

# **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

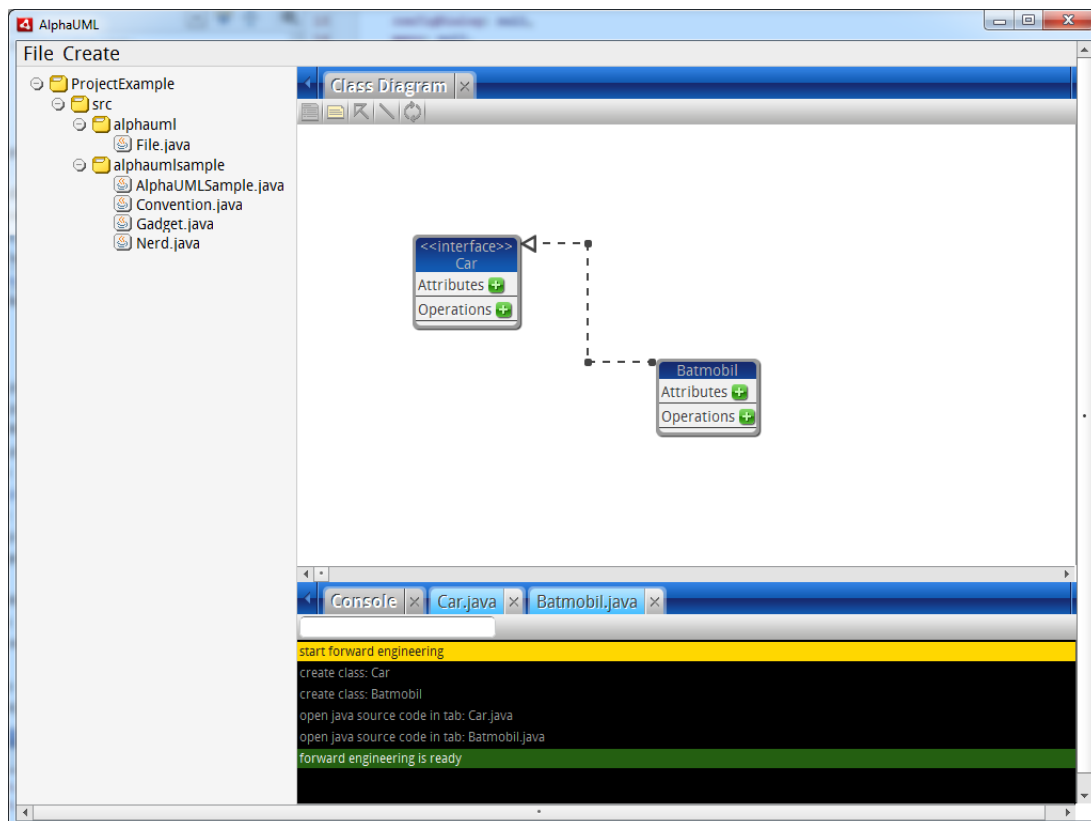
<b>1</b>	<b>AlphaUML</b>	<b>4</b>
1.1	Introduction . . . . .	4
1.2	Configuration . . . . .	4
1.3	Projects . . . . .	5
1.3.1	Folder Structure . . . . .	5
1.3.2	New Project . . . . .	5
1.3.3	Load Existing Project . . . . .	5
1.4	Console . . . . .	5
1.4.1	Java Compiler . . . . .	6
1.4.2	JavaDoc Generator . . . . .	7
1.5	Code Editor . . . . .	7
1.6	Class Diagram Editor . . . . .	8
1.6.1	Class . . . . .	9
1.6.2	Note . . . . .	13
1.6.3	Binary Association . . . . .	13
1.6.4	Generalization . . . . .	14
1.6.5	Required Interface . . . . .	15
1.6.6	Provided Interface . . . . .	15
1.6.7	Round-trip Engineering . . . . .	15
<b>2</b>	<b>Implementation</b>	<b>17</b>
2.1	Frameworks . . . . .	17
2.1.1	Adobe Air . . . . .	17
2.1.2	Dojo Toolkit . . . . .	17
2.1.3	PEG.js . . . . .	18
2.1.4	Ace . . . . .	18
2.2	User Interface (UI) . . . . .	18
2.2.1	Graphical Design . . . . .	18
2.2.2	The basics of the GUI design . . . . .	20
2.2.3	Additional GUI components . . . . .	22
2.3	Single Page Web Application . . . . .	22
2.4	Single Source Publishing . . . . .	23
2.4.1	Class Diagram Specifics . . . . .	23
2.4.2	Project Specifics . . . . .	24
<b>3</b>	<b>Round-trip engineering</b>	<b>25</b>
3.1	Reverse engineering . . . . .	25
3.1.1	Parser . . . . .	25
3.1.2	Parser grammar rules . . . . .	25
3.1.3	Parser grammar example . . . . .	27
3.1.4	Parser output . . . . .	27
3.1.5	Class diagram generation . . . . .	30

3.2	Forward engineering . . . . .	31
3.2.1	Generalization . . . . .	31
3.2.2	Binary association . . . . .	32
3.2.3	Provided interfaces . . . . .	33
3.2.4	Required interfaces . . . . .	34
3.2.5	Extended interfaces . . . . .	35

# 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 it's important that changes can be realized easily by a user which will enhance the learning-by-doing process. Moreover, AlphaUML is currently specialized on 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. The Java compiler 1.4.1 on the one hand and the JavaDoc generator 1.4.2 on the other hand need 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 Folder Structure**

AlphaUML uses a specific but not required folder structure. It is designed to be compatible to NetBeans and Eclipse, which are two common used integrated development environments (IDE). Moreover, it is also possible to use a self-defined folder structure by defining a specific source folder, which includes the Java files. In NetBeans and Eclipse usually the "src" folder is used to contain each Java file, which is used as main folder if it is found in the used project folder. Another important folder is the "build" folder – it is used when the source code is compiled. So each compiled Java class is found there. Furthermore, for the documentation output the "docs" folder is used.

### **1.3.2 New Project**

When a new project has to be created, the current opened project is closed and an empty project without any Java file is created. This project has the typical folder structure, which is described above 1.3.1. To create a new project the start menu entry "New Project" is used.

### **1.3.3 Load Existing Project**

To open an existing project the "Open Project" entry of the start menu is used. It opens a specific folder, which could be any folder. If the folder contains a "src" folder AlphaUML uses it as main folder for Java source files, otherwise the user has to declare a folder himself.

## **1.4 Console**

The console is a 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 occurs 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.

