

# **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>AlphaUML</b>	<b>4</b>
1.1	Introduction . . . . .	4
1.2	Configuration . . . . .	4
1.3	Projects . . . . .	4
1.3.1	New Project . . . . .	4
1.3.2	Load Existing Project . . . . .	4
1.4	Console . . . . .	4
1.4.1	Java Compiler . . . . .	5
1.4.2	JavaDoc Generator . . . . .	6
1.5	Code Editor . . . . .	6
1.6	Class Diagram Editor . . . . .	7
1.6.1	Class . . . . .	8
1.6.2	Note . . . . .	9
1.6.3	Binary Association . . . . .	9
1.6.4	Generalization . . . . .	10
1.6.5	Required Interface . . . . .	10
1.6.6	Provided Interface . . . . .	10
1.6.7	Round-trip Engineering . . . . .	10
<b>2</b>	<b>Implementation</b>	<b>11</b>
2.1	Frameworks . . . . .	11
2.1.1	Adobe Air . . . . .	11
2.1.2	Dojo Toolkit . . . . .	11
2.1.3	PEG.js . . . . .	12
2.1.4	Ace . . . . .	12
2.2	User Interface (UI) . . . . .	12
2.2.1	Graphical Design . . . . .	12
2.2.2	The basics of the GUI design . . . . .	13
2.2.3	Additional GUI components . . . . .	14
2.3	Single Page Web App . . . . .	14
2.4	Single Source Publishing . . . . .	15
2.4.1	Class Diagram Specifics . . . . .	15
2.4.2	Project Specifics . . . . .	16
<b>3</b>	<b>Round-trip engineering</b>	<b>17</b>
3.1	Reverse engineering . . . . .	17
3.1.1	Parser . . . . .	17
3.1.2	Parser grammar rules . . . . .	17
3.1.3	Parser grammar example . . . . .	19
3.1.4	Parser output . . . . .	19
3.1.5	Class diagram generation . . . . .	22

3.2	Forward engineering . . . . .	23
3.2.1	Generalization . . . . .	23
3.2.2	Binary association . . . . .	24
3.2.3	Provided interfaces . . . . .	25
3.2.4	Required interfaces . . . . .	26
3.2.5	Extended interfaces . . . . .	27

# 1 AlphaUML

## 1.1 Introduction

AlphaUML is a simple UML tool for creating class diagrams, which supports round-trip engineering. It is designed for small projects and has its focus on being a learning tool for students. So its important that changes can be realized easily by a user, which will enhance the learning by doing process. Moreover AlphaUML is currently specialized for Java, so the parser supports only Java as programming language and dialogs are structured according to the Java syntax. Furthermore it has a fully integrated editor, which is useful to edit existing source code.

## 1.2 Configuration

Some of the provided tools need to be configured. On the one hand the Java compiler 1.4.1 and on the other hand the JavaDoc generator 1.4.2 needs a specific file path, in which the executables are. In this case the "bin" directory of Java installation. Those paths can be set in the AlphaUML options. To open the options, navigate to the "File" menu and select the "Options" entry. Without the configuration, each of those programs throws an error and can't be executed.

## 1.3 Projects

### 1.3.1 New Project

### 1.3.2 Load Existing Project

## 1.4 Console

The console is a very useful tool to get information about the forward and reverse engineering process. It displays the current step of the executed process and catches errors. When a parsing error occurred the parser throws a message with all necessary information, like the Java file which causes the error and in which line and position it occurs. Furthermore it's able to execute the Java compiler 1.4.1 or the JavaDoc generator 1.4.2 by using their specific commands.

