FP7-ICT-2011-8

MARKOS

The MARKet for Open Source

An Intelligent Virtual Open Source Marketplace

WP4 – Software Modelling and Analysis

D4.3.1 – Dependency Analysis Approach

Due date: M20	**Delivery Date**: **M20**
Author: Gabriele Bavota, Massimiliano Di Penta, Sebastiano Panichella **Partners contributed**: UNISANNIO	
Dissemination level: PU	**Nature of the Deliverable:** R
Internal Reviewers: Roberto Pratola (ENG)	

VERSIONING		
VERSION	DATE	NAME, ORGANIZATION
1.0	**03/06/2014**	**GABRIELE BAVOTA, MASSIMILIANO DI PENTA, AND SEBASTIANO PANICHELLA – UNISANNIO**
2.0	**25/06/2014**	**GABRIELE BAVOTA, MASSIMILIANO DI PENTA, AND SEBASTIANO PANICHELLA – UNISANNIO**

Executive Summary: The general goal for the Code Analyser is to extract, from software repositories, information about software artefacts to support in a code search activity the retrieval of the software artefacts themselves. In particular, given a software project under analysis, the Code Analyser parses (i) the source code elements present in it with the aim of extracting structural information about the code (e.g., methods contained in the classes, dependencies between code components of the project under analysis like method invocations, etc.) and information about the declared licenses; (ii) the libraries imported by the project, in order to discover dependencies between different software projects; and (iii) textual and configuration files, to extract further information about dependencies between software projects and licenses. This document describes the approaches implemented in the Code Analyser aimed at analysing dependencies existing between software entities belonging to the same or to different software projects.

D4.3.1 – Dependency Analysis Approach

Table of Contents

LIST OF TABLES

1. INTRODUCTION

The general goal of the Code Analyser is to extract information about software artefacts that are afterwards retrieved, by the Markos system, during the code search activity. The code search can be performed at different levels of granularity, ranging from the search of packages/libraries able to fulfil a given piece of functionality, through the search of a source code file/class, to the search of a single method that can be relevant for a specific task, e.g. a method useful to implement a particular sorting algorithm.

Moreover, the Code Analyser must extract and process the source code licensing information that will be used to determine whether the discovered artefact is compliant from a legal point of view. In addition, it is necessary to extract dependencies, to identify requirements necessary to be fulfilled when one wants to use a discovered code artefact, and to determine whether the dependencies would create legal issues, e.g., because of interconnection with artefacts that are not compatible from a licensing point of view with the system.

Currently the Code Analyser supports the parsing of Java source code, including its extensions for developing web-applications (i.e., JSP/Servlet), HTML pages and Javascript code snippets. This document describes the approaches adopted by the Code Analyser in the analysis of dependencies existing between software entities.

1.1. Document organization

The document is organized in four sections including this introduction. Section 2 and 3 describe the approaches adopted by the Code Analyser to extract the dependencies from the code of the analysed projects. They report, respectively, the approaches for the analysis of the Java code and for the analysis of JSP, HTML and JavaScript code. Finally, section 4 concludes the document.

2. APPROACHES TO EXTRACT DEPENDENCIES IN JAVA CODE

Table 1 summarizes the dependencies extracted by the Code Analyser when parsing Java code. The reported dependencies can exist between code entities belonging to the same software release (e.g., a method m1 implemented in class C1 of Apache Tomcat 2.0 calls a method m2 implemented in class C2 of Apache Tomcat 2.0) or between code entities belonging to different releases (e.g., a method m1 implemented in class C1 of Apache Tomcat 2.0 calls a method m3 implemented in class C3 of an imported library). The dependencies *Import-as-jar* and *Declare-dependency* (Table 1) represent an exception to that as they can exist only in the latter case.

From now on we refer to the first as *internal dependencies* and to the second as *external dependencies*.

Name	Granularity	Description
Extend	class-to-class	A class can extend (i.e., specialize) another class
Implement	class-to-class(es)	A class can implement one or more interface
Import	file-to-class(es)	A source code file can import one or more classes
Contain_file	file-to-class(es)	A source code file can contain one or more classes
Contain_class	class-to-class(es)	A class can contain one or more classes (i.e., nested classes)
Use-as-type	class-to-class(es)	A class can declare attributes of a type C, where C is another class of the system
Call	method-to-method(s)	A method can invoke one or more methods
Use-as-type	method-to-class(es)	A method can declare local variables/parameters/return of type C, or use a class attribute of type C, where C is a class of the system
Copied	file-to-file	A file can be an exact copy of another file
Cloned	file-to-file	A file can be a clone of another file (i.e., not exactly copied, but very similar)
Import-as-jar	release-to-release	A release can import a jar representing another release
Declare-dependency	release-to-release/package	A release can declare in specific file its dependencies toward another release or a specific package of a release

Table 1: Dependencies captured in Java Code by the Code Analyser

The approaches used to extract *internal* and *external* dependencies are discussed in the following two subsections.