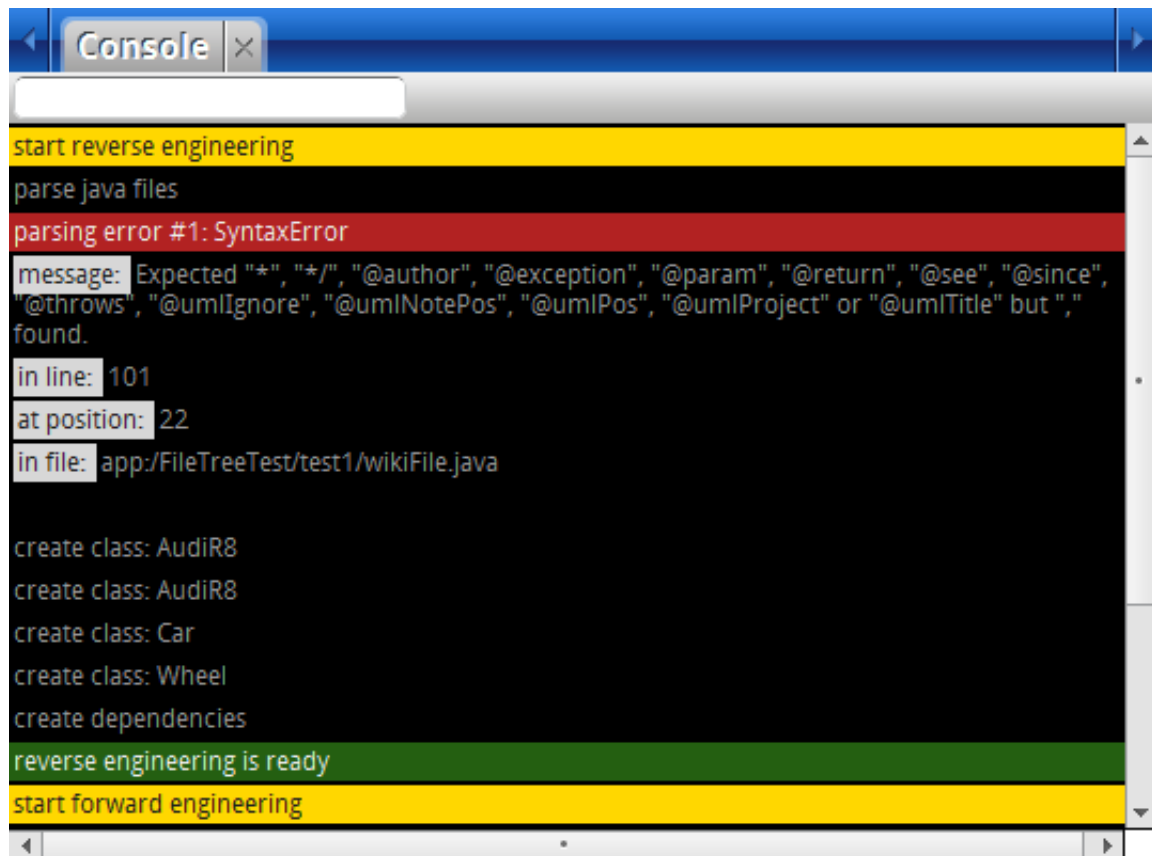


Figure 1: Console which showing the round-trip engineering process

#### 1.4.1 Java Compiler

To compile the currently opened project, the command "javac" can be used 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 can be found in the project's 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

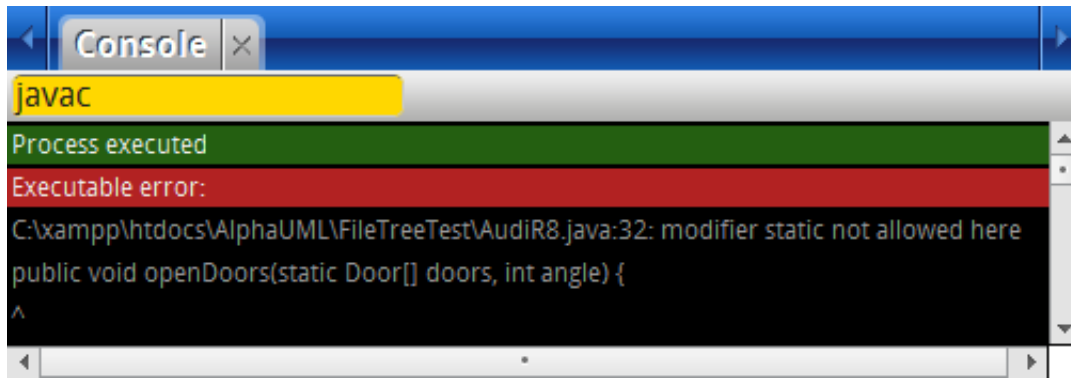


Figure 2: Execution of the Java compiler

### 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 project's root.

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

**Command**    `javadoc`

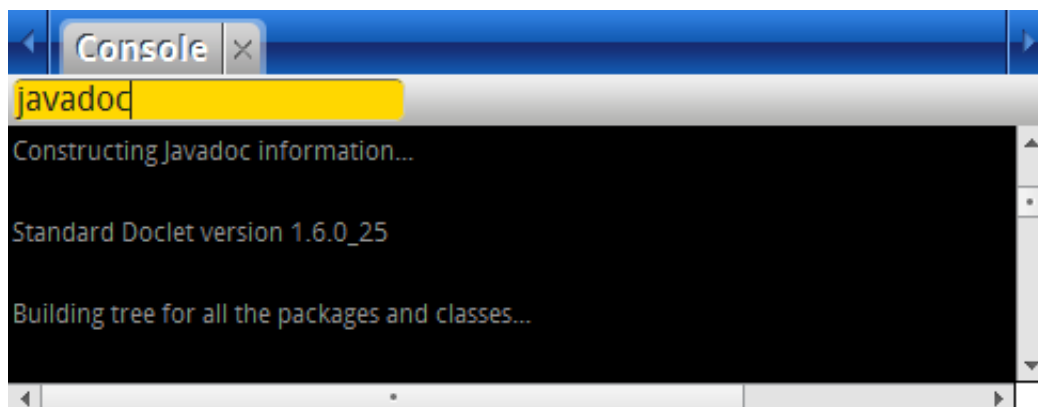


Figure 3: Execution of the JavaDoc generator

## 1.5 Code Editor

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

reason why some features aren't supported in AlphaUML like showing syntax errors. But this feature is available through the round-trip engineering and compiling process, where 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 is created without a specific file path – for example after the code generation process – a file dialog is opened, which can be used to save those files. Another feature provided by the code editor is to increment or decrement the font-size. This is especially useful whenever a user believes that the used font is too small which can be caused by an amblyopic. It's also possible to search and replace parts of the source code. This can be necessary if a document contains a huge number of characters and the user searches a specific part of the source code.