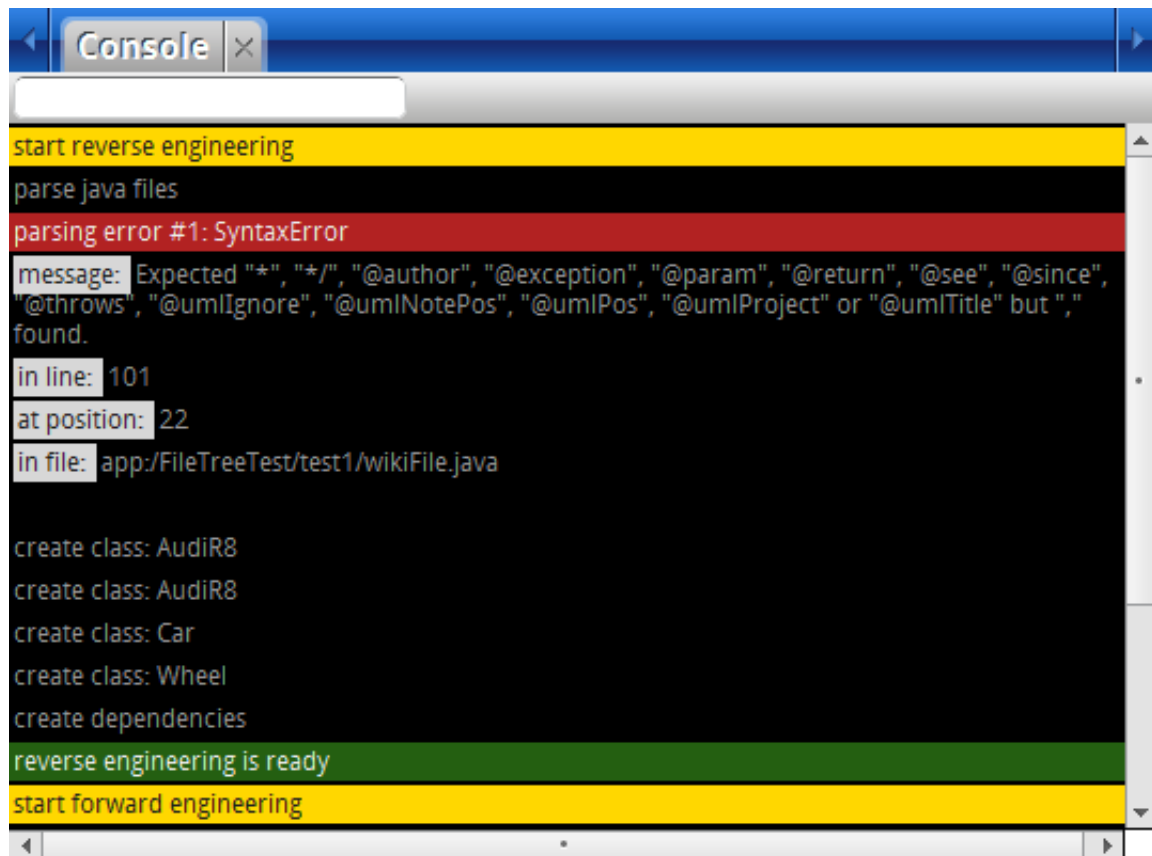


Figure 1: Console which showing the round-trip engineering process

#### 1.4.1 Java Compiler

To compile the current opened project, it's possible to execute the "javac" command via the command line of the console. Currently it saves the generated class file of an associated Java File in the "build" folder of the project, which will be found in the projects root. Moreover when the compiler throws an error, the console displays it. To be able to execute the "javac" command the console needs the path of the Java compiler, which has to be set in the AlphaUML options 1.2.

**Command**    javac



Figure 2: Execution of the Java compiler

### 1.4.2 JavaDoc Generator

It's also possible to generate documentation with the help of the JavaDoc generator. It collects each Java file of the project, which isn't ignored for those processes, and places the generated documentation into the "docs" folder of the projects root.

Like the "javac" command the "javadoc" command will be executed via the command line of the console and needs a correct configuration in the AlphaUML options 1.2.

**Command** javadoc

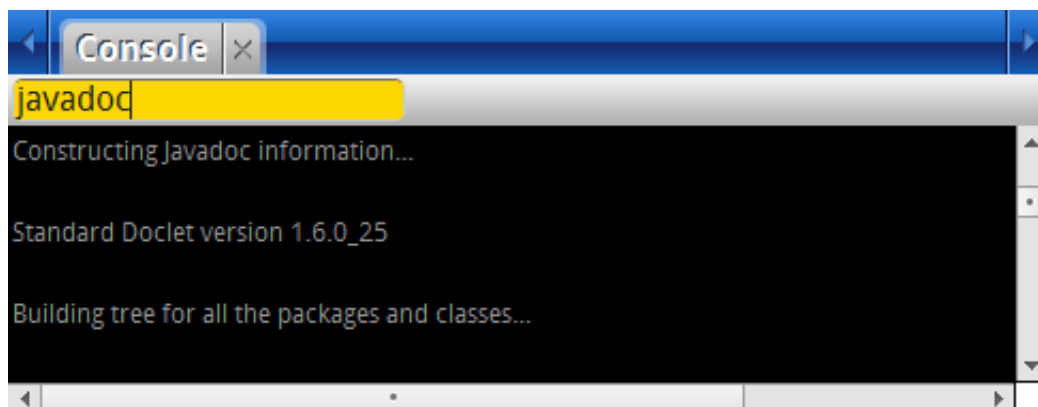
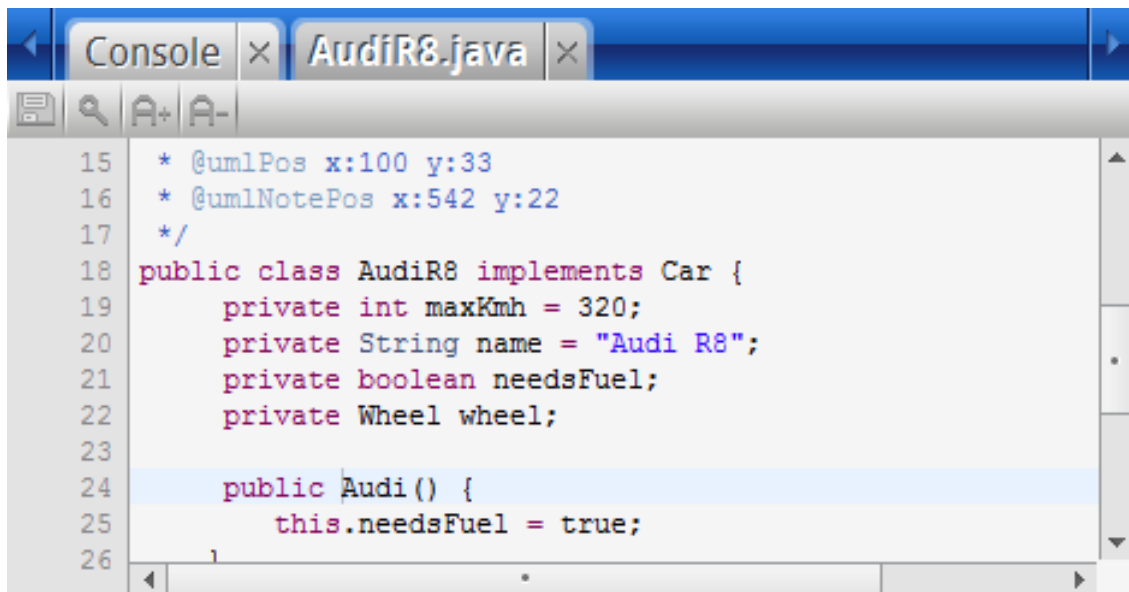


