"  
10    /    "short"  
11    /    "void"  
12    /    Identifier  
13 )
```

Listing 2: data type rule

```
1 DataType =  
2     $d:DataTypeKeyword --  
3     $g:($g:Generic -- {return $g;})?  
4     $a:($a:Array -- {return $a;})?  
5     {  
6         return {  
7             generic: $g != "" ? $g : null ,  
8             array: $a != "" ? $a : false ,  
9             dataType: $d  
10        };  
11    }
```

#### 1.1.4 Parser output

The produced output from the source code will be used to generate the class diagram. The JSON format of the output makes the next process easier, so it represents a specific logical structure and allows the immediate access to all necessary information about the parsed Java program. The output provides, for example, the info about used libraries, the class package, all class methods or variables to generate the class diagram. In addition the JavaDoc is available and will be pinned as note at the specific class in the diagram. The following JSON sample gives an overview about the output format and shows each possible fields like the package or classes fields.

Listing 3: file output

```

1 {
2   "package": "packageName",
3   "imports": ["path"],
4   "classes": []
5 }

```

Listing 4: classes output

```

1 "classes": [
2   {
3     "type": "interface",
4     "javaDoc": null,
5     "visibility": "public",
6     "name": "className",
7     "extend": "superClass",
8     "implement": [
9       "interfaceName"
10    ],
11    "body": {
12      "variable": [],
13      "method": []
14    }
15  }
16 ]

```

Listing 5: variable output

```

1 "variable": [
2   {
3     "type": "variable",
4     "javaDoc": null,
5     "name": "varName",
6     "visibility": "public",
7     "modifier": [
8       "static",
9       "final"
10    ],
11    "array": false,
12    "generic": null,
13    "dataType": "int",
14    "value": "42"
15  }
16 ]

```

Listing 6: method output

```

1 "method": [
2   {
3     "type": "method",
4     "javaDoc": {},
5     "name": "methodName",
6     "visibility": "public",
7     "modifier": [],
8     "generic": null,
9     "array": true,
10    "dataType": "String",
11    "parameter": [
12      {
13        "type": "parameter",
14        "modifier": ["static"],
15        "generic": "T",
16        "array": false,
17        "dataType": "Object",
18        "name": "paramterName"
19      }
20    ],
21    "body": "methodBody"
22  }
23 ]

```

Listing 7: JavaDoc output

```

1 "javaDoc": {
2   "since": [
3     {
4       "tag": "since",
5       "description": "desc"
6     }
7   ],
8   "throws": [
9     {
10      "tag": "throws",
11      "classname": "className",
12      "description": "desc"
13    }
14  ],
15  "exception": [
16    {
17      "tag": "exception",

```