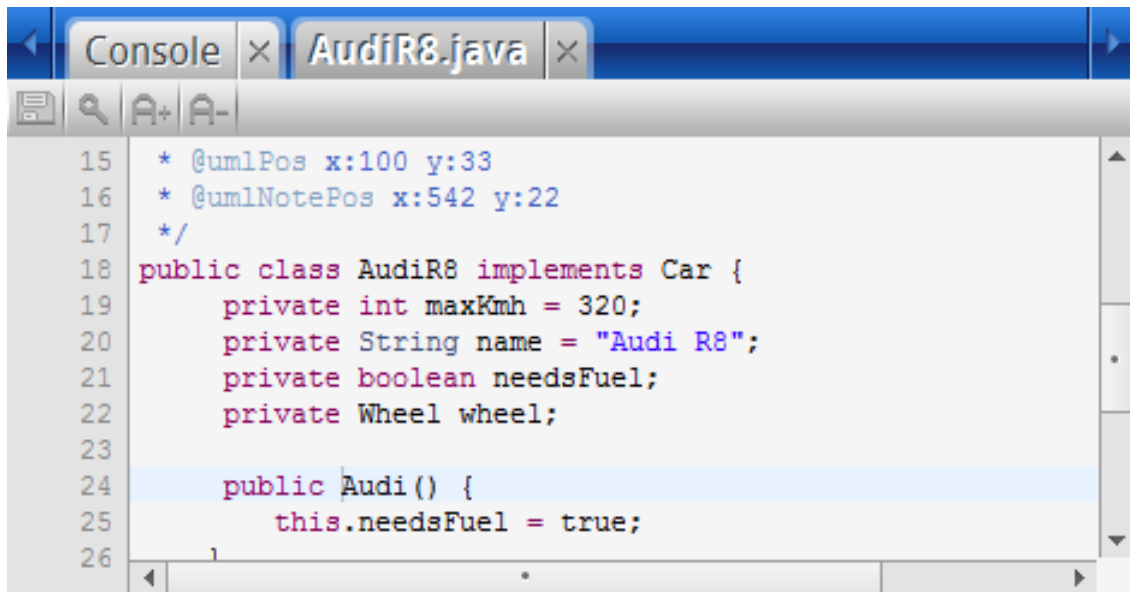


Figure 4: View of the code editor

## 1.6 Class Diagram Editor

The class diagram editor is a small editor to draft class diagrams with a handful of tools. These tools create classes, notes or relations like generalizations between existing classes. It consists of two components: the workspace and the toolbar, which gives access to each tool. Each tool is optimized for the Java programming language, to enhance Java developers.



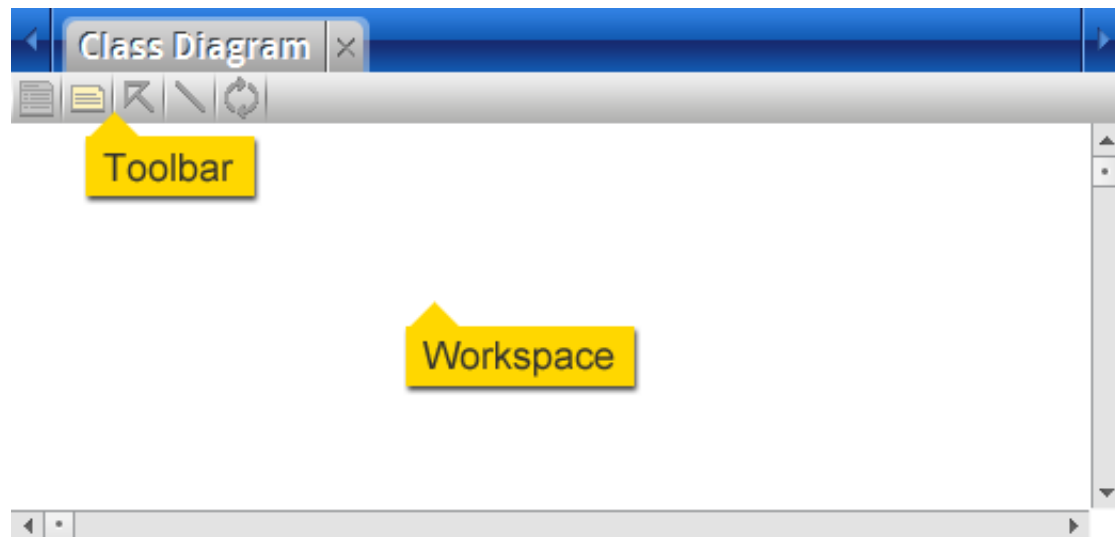


Figure 5: Layout of the class diagram editor

**Multiple use-cases of tools** Some of the tools can be used in multiple ways. This is because certain UML components visually resemble each other. For example the generalization- and a provided interface arrow use the same arrowhead. Moreover it allows keeping the UI simple, because it isn't necessary to provide a huge number of buttons. If a user uses the "Add Generalization" tool and selects two classes a generalization will be created, but if the user selects an interface and a class, the tool adds a provided interface. To simplify the usage of those tools tooltips will be used to show each possible usage.

**Supported UML notation elements** The editor supports the following UML notation elements, which are described in the next chapters.

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.