Figure 3: Execution of the JavaDoc generator

## 1.5 Code Editor

The code editor, as a main feature of AlphaUML, is an advanced editor, with various functionalities, like the searching and replacing of code snippets and the syntax highlighting. The editor is based on Ace 2.1.4 an open source editor written in JavaScript. Mainly Ace provides a huge collection of supported programming languages, like JavaScript, C/C++ or Java, but is optimized for editing JavaScript and CoffeeScript. So some

features aren't supported in AlphaUML, like the showing of syntax errors. But this feature is available thru the round-trip engineering and compiling process, which errors are shown in the console. The editor in AlphaUML is optimized for Java and text files and is able to save those files. If a new Java file will be created without a specific file path - for example after the code generation process - a file dialog will be opened, which can be used to save those. Another feature, - which is useful if a user believes that the used font is too small, especially 'caused an amblyopic - can increment or decrement the font size. The last possibility is to search and replace parts of source code or other keywords, which will be necessary if a document includes a huge number of characters, and the user will search something special.

The image shows a screenshot of the AlphaUML code editor. The window has a title bar with 'Console' and 'AudiR8.java'. Below the title bar is a toolbar with icons for file operations and font adjustments. The main area displays Java code for the 'AudiR8' class, which implements the 'Car' interface. The code includes private attributes for 'maxKmh', 'name', 'needsFuel', and 'wheel', along with a constructor 'Audi()' that initializes 'needsFuel' to true. Line numbers 15 through 26 are visible on the left side of the editor.

```
15  * @umlPos x:100 y:33
16  * @umlNotePos x:542 y:22
17  */
18  public class AudiR8 implements Car {
19      private int maxKmh = 320;
20      private String name = "Audi R8";
21      private boolean needsFuel;
22      private Wheel wheel;
23
24      public Audi() {
25          this.needsFuel = true;
26      }
```

Figure 4: View of the code editor

## 1.6 Class Diagram Editor

The class diagram editor is a small editor to draft class diagrams with a handful of tools to create classes, notes or for example generalizations. It simply consists of two components, the workspace and the toolbar, which gives access to each necessary tool. Moreover each tool is optimized for the Java programming language, to enhance Java developers. The editor supports the following UML notation elements, which will be described in the next chapters.