2.1. Internal dependencies

To parse Java code the Code Analyser exploits a *Java Fact Extractor* built on top of *srcML*[1], a toolkit able to construct an XML-based representation (called srcML representation) of a

[1] M. L. Collard, H. H. Kagdi, and J. I. Maletic, "An XML-based lightweight C++ fact extractor," in 11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA. IEEE Computer Society, 2003, pp. 134–143.

given source code (more details are available in deliverable D4.2.1b). In the following, it is reported an example created around the Java class LucenePackage.java also highlighting its corresponding srcML representation.

Original class:

```
package org.apache.lucene;

/**
 * Licensed to the Apache Software Foundation (ASF) under one or more
 * contributor license agreements.  See the NOTICE file distributed with
 * this work for additional information regarding copyright ownership.
 * The ASF licenses this file to You under the Apache License, Version 2.0
 * (the "License"); you may not use this file except in compliance with
 * the License.  You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

/** Lucene's package information, including version. **/
public final class LucenePackage {

  /** Return Lucene's package, including version information. */
  public static Package get() {
    return LucenePackage.class.getPackage();
  }
}
```

srcML representation:

```
<unit xmlns="http://www.sdml.info/srcML/src" language="Java"
filename="LucenePackage.java"><package>package
<name>org</name>.<name>apache</name>.<name>lucene</name>;</package>

<comment type="javadoc">/**
 * Licensed to the Apache Software Foundation (ASF) under one or more
 * contributor license agreements.  See the NOTICE file distributed with
 * this work for additional information regarding copyright ownership.
 * The ASF licenses this file to You under the Apache License, Version 2.0
 * (the "License"); you may not use this file except in compliance with
 * the License.  You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */</comment>

<comment type="javadoc">/** Lucene's package information, including
```

```
version. **/</comment>
<class><specifier>public</specifier> <specifier>final</specifier> class
<name>LucenePackage</name> <block>{

  <comment type="javadoc">/** Return Lucene's package, including version
information. */</comment>
  <function><type><specifier>public</specifier>
<specifier>static</specifier> <name>Package</name></type>
<name>get</name><parameter_list>()</parameter_list> <block>{
    <return>return
<expr><name>LucenePackage</name>.class.<call><name>getPackage</name><argume
nt_list>()</argument_list></call></expr>;</return>
  }</block></function>
}</block></class>
</unit>
```

The *Java Fact Extractor* is mainly based on the Document Object Model (DOM) Java API. This API represents an entire XML document in a tree-like data structure that can be easily manipulated by a Java program.

As soon as the Code Analyser analyses a software release it is able to also run the analysis of internal dependencies as the release contains all the needed information. In particular:

- *Extend* dependencies are captured by analysing the **<extend>** srcML tag, reporting the class (if any) extended by a class under analysis.
- *Implement* dependencies are captured by analysing the **<implement>** srcML tag, reporting the classes (if any) implemented by a class under analysis.
- *Import* dependencies are captured by analysing the **<import>** srcML tag, reporting the classes (if any) imported in a source code file. This type of dependency is computed between files and classes (i.e., classes imported in a file) since the imported classes are declared inside the source code file for all classes contained in it.
- *Calls* between methods/functions are captured by analysing the **<call>** srcML tag, assigned to calls performed by methods.
- *Use-as-type* dependencies are captured by analysing:
    - The method's *return type*, tagged by srcML with the tag **<type><name>**return type**</name></type>**.
    - The method's *parameters type*, reported by srcML inside the tag **<parameter_list>**()**</parameter_list>**.
    - The type of the local variables declared by the method, reported by srcML inside the tag **<decl_stmt>**.
    - The type of the class attributes used by the method, and reported by srcML inside the tag **<expr_stmt>**.
- *Copied* files are detected by exploiting the MD5 (Message Digest algorithm 5) representation and the dependencies of a file. The MD5 algorithm is a widely used cryptographic hash function producing a 128-bit (16-byte) hash value, typically