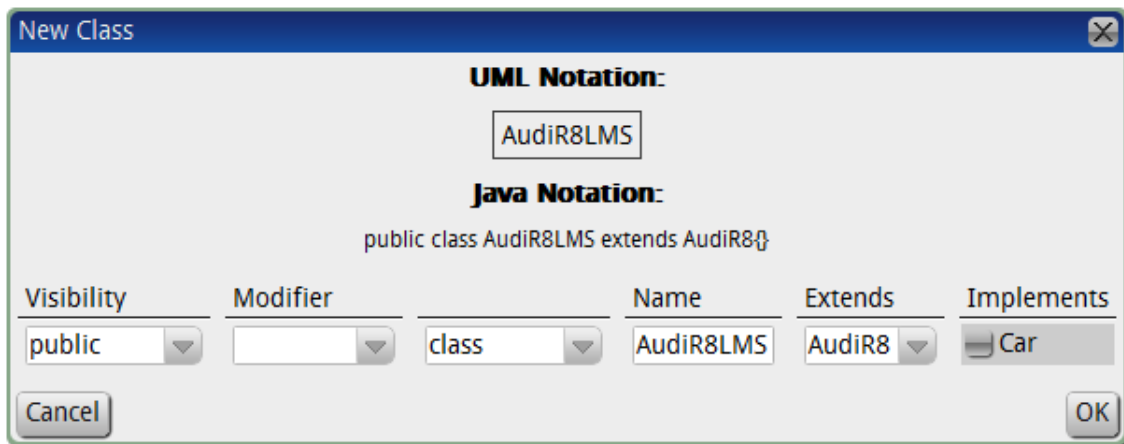


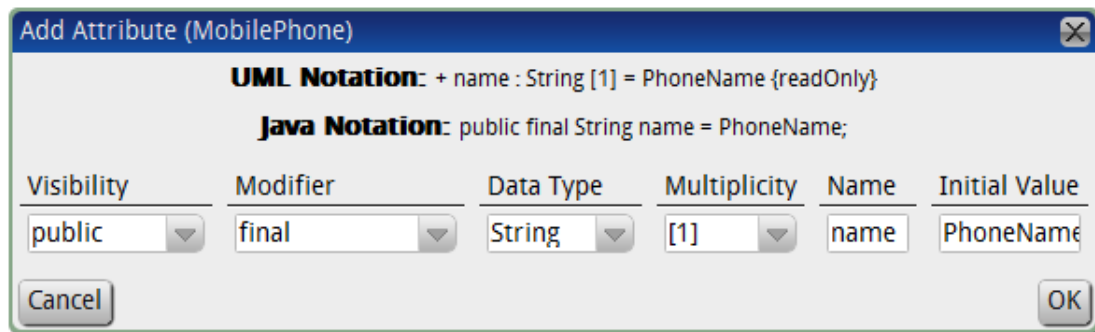
Figure 6: Dialog for adding a new class

After placing the class a dialog is opened to configure the new class. It allows setting a class name, modifier or its visibility. Furthermore it's also possible to implement interfaces or to inherit from another class via the "Extends" option. A special characteristic of each dialog, which is used to add a class diagram-component, is its subdivision into 2 parts. On the top there is a preview, which shows the result of the current configuration as Java and a UML component. And at the bottom each component property can be set. So the user can see his changes directly.



Figure 7: View of a class

When a class is placed, it provides a number of functionalities like adding attributes or operations. The most important functionality is the context menu of a class, which gives access to the "Edit Class" tool and gives possibilities to rename or remove the class. The context menu is opened by a click on the class name. For adding a new attribute the plus icon beside the "Attribute" header – Figure 7 number 1 – has to be clicked, which opens the "Add Attribute" dialog.



**Add Attribute (MobilePhone)**

**UML Notation:** + name : String [1] = PhoneName {readOnly}

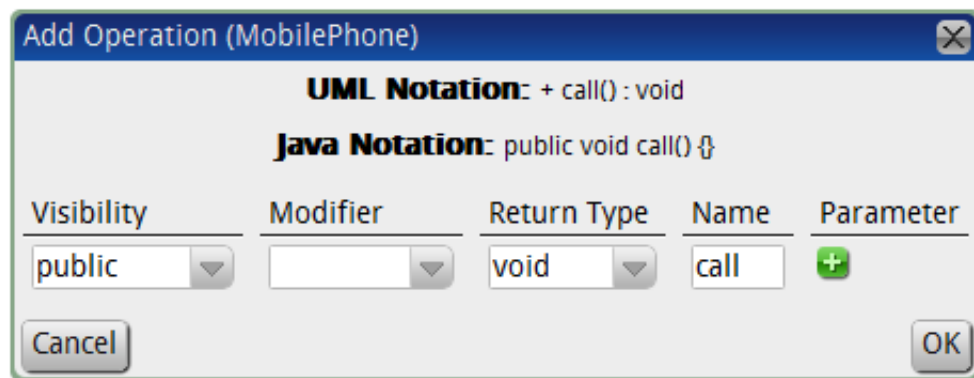
**Java Notation:** public final String name = PhoneName;

Visibility	Modifier	Data Type	Multiplicity	Name	Initial Value
public	final	String	[1]	name	PhoneName

Cancel OK

Figure 8: Dialog for adding a new attribute

To create a new operation, the plus icon beside the "Operation" header – Figure 7 number 2 – is the way to open the specific dialog.



**Add Operation (MobilePhone)**

**UML Notation:** + call() : void

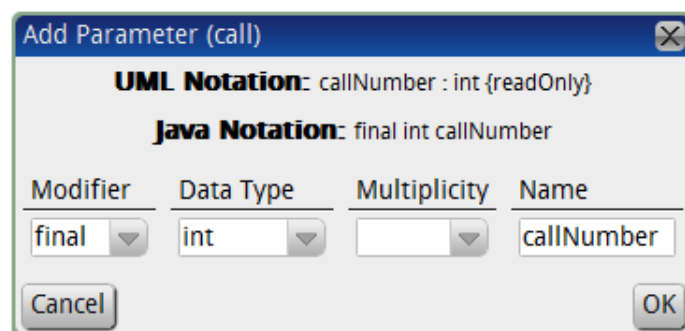
**Java Notation:** public void call() {}

Visibility	Modifier	Return Type	Name	Parameter
public		void	call	+

Cancel OK

Figure 9: Dialog for adding a new operation

In addition to the typical operation properties it's possible to add parameters via the "Add Operation" dialog. Parameters can also be added by using its plus icon, which is shown in any operation entry. In both cases the same dialog for adding parameters is opened.



**Add Parameter (call)**

**UML Notation:** callNumber : int {readOnly}

**Java Notation:** final int callNumber

Modifier	Data Type	Multiplicity	Name
final	int		callNumber

Cancel OK

Figure 10: Dialog for adding a new parameter

Each operation and attribute is build on so-called tags, which are all clickable. A click on the name of an attribute or operation will open a context menu for configuring and deleting it. Each other tag is used for a quick access on its value. So it's possible to change the data type of an attribute by clicking on it and selecting another one from the opened context menu.

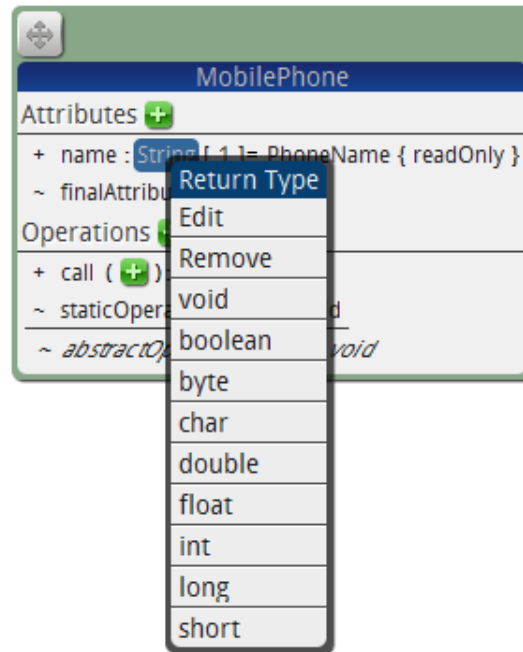


Figure 11: Quick access

In some cases a Java modifier can't display as specific UML keyword. So it's commonly accepted to display attributes or operations with a different formation. Abstract class members are shown with an italic font and static members are underlined. In addition final members use the {readOnly} property.