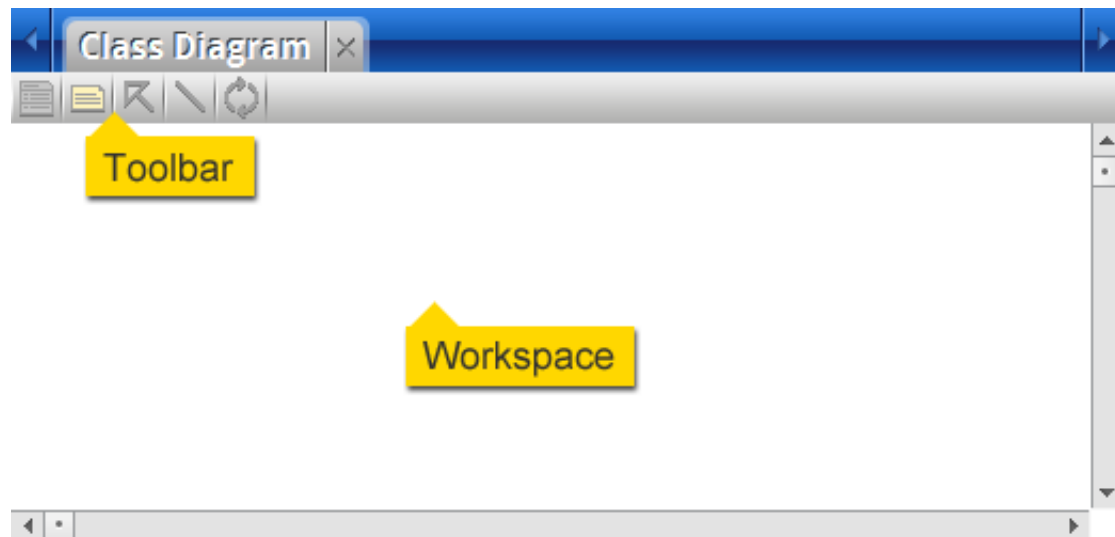


Figure 5: Layout of the class diagram editor

1. Class 1.6.1
2. Note 1.6.2
3. Provided Interface 1.6.6
4. Required Interface 1.6.5
5. Binary association 1.6.3
6. Generalization 1.6.4

### 1.6.1 Class

A class is a structural design for objects and contains attributes and operations, which describes its properties and functionalities. It is subdivided into 3 parts, a part for its name and stereotype or abstract declaration, another part for its attributes and at last a part for each operation of the class. To add a class the Add Class tool from the toolbar has to be selected, which allows placing a class into the workspace, by clicking on a specific place in it.



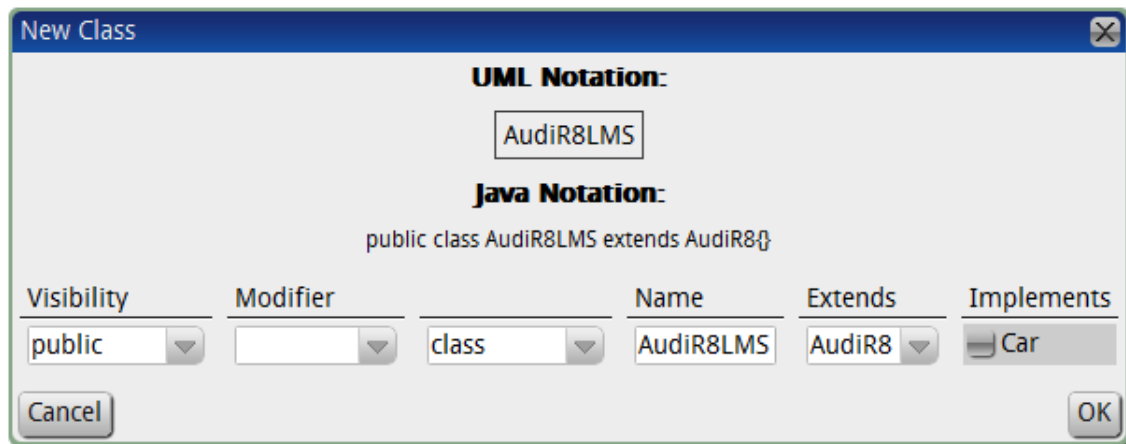


Figure 6: Dialog for adding a new class

### 1.6.2 Note

### 1.6.3 Binary Association

When a class refers to another class, a binary association is used to visualize a relation between these classes. An association allows declaring the direction of the navigation, which defines in which class a reference, will be used. Moreover it's possible to set the role name of its reference and the multiplicity to set the min and max number of instances. The association name is only a notation element for an easier readability of the association and won't be used as a property of an instance.

To create a binary association the "Add Association" tool has to be activated. Then the two classes which will be in relation have to be selected. And after these steps a dialog will be opened, which provides the configuration of all necessary association properties.