expressed in text format as a 32 digit hexadecimal number. In few words, the MD5 of a file is a short representation of its textual content. When the Code Analyser analyses a source code file it computes its MD5 (on the entire text excluding white spaces). The MD5 is then compared with those of all files previously analysed. If the MD5 of a file F_i, from the release R_1 which is issued on date D_1 (note that the Code Analyser knows the issue date of each release), is equal to the MD5 of a file F_j, from the release R_0 issued on date D_0 (with D_0 earlier than D_1), than file F_i is likely to be an exact copy of file F_j. As a further check, the Code Analyser also verifies if dependencies of code entities contained in F_i are the same of those of code entities contained in F_j. If it is the case, the Code Analyser keeps track that F_i is a copy of F_j. Note that the current implementation of the Code Analyser maintains all analysed projects inside its database. This is why it is possible to check in it for copies of the same file. In the future, we do not exclude to query the semantic store (via the Querying&Browsing component) to retrieve all files having an MD5 equal to one of interest.

- *Cloned* files (i.e., a file F_i is **very** similar to a file F_j) are captured by using a finger-print based mechanism. In particular, for each parsed Java file the Code Analyser stores a fingerprint representing its structural properties, and in particular it is computed considering:
    - the number of classes it contains;
    - the overall number of methods it contains;
    - the overall number of attributes it contains;
    - the number of classes it imports;
    - the overall number of *if* statements;
    - the overall number of *while* statements;
    - the overall number of *case* statements;
    - the overall number of *for* statements;
    - the overall number of *return* statements;
    - the overall number of exceptions thrown by the methods it contains;
    - the overall number of parameters in the methods it contains.

  If two files have the same fingerprint we assume that the newest one (i.e., the one released more recently) is a clone of the oldest one. Of course such a fingerprint is studied for the Java language, but it could be easily adapted in the future for other programming languages. From a manual analysis conducted on 200 files marked as cloned by the Code Analyser, 93% of them were correct. Note that the clones identified by the Code Analyser are those defined in the literature as Type-2 clones[2]:

[2] ADD REFERENCE

Lexically identical fragments except for variations in identifiers, literals, types, whitespace, layout, and comments. This is why the fingerprint we exploit just takes into account of structural properties of the file without considering textual information.

2.2. External dependencies

Concerning the external dependencies, we can distinguish between two different levels of granularity: dependencies at release levels (i.e., *import-as-jar* and *declare-dependency*) and dependencies at code level (i.e., the same discussed in Section 2.1 for internal dependencies).

The *Java Fact Extractor* identifies *Import-as-jar* dependencies by looking for all jar files contained in the root folder of the release under analysis and tries to match each of those files with one of the project releases contained in its database. Note that also in this case the current version of the Code Analyser looks inside its database for releases to match with the jar files and not inside the semantic store because all analysed projects are currently maintained in the Code Analyser database. However, the Browsing&Querying already provides a service which retrieves the provenance of a given code element.

If the *Java Fact Extractor* is able to identify at least 95% of files contained inside the jar file as belonging to a specific release analysed in the past, it creates a dependency between the release *importing* the jar and the release *contained* in the jar. Otherwise, the dependency is marked as unresolved, and will be analysed again time by time, when more projects will be analysed. Currently, the Code Analyser checks unresolved dependencies each 200 new projects analysed.

The *declare-dependency* dependencies are, instead, retrieved by looking in the folder release for files that explicitly reports inter-project dependencies. As a de facto standard, in the Java projects, these files are mostly of four types: *libraries.properties*, *deps.properties*, the Maven *pom.xml* file, and the Ant build file. Note that the dependency information reported in these files is generally detailed. Indeed, for a declared dependency these files generally report one of the following information:

- Both the name of the project as well as the used release. In this case the declared dependency is retrieved by matching the name of the project and the release present in the dependency file with the name of one of the releases already analysed and present in the database. For example, the following part of a *pom.xml* file highlights a dependency of the release containing it toward the *junit 3.8.1* release:

```
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
</dependency>
```

- A specific package toward which it depends. In this case the declared dependency is retrieved by looking for releases already analysed and present in the database containing such a package. For example, the following part of a *pom.xml* file highlights a dependency of the release containing it toward a package from the Apache cayenne project:

```
<dependency>
<groupId>org.apache.cayenne.unpublished</groupId>
<…>
</dependency>
```

Note that it is possible that the dependencies declared in such files cannot be resolved. In this case, as previously discussed for the *Import-as-jar* dependencies, they are stored as unresolved and re-analysed when more projects are present in the database.