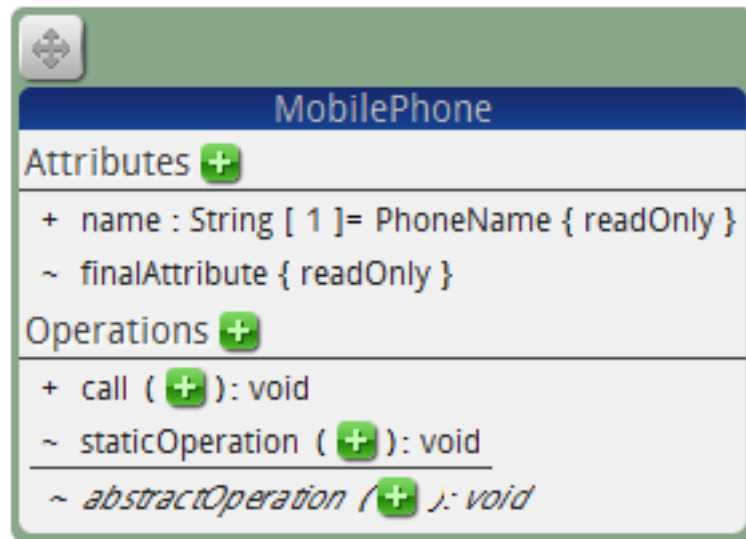


Figure 12: View of an final attribute and an abstract and static operation

### 1.6.2 Note

A note is used to document a specific class. Its comment will also be used to generate a JavaDoc comment during the forward engineering process 1.6.7. It's also possible to create a note without a relation to a class, which could be useful for small memos. To create a note the toolbar provides the "Add Note" tool, which allows placing a note into the workspace. If it's necessary to connect a note to a class the "Add Association" tool 1.6.3 can be used.

### 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 is used. Moreover, it's possible to set the role of a class which is the name of its reference. Furthermore the multiplicity to set the min and max number of allowed instances, can also be set. The association name is 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 desired classes have to be selected. And after these steps a dialog is opened, which provides the configuration of all necessary association properties.

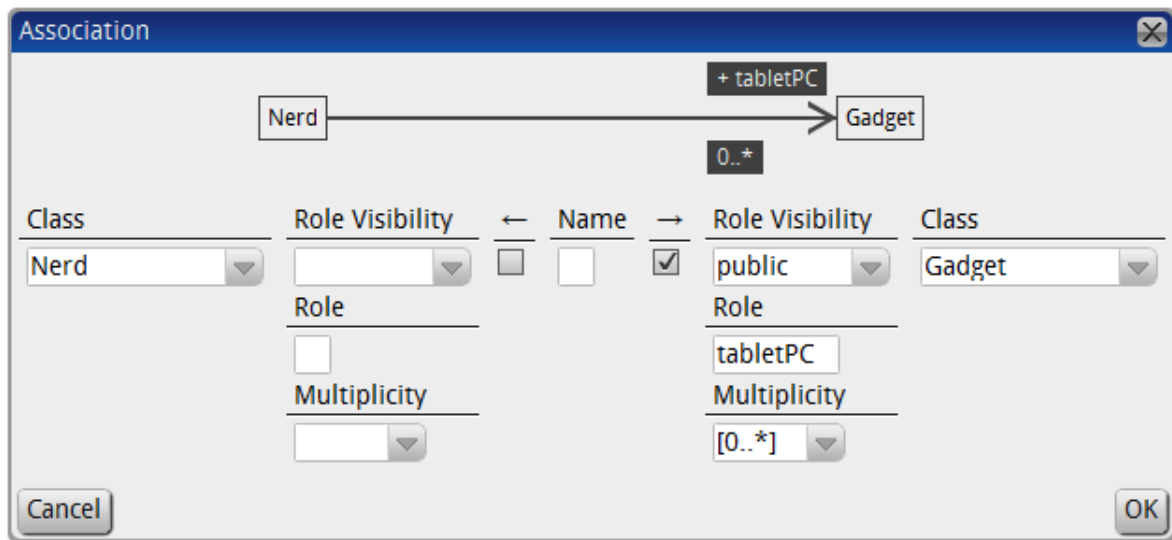


Figure 13: Dialog for adding a new binary association

#### 1.6.4 Generalization

A generalization describes a relation between a super- and a subclass and specifies that the subclass inherits from the superclass. So each attribute and operation of the super-class is also available in the subclass. It's also possible to overwrite an operation, which can be handled by declaring the same operation with identical parameters like the one in the superclass. Otherwise the superclass operation will be used. To set a generalization the "Add Generalization" tool of the toolbar is used. After activating it the superclass has to be selected and then the subclass, which opens a dialog for setting it up. To edit this relation it's possible to click the generalization for opening a context menu, which also allows its deletion.

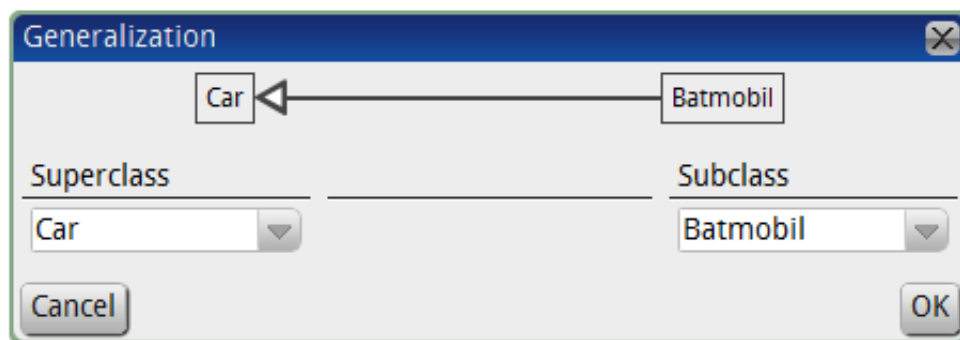


Figure 14: Dialog for adding a new generalization

### 1.6.5 Required Interface

An interface is a special kind of an abstract class, with the difference that it's possible to inherit more than one interface. When a class uses a specific interface it's called a required interface, which can be set with the "Add Association" tool.