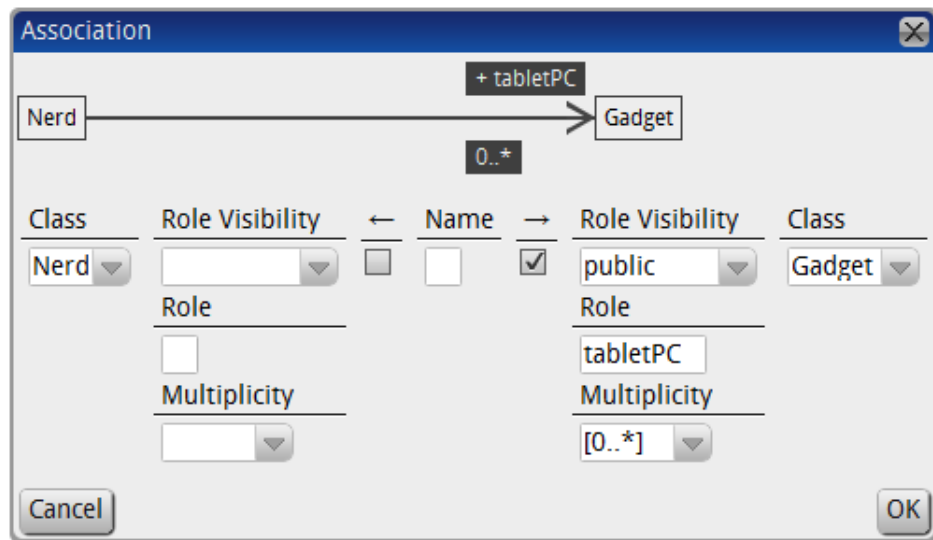


Figure 7: Dialog for adding a new binary association

#### 1.6.4 Generalization

#### 1.6.5 Required Interface

#### 1.6.6 Provided Interface

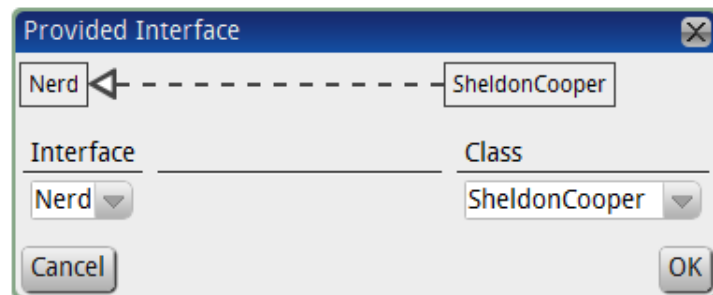


Figure 8: Dialog for setting a new provided interface

#### 1.6.7 Round-trip Engineering

## 2 Implementation

### 2.1 Frameworks

AlphaUML use a handful collection of libraries, which are used to improve the development process and allows the access to necessary system information, like the file system. Each framework is based on JavaScript, like AlphaUML, or supports JavaScript, so its theoretical possible to run the application in a web browser.

#### 2.1.1 Adobe Air

With Adobe Air its possible to create web applications, which will be executed as a native application on a desktop PC or a tablet. It supports the development in JavaScript and HTML and provides a library, which gives access to some system side functions and programs, like the command console or the file system. Adobe Air use the WebKit engine to allow to execute a JavaScript application. This fact is also necessary to know for designing the application 2.2.1.

#### 2.1.2 Dojo Toolkit

Dojo is an enhanced framework to build JavaScript web applications (apps) and provides a huge number of features, which speeds up the development. Furthermore its very useful to build JavaScript apps in an object oriented style. So it allows to use a package system, has a build-in class system, which simulates for example inheritance. In addition one of the most important features is the event system. So user-triggered events, like an "onclick" event can be caught. Moreover self-defined events can be thrown, which release specific functions to react on these events.

**Package System** The package system is based on the folder structure of its project. So if a JavaScript file has the following path "lib/ui/Dialog.js" and its necessary to get access to the Dialog class, the dojo.require function has to include this file by setting its path as first parameter. The require function includes the file during the runtime, so it isn't necessary to load the whole script at the beginning, which will saving a lot of loading time.

**Class System** The class system allows the declaration of classes and provides inheritance, which is necessary to develop software in object oriented style. So JavaScript is object oriented, but doesn't support to inherit from another object and it doesn't have classes. So Dojo uses some tricks to simulate these features.

- Prototypal Inheritance
- Classical Inheritance
- Private Members

**Event System** The use of events is particularly in JavaScript useful. So on the one hand its necessary to catch events from the DOM, which thrown by the user, for example on a click on a button. On the other hand self-defined events allow reacting in special cases, if an error is thrown or a specific routine is ready. In those cases a so called callback method can be executed, which can handle the result of the routine or shows for example an error message.

### 2.1.3 PEG.js

PEG.js is a JavaScript library, which generates a parser with the help of a specific grammar. So its possible to parse each programming language or to build a calculator, or something else. For more information show 3.1.1.

### 2.1.4 Ace

Ace is a fast code editor with syntax highlighting and search- and replace-functions. It is fully developed in JavaScript and currently focused on JavaScript and CoffeeScript development, but allows using syntax highlighting for other languages, like Java, too.

## 2.2 User Interface (UI)

### 2.2.1 Graphical Design

To design the graphical user interface a style sheet language called cascading style sheets (CSS) is used. It describes the presentation semantics, which formats and sets the look of a HTML document. So it's possible to set the floating of elements, the size, the background color, specific fonts and many other formatting. In addition the style sheet is able to format each HTML element with an id or a class attribute. Classes will be accessed by a point and ids by a hash-symbol. If an element has a class called "Button" the selector looks like .Button and if an element has the id "dialog\_1" the selector will be #dialog\_1. 'Cause of the HTML limitation, that an id is unique, the id selector will format only one element, wherever the class selector is able to format an unlimited number of elements.

**JavaScript Specifics** In JavaScript CSS is also used for animations. So if the position of a DOM node will be changed, the element has to have an absolute position and the top and left property have to set. JavaScript writes that information in the DOM node and the web browser updates its view, which is need to realize a drag and drop feature or animations, like color and size changes.

**Adobe Air Specifics** The Adobe Air HTML- and JavaScript engine is based on WebKit, an open source web browser engine. So it's necessary to know that some CSS features like text- or box-shadows need the prefix "-webkit-", which usage will shown in the example below.