```

18         "classname": "className",
19         "description": "desc"
20     },
21 ],
22 "param": [
23     {
24         "tag": "param",
25         "name": "paramName",
26         "description": "desc"
27     }
28 ],
29 "return": [
30     {
31         "tag": "return",
32         "description": "desc"
33     }
34 ],
35 "see": [
36     {
37         "tag": "see",
38         "description": "desc"
39     }
40 ]
41 }

```

### 1.1.5 Class diagram generation

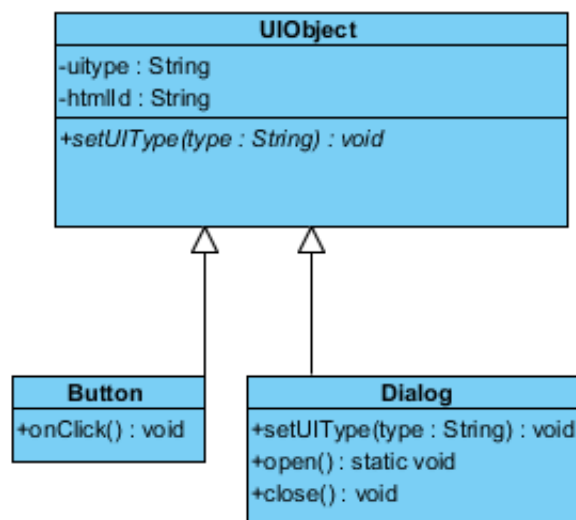
After parsing the Java source code the new data structure can be used to build a class diagram, which visualize the logical structure of its program. The process which builds the class diagram is subdivided into 3 parts. In the first step each class with all selected info - like the name, its variables and operations - will be created. If two classes with the same name exist, the name of the second class will be renamed. In this case each relation which has a reference on its class name could not be created anymore. During the first step any information which is needed for generalizations or the relations of provided interfaces will be stored. So it is not necessary to read the parser output twice. With the help of the stored information each generalization and all relations of provided interfaces can be created in step two. The last step needs the interpretation of each created class. To generate associations or the relations of required interfaces, each attribute of each class has to be analyzed. If an attribute has a data type of an existing class, it's a candidate for an association. In this case this class will be analyzed, too, to figure out whether the association has bidirectional navigation.

## 1.2 Forward engineering

To generate the source code from an existing class diagram, its essential to analyze each available class and connector, like generalizations or associations. The analyzing process will be handled from the classes themselves. So a class contains all necessary information and can handle it without an additional effort. Operations, parameter, attributes and classes own a method called *getJSON()* that creates an output which is identical to the parser output 1.1.4 and will be used to generate the source code. Furthermore it gets the relations between classes, so an additional process isnt necessary. On getting each declared attribute, associations of the two classes and each required interface will be collected and set as attribute. Moreover if an interface extends another interface it will set as superclass and declare with the extend keyword, in the other case if a class provides an interface the implements keyword will be used. The following sections explain each available relation and their implementation.

### 1.2.1 Generalization

If a class inherits from another class the subclass has to signalize the relation with the keyword extends in its declaration. In addition each operation and attribute which exists in the superclass doesnt need to be implemented into the subclass twice. If the user defines an operation which has the same name and parameter list of another one in the superclass, the superclass operation will be overwritten. To execute the method of the superclass Java contains the super keyword which is a reference on its superclass.



The class diagram shows the superclass UIObject and two subclasses which inherit from it. Moreover the class Dialog overrides the operation setUIType and implements some other methods.

Listing 8: The following code snippet shows the result of the code generation.



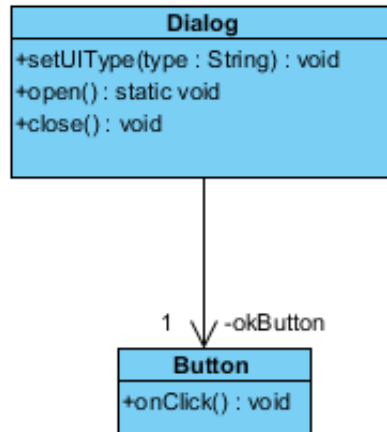
```

1 public class UIObject {
2     private String uitype;
3     private String htmlId;
4
5     public void setUIType(String type) {}
6 }
7
8 public class Button extends UIObject {
9     public void onClick() {}
10 }
11
12 public class Dialog extends UIObject {
13     public void setUIType(String type) {}
14
15     public void open() {}
16
17     public void close() {}
18 }

```

### 1.2.2 Binary association

A binary association specifies a semantic relationship between two classes. So each class knew about the other one and is able to interact or communicate with it. An association provides a collection of properties, which are important to interpret for the code generation. First of all the multiplicity of a class defines the frequency of occurrence. If a multiplicity is declared as [1..5] the class has to declare once and can be declared at most five times. In the source code a multiplicity represents an array if a class can declare more than one time. Otherwise it's a simple variable. Furthermore the role of a class gives the instance of itself a specific name. So if a role is set, the variable gets the name of it. In addition it's possible to set the visibility of a role. The last necessary association property is the navigation. An arrow visualizes the direction of the navigation and specifies in which class the other one will be accessible. The bidirectional navigation allows the navigation in both directions.



In this example the dialog implements a button which is called "okButton". In addition it's accessible in the dialog class only, 'cause of the "private" visibility. Moreover the multiplicity of 1 provides that only one button can instantiate.

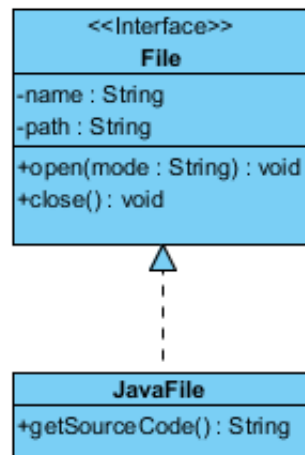
Listing 9: The following code snippet shows the result of the code generation.