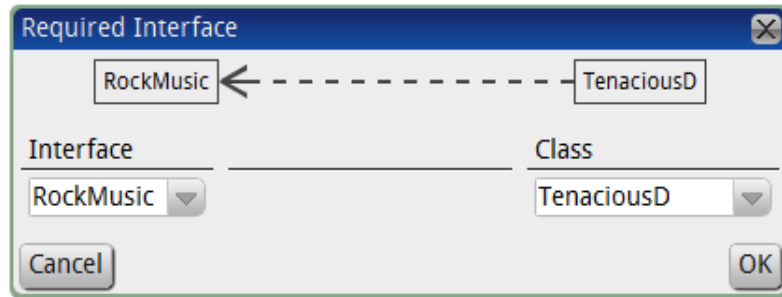


Figure 15: Dialog for setting a new required interface

### 1.6.6 Provided Interface

Moreover, if a class inherits from an interface, it's called provided interface and can be set via the "Add Generalization" tool.

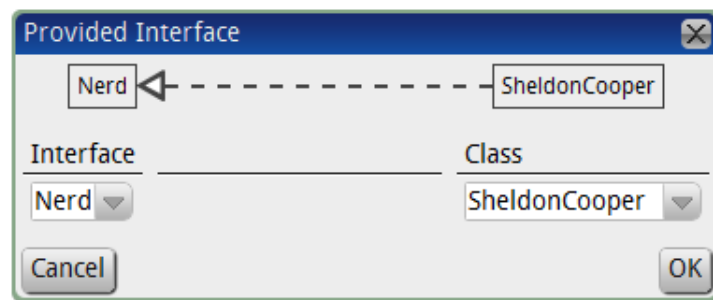


Figure 16: Dialog for setting a new provided interface

### 1.6.7 Round-trip Engineering

AlphaUML is able to generate a class diagram with the help of included Java source files of a project. Furthermore it's also possible to generate Java source code by analyzing a class diagram. These features are provided by the "Round-trip engineering" tool and can be executed by a simple click on its specific button. If a diagram has to be built and another one already exists in the parent tab, the old one is removed before a new one can be generated. So it's necessary to use a new tab, if the old diagram shouldn't be removed. To read more about the round-trip engineering process see 3.1.

**Forward Engineering** When Java source files are generated from a diagram, they are opened in blue accented editor tabs, which signalize that their included documents aren't

saved yet and they are a result of the code generation. Furthermore the generated sources include special JavaDoc tags 2.4, which allow generating the identical diagram after a restart of AlphaUML. The whole process with all necessary status information is shown in the console tab, so the user gets an overview of the running process.

**Reverse Engineering** In another case if a class diagram has to be generated, a parser parses each Java file and generates it with the help of the grammatical logic of the Java source. So it's necessary to use the correct Java grammar, otherwise the parser throws an error and skips the currently parsed file.



## 2 Implementation

### 2.1 Frameworks

AlphaUML uses a collection of libraries, which are used to improve the development process and allow access to necessary system information like the file system. Each framework is based on JavaScript or supports JavaScript, so it's theoretically possible to run the application in a web browser.

#### 2.1.1 Adobe Air

With Adobe Air it's possible to create web applications, which can be executed as native applications 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 uses the WebKit engine to allow executing JavaScript applications. It is necessary to keep this fact in mind when 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 speed up the development. Furthermore it's 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 various oop-related features – such as inheritance. In addition one of the most important features is the event system. With the help of this system 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 it's 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 saves 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 an object oriented style. JavaScript is object oriented, but doesn't support inheritance from another object and it doesn't have classes. So Dojo uses some tricks to simulate these features. The most important JavaScript tricks are shown above.

- Prototypal Inheritance <sup>1</sup>

---

<sup>1</sup><http://javascript.crockford.com/prototypal.html>

- Classical Inheritance <sup>2</sup>
- Private Members <sup>3</sup>

**Event System** The use of events is particularly useful in JavaScript. On the one hand it's necessary to catch events from the DOM, which are thrown by the user, for example in reaction to 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 an error message for example.

### 2.1.3 PEG.js

PEG.js is a JavaScript library, which generates a parser with the help of a specific grammar. So it's possible to parse each programming language or to build a calculator, or something else. For more information see 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-related features. In addition the style sheet is able to format each HTML element with an id or a class attribute. Because 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.

Listing 1: HTML snippet

```

1 <div class="outer">
2     <div class="inner"></div>
3 </div>
```

Listing 2: CSS snippet

```

1 .outer {
```

<sup>2</sup><http://javascript.crockford.com/inheritance.html>

<sup>3</sup><http://javascript.crockford.com/private.html>

```

2     width: 45px;
3     height: 60px;
4     margin: 10px;
5     background: #778899;
6     position: relative;
7     -webkit-border-radius: 5px;
8     -webkit-box-shadow: 1px 1px 1px #313131;
9 }
10
11 .inner {
12     top: 20px;
13     left: 10px;
14     width: 30px;
15     height: 30px;
16     background: #d3d3d3;
17     position: absolute;
18     -webkit-border-radius: 2px;
19     border: 1px solid #313131;
20 }

```



Figure 17: result of the css formatting

**JavaScript Specifics** In JavaScript, CSS is also used for animations. So if the position of a DOM node is changed, the element has to have an absolute position and the top and left property have to be set. JavaScript writes that information in the DOM node and the web browser updates its view, which is needed 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-". The usage will be shown in the example above – Listing 2.

### 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 management of event listeners, 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 listens to a removed DOM node. In addition, the event manager provides the registration of new event listeners 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 is destroyed. The "activate" and "deactivate" function can be overwritten to extend the functionality, for example to destroy the DOM node on deactivation. 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 be used only once. 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 is destroyed, the base class has a specific destroy function, which handles the "garbage collection". To initialize an UI object a function called "create" is used. It generates a unique id, places the DOM node and sets up all necessary event listeners. An UI object has a specific uitype, which is used to identify a DOM node as a UI element. It is 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 are placed and resized with the help of the current window properties.