### 2.2.2 The basics of the GUI design

**UI Object** Each UI element, like a tab or the file tree, inherits from a base class called `ui.Object`. This class provides the managing of event listener, the creation of a unique id or handles the deletion of the UI element. Furthermore it implies some useful utilities, like hiding or showing the UI element. But one of the most important features is the event manager. So it's necessary to unset each event listener, which listen to a removed DOM node. In addition the event manager provides the registration of new event listener for specific events and can deactivate, activate or remove each listener by name. Furthermore it's possible to activate or deactivate all listeners at the same time, what's necessary if a UI object will be destroyed. The "activate" and "deactivate" function can be overwritten to extend the functionality, by example to destroy the DOM node at deactivating. Another important feature is the generation of a unique id for the DOM node of the UI element. So the HTML standard defines that an id has to use once only. If two nodes have the same id, the JavaScript engine of the web-browser can't get access to both. To get a unique id the id-creator uses the class name as base and searches for other nodes of that class, counts each appearance and appends the counter to the class name. The last mentionable function of the base class is the deletion handling. To be sure that each registered event listener and each used UI element of an UI element, like a button of a dialog, including the DOM node will be destroyed, the base class has a specific destroy function, which handles the "garbage collection". To initialize an UI object a function called "create" will be used. It generates a unique id, places the DOM node and sets up all necessary event listeners. A UI object has a specific `uitype`, which will be used to identify a DOM node as a UI element. It will be set as a HTML attribute and can be accessed on a catch event.

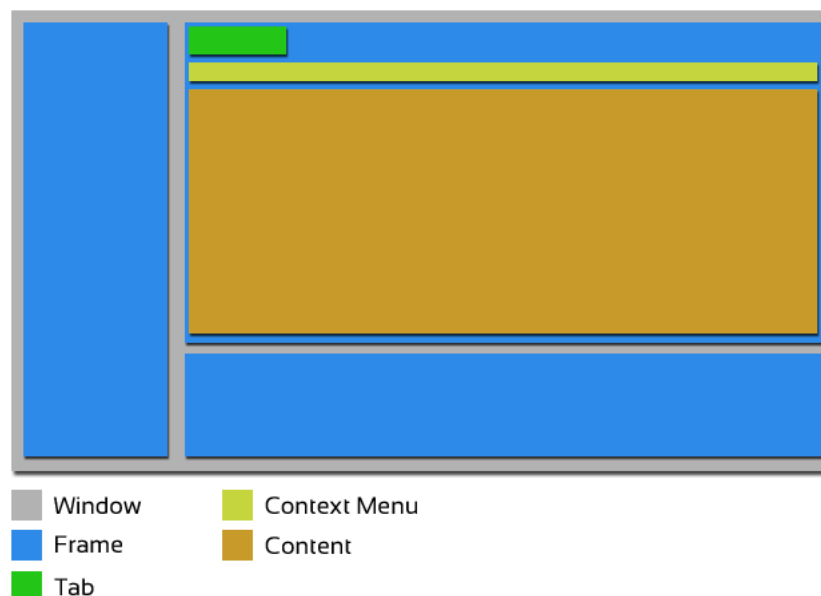
**Window** On the top of the GUI the window object manages the resizing and the full-screen mode of the Adobe Air window. It contains a collection of frames, which will be placed and resized with the help of the current window properties.

**Frame** A frame is the main actor in handling tabs and its content. In addition it resizes the tab content, like the Java editor or the file tree, and allows opening new tabs. In addition it handles the replacing and deletion of tabs. To drag a tab into another frame, the tab, its content and context menu will be removed from the old frame and replaced into the new one. In this case only the DOM nodes will be destroyed, but the Meta data - like the source code of a specific editor tab - because its necessary to be able to recreate the tab.

**Tab** A tab is only a draggable button, which references to its content and context menu. It's used to show or hide a specific content, like a code editor. A tab is subdivided into 3 parts. At first the tab themselves, which handles the create- and destroy-process of its content and context menu. In addition it starts its replacing process, which will be handled by the parent frame, if a user begins to drag the tab.

**Context Menu** The next part of a tab is its context menu. The context menu holds a collection of buttons, which can be used to execute specific functions of a tab. For example the editor tab has buttons to save its current document or to resize the font size.

**Content** The last part is the content object. It provides the access of some tools, like the class diagram editor, a Java editor or a file tree.



### 2.2.3 Additional GUI components

The GUI framework provides a number of UI components, which will be necessary for an application. The following list shows each available component.