Concerning the dependencies at code level, we keep track for each of them of the entity using (i.e., *source* entity) and of the entity used by the source (i.e., *target* entity). There could be dependencies stored in the database for which we know the *source* entity without having identified the *target* entity, yet. As example, it is possible that during the analysis of a project the *Java Fact Extractor* finds a class implementing an interface contained in an external library (note that the dependency extraction works exactly as for internal dependencies---See section 2.1). In that case, in order to solve the dependency, it is necessary to identify the project which the external library corresponds to. If the release of provenance of the *target* entity is identified, then the dependency is stored. Otherwise, a placeholder for the *target* entity is stored; it does not contain the entity provenance which is considered still unknown but, whenever, more projects are retrieved and analysed the Code Analyser, it will try to solve the value of the provenance.

3. APPROACHES TO EXTRACT DEPENDENCIES FROM WEB-APPLICATIONS COMPONENTS

Table 2 summarizes the dependencies extracted by the Code Analyser when parsing JSP, HTML and JavaScript code.

Name	Granularity	Description
Import	HTML/JSP-to-javascript code	A HTML/JSP page can *import* javascript files
Definition of javascript code	HTML/JSP-to-javascript code	A HTML/JSP page can *define* javascript code
Include	JSP-to-JSP	A *JSP page can include others JSP pages*
Include	JSP-to-HTML	A *JSP page can include others HTML pages*
Use-Tag Libraries	JSP-to-Tag libary	A *JSP page can use Tag Libraries*
Import	JSP-to-java class(es)	A JSP page can *import* java class(es)
Invoke	JS function(s)-to-JS function(s)	A Javascript function can invoke one or more Javascript functions. Note that the function(s) can be local (inner function) or global function(s)
Use-as-type	JS function-to-JS object(s)/JS variable(s)	A Javascript function can declare local variables/parameters/return of a given type T (a type T can be a JS standard type or a custom Object), or use an object of type O, where O is an object defined in the javascript code.
Use-as-type	JS object-to JS function(s)/ JS attribute(s)/JS object(s)	An object can define inner functions, inner attributes (also of type Object)
Copied	file-to-file	A file can be an exact copy of another file
Cloned	file-to-file	A file can be a clone of another file (i.e., not exactly copied, but very similar)

Table 2: Dependencies captured in web-application components by the Code Analyser

3.1. HTML/JSP dependencies

The analysis of the dependencies between web application elements is quite simple when the dependencies are related to HTML pages and JSP pages.

Indeed, to identify in a HTML/JSP page the *import* of Javascript code it is sufficient to match the following pattern:

<script type="text/javascript" src=".…."></script>

Instead, to identify Javascript code *defined* in HTML/JSP pages it is sufficient to match:

<script language="JavaScript">

….

</script>

or

<script type="text/javascript" >

….

</script>

When no type or language is specified the browsers assume that it is JS. Thus, there is another pattern useful to identify Javascript code *defined* in HTML/JSP:

<script>… </script>

When a JSP page *include* others JSP pages we match the following patterns:

<%@ include file="…….jsp" %>

or

<jsp:directive.include file=".jsp" />

Examples of JSPs *included*:

<%@ include file="/includes/top.jsp" %>

<%@ include file="/includes/navbar.jsp" %>

<jsp:directive.include file="./pages/header.jsp" />

However, differently from a HTML page, a JSP page can also *use Tag Libraries* (XML components used within the JSP pages) as in the following example:

<%@ taglib uri="/WEB-INF/tld/struts-html.tld" prefix="html" %>

<jsp.directive.taglib uri="/WEB-INF/tld/struts-nested.tld" prefix="prefix" %>

or

<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" ***%>***

There are two kinds of Tag Libraries:

- Standard Tag Libraries (libraries of custom tags that provide a set of core functionalities which is common to many Web applications.)
- Custom Tag Libraries (libraries created by the developer containing a set of custom tags invoking custom actions in a JavaServer Pages)

A Standard tag library can be identified by matching the following pattern:

***<%@ taglib prefix**="fmt" **uri**="http://..." %>*

While a Custom tag Library can be identified by matching:

***<%@ taglib uri**="/WEB-INF/"…….".**tld" prefix**="...." %>*

Alternatively, the taglib can be included by developers in jar files. In such scenario a deep analysis of the content of jar files it is required. Moreover, very often, it is possible to download the jar files (when the jar files are not available in the local repository) that include the taglib from the web. For example, the Standard Tag libraries of the Apache Software Foundation can be downloaded at the URL "***http://tomcat.apache.org/taglibs/standard/***

Vice versa, the custom tag libraries are located using the CA that has information about all the "*.tld" files contained in the source code repository.