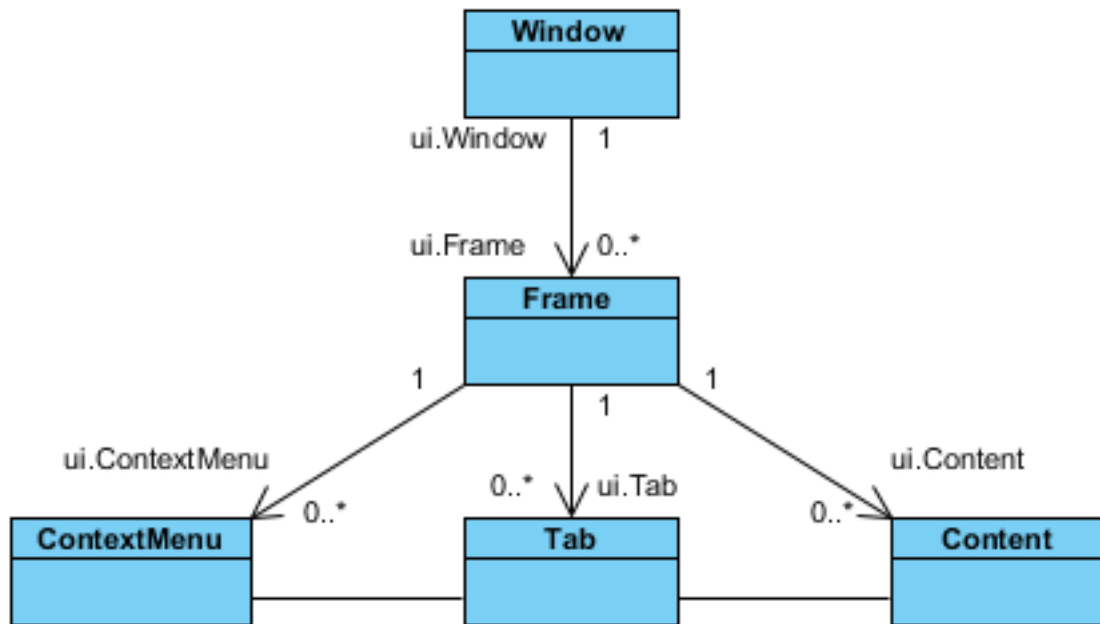


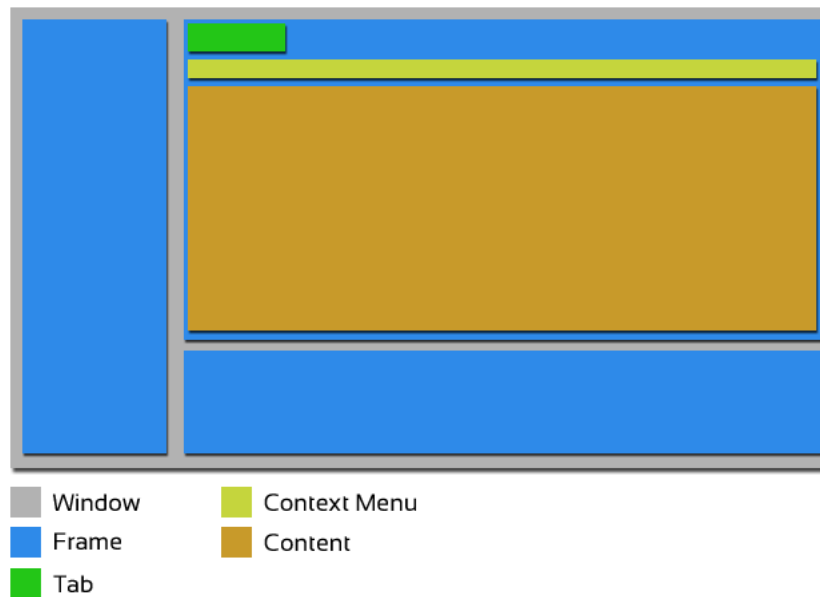
Figure 18: logical structure of a window

**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 replacing and deleting tabs. To drag a tab into another frame, the tab, its content and context menu are removed from the old frame and placed into the new one. In this case, only the DOM nodes are destroyed, in which the Meta data like the source code of a specific editor tab keep stored because the Meta data is necessary for recreating the tab.

**Tab** A tab is a draggable button, which references its content and context menu. It's used to show or hide a specific content, like a code editor. A tab is subdivided into three parts. At first, the tab itself, 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 access to 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**

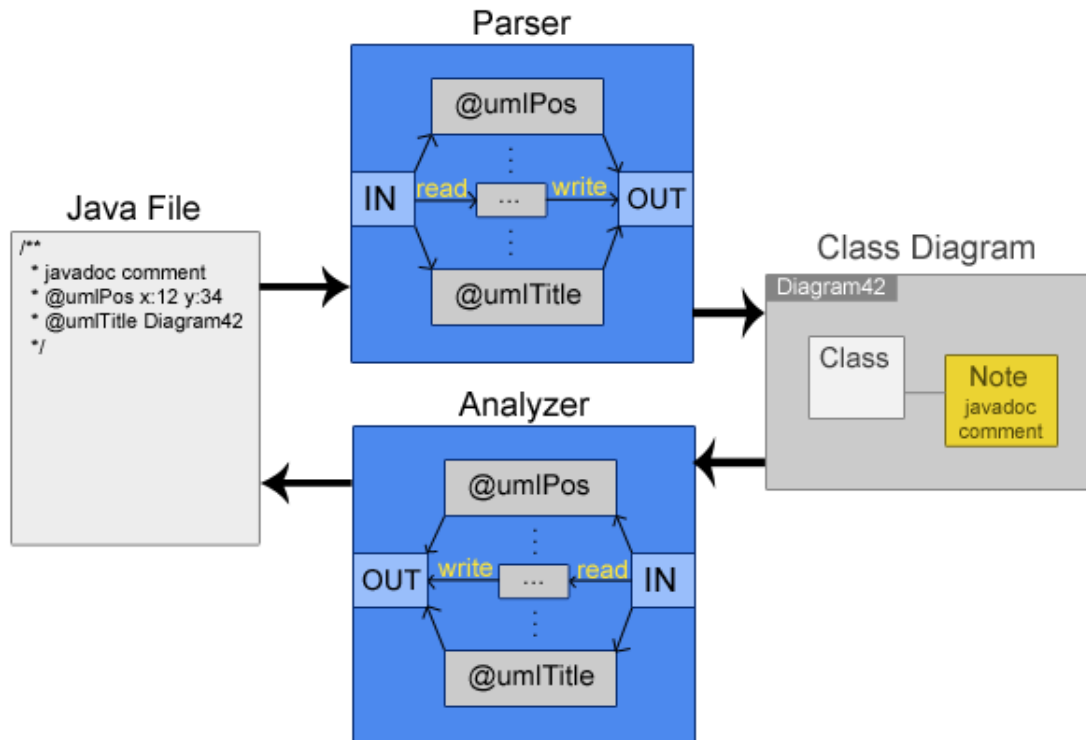
**MouseInfo**

## 2.3 Single Page Web Application

Usually a web application is using a number of HTML pages for rendering its content. A single page web application (app) has only one HTML document, which is updated by some JavaScript functions. So it's possible to place, replace or remove DOM nodes during runtime. Another characteristic of a web application is its fragmentation into backend – usually server-side – and frontend – client-side. This concept is also used by single page web apps. AlphaUML however, is designed as a single page web app without a backend, which could be used for parsing or getting access to files from storage. All these functionalities are transferred to the frontend, so each process is 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 doesn't have. For example JavaScript can't get access to the file system of a client, because of security reasons. In this case Adobe Air helps.