**Dialog**

**Tooltip**

**MouseInfo**

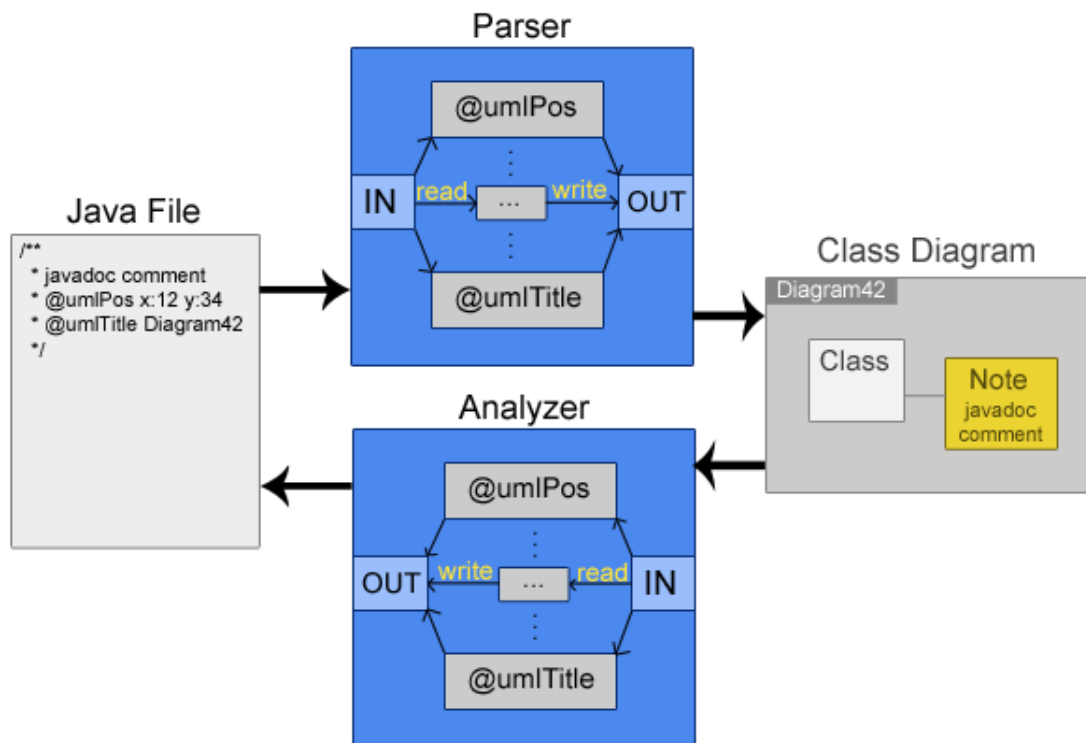
## 2.3 Single Page Web App

Usually a web application is using a handful of HTML pages for rendering its content. A single page web app has only one HTML document, which will be updated by some JavaScript functions. So it's possible to place, replace or remove DOM nodes during the runtime. Another characteristic of a web application is its fragmentation into backend - usually server-side - and frontend - client-side -, which concept is also be used in a single page web app. But AlphaUML is designed as a single page web app without a

backend, which could be used for parsing or for getting access to files from storage. All these functionality is transferred to the frontend, so each process will be executed on the client side. To realize that concept, the compiler and any other feature is completely written in JavaScript. In addition Adobe Air is used to get access to functionalities, which JavaScript isn't allowed to. For example JavaScript can't get access to the file system of a client, 'cause of security reasons. In this case Adobe Air is the way to go.

## 2.4 Single Source Publishing

To avoid project files AlphaUML follows the single source publishing concept. It allows using the context of a document in different ways. In this case a Java document is used to initialize a project or to generate a class-diagram, without losing its specific usage as source file. So each necessary project-information will be saved into existing Java files. This is realized by using a various selection of specific JavaDoc tags, which begin with the @uml prefix. To use those tags the compiler is optimized to read them during the reverse engineering process 3.1. Furthermore the UML specific tags will be generated by the code generator 3.2.



### 2.4.1 Class Diagram Specifics

**@umlPos x:42 y:42** The position-tag defines the position - in pixel - of a class in the class diagram.

**@umlNotePos x:42 y:42** If a class has a comment included, the position of the note will be stored in the note-position-tag. In addition if more than one note is connected to a class, the position of the first note will be used.

**@umlTitle title-text** The title-tag defines the title of the last generated class-digram.

**JavaDoc Comment** A special case is the JavaDoc comment itself. It is used to generate notes, which will be pinned to its associated class. In addition if the user creates a new note, the code generator uses as JavaDoc class comment. Moreover multiple usages of notes are allowed, so each note comment will be merged to one comment.

## 2.4.2 Project Specifics

**@umlProject root:path/folder main:path/file.java** The project-tag is used to define the project main file. It includes the relative root path and the main Java file with an absolute path.

**@umlIgnore path** To exclude specific files or folder, the ignore-tag allows ignoring those for the class diagram generation. That could be necessary if only a part of files should include in the class diagram, or if some Java files dont follow the java syntax completely, 'cause of its use to store only Java snippets.