```

1 public class Button {
2     public void onClick() {}
3 }
4
5 public class Dialog {
6     private Button okButton;
7
8     public void setUIType(String type) {}
9
10    public void open() {}
11
12    public void close() {}
13 }
  
```

### 1.2.3 Provided interfaces

Provided interfaces are like generalizations with abstract classes. Each operation will be inherited and has to overwrite in the subclass. The only two differences are the "implements" keyword instead of the extends keyword which is use for inheritance and the fact that it's possible to implement more than one interface. When using more than one interface, each interface will be separated by a comma, at its declaration. An interface provides abstract methods only, which order a specific realization of the class that inherits from it.



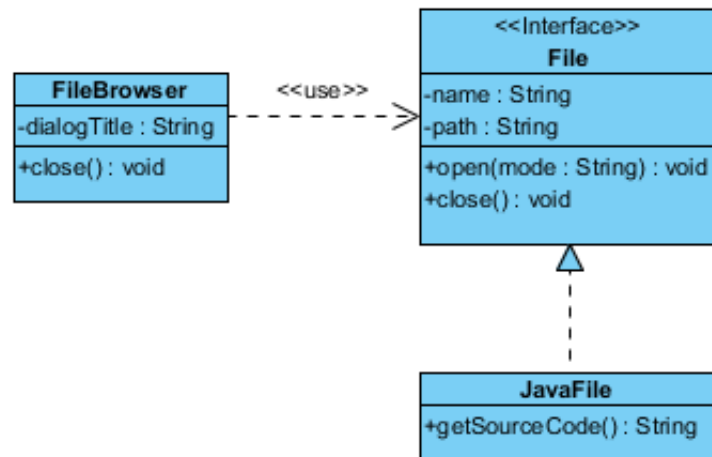
Listing 10: The following code snippet shows the result of the code generation.

```

1 public interface File {
2     private String name;
3     private String path;
4
5     public void open(String mode);
6
7     public void close ();
8 }
9
10 public class JavaFile implements File {
11     public String getSourceCode () {}
12
13     public void open(String mode);
14
15     public void close ();
16 }
  
```

#### 1.2.4 Required interfaces

A required interface allows the communication between a class and another class which implements an interface. If more than one class inherit from a specific interface it's possible to get access to each of them in the same way. This means that all classes which communicate through that interface are constrained of its operations. So it would throw exceptions if one of those operations is changed. A required interface will be declared as an attribute and can be initialized by any constructor of classes which implement the interface.



Listing 11: The following code snippet shows the result of the code generation.

```

1 public interface File {
2     private String name;
3     private String path;
4
5     public void open(String mode);
6
7     public void close();
8 }
9
10 public class FileBrowser {
11     private String dialogTitle;
12     private File _file;
13
14     public void close() {}
15 }
  
```

### 1.2.5 Extended interfaces

If an interface will be extended, a generalization 1.2.1 has to be used. That includes the UML generalization arrow and on the Java side, the use of the extends keyword.