## 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 is saved into existing Java files. This is realized by using a 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 UML specific tags are 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 are pinned to their associated classes. In addition, if the user creates

a new note, the code generator uses it as JavaDoc class comment. Moreover multiple usages of notes are allowed, so each note comment is 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 don't follow the java syntax completely, because of its use to store only Java snippets.



## 3 Round-trip engineering

Today's complexity of software requires the use of diagrams that describe the structure of programs and how the internal processes communicate, during and primarily at the beginning of development. Moreover for object oriented languages like Java, the UML standard is the way to go, because of its focus on object oriented development 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 is to convert it into a specific data format with the help of a parser, in this case into a JSON object, which can be used to interpret the grammatical structure. After this 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 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 is 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. It's 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 3: 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 4: 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 is 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 5: file output

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

Listing 6: 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 7: 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 8: 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 9: 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 three parts. In the first step each class with all selected info – like the name, its variables and operations – are created. If two classes with the same name exist, the name of the second class is renamed. In this case each relation which has a reference on its class name can not be created anymore. During the first step any information which is needed for generalizations or the relations of provided interfaces is 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.

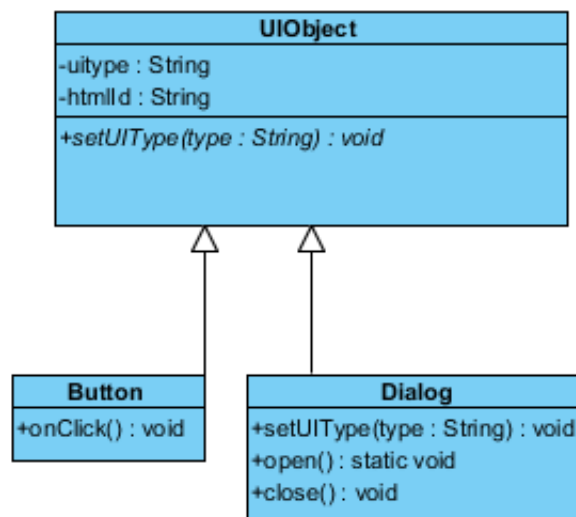
## 3.2 Forward engineering

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. A class contains all necessary information and can handle it without 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.

Attributes, associations and required interfaces are set as a variable. Moreover, if an interface extends another interface it will be set as superclass and be declared with the extend keyword. In the other case, if a class provides an interface, the implements keyword is 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. If the user defines an operation which has the same name and parameter list of another one in the superclass, the superclass operation is 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.

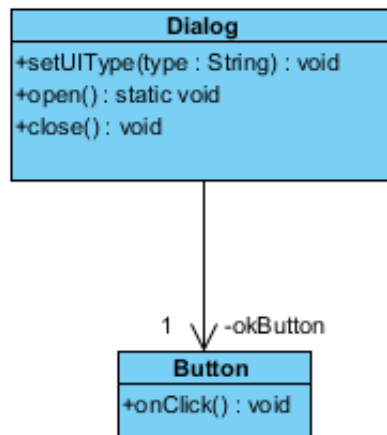
Listing 10: The following code snippet shows the result of the code generation.

```
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 knows 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 number of allowed instances. If a multiplicity is declared as [1..5] the class has to be instantiated at least once and can be instantiated at most five times. In the source code a multiplicity with more than one allowed instances, is represented as an array. Otherwise it's a simple variable. Furthermore the role of a class gives the instance of itself a specific name. 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 called "okButton". In addition it's accessible in the dialog class only, because of the "private" visibility. Moreover the multiplicity of 1 provides that only one button can be instantiated.

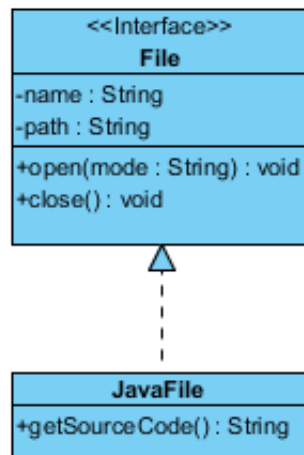
Listing 11: The following code snippet shows the result of the code generation.

```

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 be overwritten in the subclass. The only two differences are the "implements" keyword - instead of the "extends" keyword - which is used 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 during its declaration. An interface provides abstract methods only, which force a specific implementation of the class that inherits from it.



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

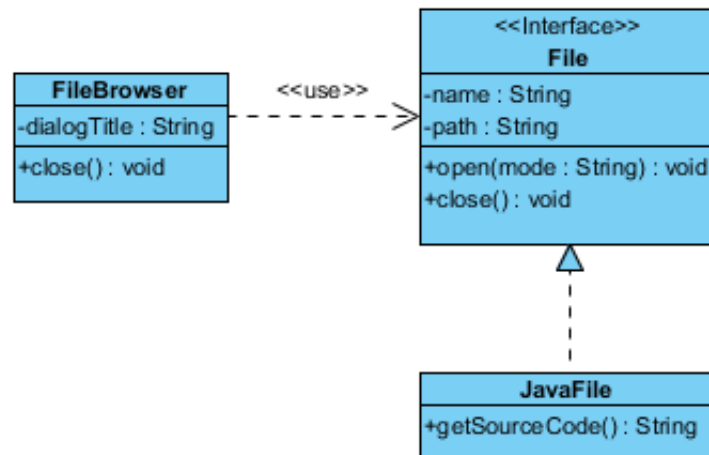
Listing 12: The following code snippet shows the result of the code generation.

```

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 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 inherits 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. Exceptions would be thrown if one of those operations is changed. A required interface is declared as an attribute and can be initialized by any constructor of a class that implements 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 only sets the attribute.

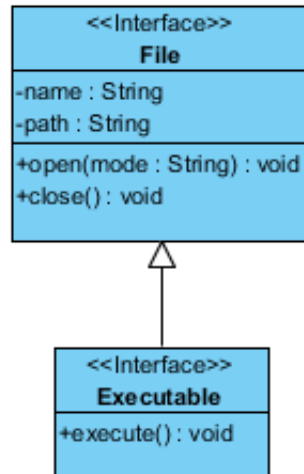
Listing 13: The following code snippet shows the result of the code generation.

```

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 is extended, a generalization 3.2.1 has to be used. That includes the UML generalization arrow and the use of the "extends" keyword on the Java-side.



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

Listing 14: 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 }
```