## 3 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.

### 3.1 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.

#### 3.1.1 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.

#### 3.1.2 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.

#### **3.1.3 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    }
```

#### **3.1.4 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  }  
12 ]
```

```

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 }

```

### 3.1.5 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

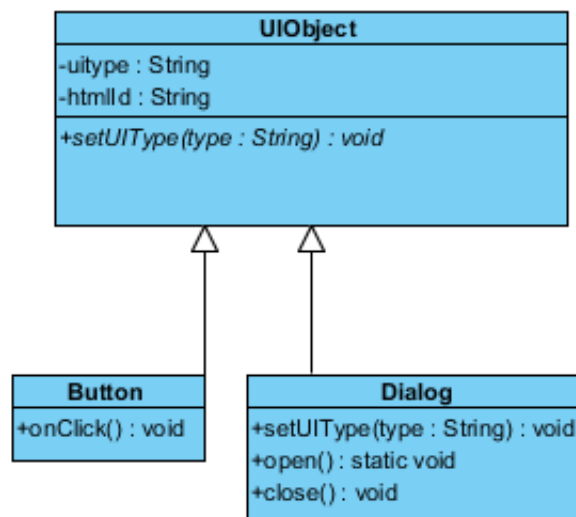
provided interfaces can be created in step two. The last step needs the interpretation of each created class. To generate associations or the relations of required interfaces, each attribute of each class has to be analyzed. If an attribute has a data type of an existing class, it's a candidate for an association. In this case this class will be analyzed, too, to figure out whether the association has bidirectional navigation.

## 3.2 Forward engineering

To generate the source code from an existing class diagram, it's 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 3.1.4 and will be used to generate the source code. Furthermore it gets the relations between classes, so an additional process isn't 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.

### 3.2.1 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 doesn't 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.

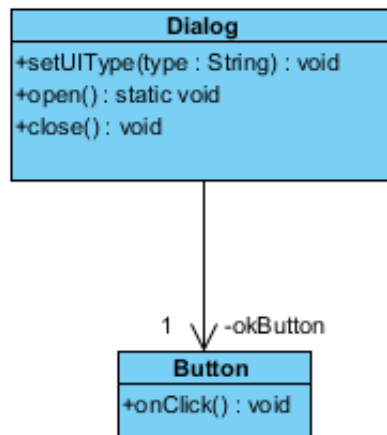
Listing 8: The following code snippet shows the result of the code generation.

```
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 }
```

### 3.2.2 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.

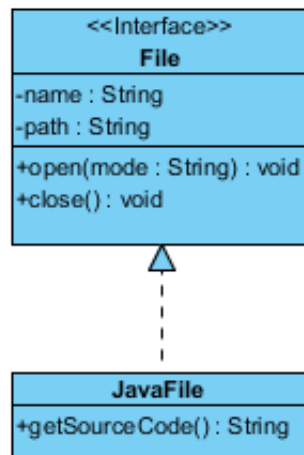
Listing 9: The following code snippet shows the result of the code generation.

```

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 }
  
```

### 3.2.3 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.



This UML diagram shows the class "JavaFile", which use the "File" interface for a standardized communication. To use the interface the interface visibility is set to "public" and the class, which implements it, will redefine each operation of the interface.

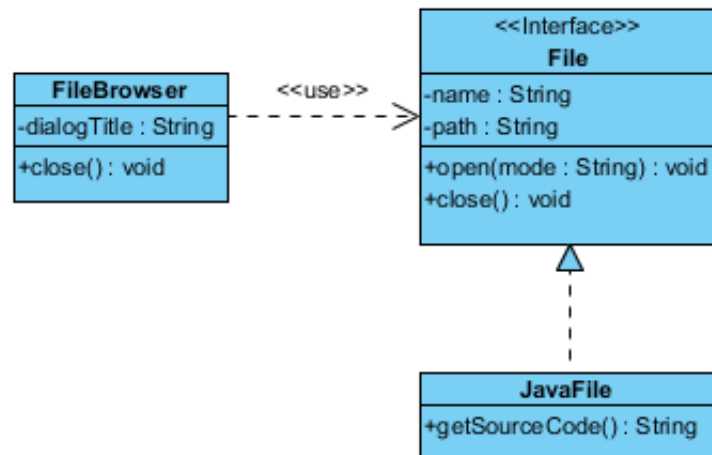
Listing 10: The following code snippet shows the result of the code generation.

```

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 }
  
```

### 3.2.4 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.



In this example the **FileBrowser** uses an interface to get a standardized access to each file type, which implements the "File" interface. In this case a file could be a **JavaFile** or maybe a **TextFile**, or something else. The required interface will be declared as an attribute with the data type "File" and can be initialized with the **JavaFile** constructor *private File javaFile = new JavaFile("read");*. But the code generator sets only the attribute.

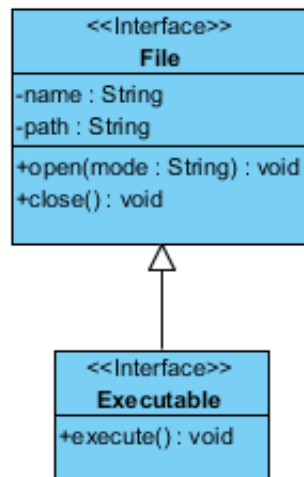
Listing 11: The following code snippet shows the result of the code generation.

```

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 }
  
```

### 3.2.5 Extended interfaces

If an interface will be extended, a generalization 3.2.1 has to be used. That includes the UML generalization arrow and on the Java side, the use of the `extends` keyword.



In this case the Executable interface extends the File interface with the execute function.

Listing 12: 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 interface Executable extends File {
11     public void execute() {}
12 }
```