In general we identify classes or methods imported in JSP pages using the following patterns:

A) To import one class:

<%@ page import**="com.xyz.MyClass" **%>

B) To import multiple classes from the com.xyz package:

<%@ page import="**com.xyz." %>***

or

<%@ page language="java" import**="com.xyz.*" **%>

3.2. Javascript dependencies

To parse Javascript code, the Code Analyser exploits a Javascript Fact Extractor (more details are available in deliverable D4.2.1b) that relies on the API of the JavaScript Development Tools (i.e., the JSDT Core). The JSDT Core makes available various APIs implementing a parser of Javascript. The JSDT parser returns an abstract syntax tree (AST) as the result of parsing the code. The AST is the representation of the structure of the program

as a tree of nodes. Specifically, the tree is composed by different kinds of node where each of them corresponds to a syntactic construct of the code, e.g. a function definition f_1, a variable declaration v_1 or an if-statement. The JSDT parser returns the root AST node of the type `JavaScriptUnit`, which is derived from a basic AST node, IASTNode. There are different API in the JSDT Core representing the different Javascript elements: *`FunctionDeclaration`*, `FunctionInvocation ObjectLiteral, VariableDeclarationExpression.` It is possible to navigate the tree by use instances of ASTVisitor, which can be passed to any node via *accept*(ASTVisitor). Hence, using AST Visitors we navigate the nodes in the tree starting from the root node and extract the information for each Javascript element.

For example, if we are interested to navigate the tree and extract the information about the declared functions, it is sufficient to use a visitor that accept a `FunctionDeclaration` node and returns the collection of the node of type `FunctionDeclaration` (more details about the AST Visitors are available in deliverable D4.2.1b). Each generic element in the collection (in this case each element in this collection is of `FunctionDeclaration` type) has several methods to obtain the information about the declared functions:

- the function name: method *getName();*
- the body of the function: *getBody();*
- the set of parameters: parameters().

Hence, with the purpose of extracting information about each Javascript element we use different AST Visitors, one for each Javascript element.

However, as explained in Table 2 a function can *invoke* one or more functions. Note that the function(s) can be local (inner function) or global function(s). Thus, we have to verify if a *declared function* invokes another function(s) and check if that function(s) is a local (inner) function or a global function. In The JSDT Core make possible to identify all of the Javascript dependencies described in Table 2 navigating the information that are available in the tree. Specifically, a generic AST node has two method that allow to have information about her node "parents" and her node "children": (i) `getParent()` and (ii) `getBodyChild()`.The method getParent() allow to have the collection of parent nodes of a node, while the method getBodyChild() allow to have the collection of children nodes of a node.

Clearly, the type of the children nodes depends from the type of the parent node. For example, a node of type *`FunctionDeclaration`* can have a children node of type (i) *`FunctionDeclaration,`* (ii) *`FunctionInvokation,`* (iii) `ObjectLiteral,` (iv) `VariableDeclarationExpression.` Similarly, a node of type `ObjectLiteral` can have a children node of type (i) *`FunctionDeclaration,`* (ii) `VariableDeclarationExpression.` However, in the specific case of object of type ObjectLiteral, there is an alternative method to getBodyChild() named *fields()*. This method return the the list of functions (*`FunctionDeclaration type`*), objects (`ObjectLiteral`) and variabes (`VariableDeclarationExpression`) defined in the body of a Javascript object.

To identify all the Javascript dependencies described in Table 2, it is sufficient to invoke the method "getBodyChild" (alternatively, the method fields() for `ObjectLiteral`) on each node. In particular:

- *Invoke* dependencies between functions are captured by analysing the children nodes of a given function, reporting the children nodes (if any) that are of type `FunctionInvokation`. To distinguish between an inner function and a global function, it is sufficient to verify whether the function is also "declared" locally (if exist a children of type `FunctionDeclaration` that references the same invoked function) then it is an inner function; if it is not declared locally, it is sufficient to check whether it is not declared by any other function.

- *Use-as-type* of functions are captured by analysing:

    - the children nodes of a given function, reporting the children nodes (if any) that are of type `VariableDeclarationExpression`. Such nodes are the local variables of the function.

    - the children nodes of a given function, reporting the children nodes (if any) that are of type `ObjectLiteral`. Such nodes are the local variables of the function.

- *Use-as-type* of objects are captured by analysing:

    - the children nodes of a given object, reporting the children nodes (if any) that are of type `VariableDeclarationExpression`. Such nodes are local variables of the function.

    - the children nodes of a given object, reporting the children nodes (if any) that are of type `ObjectLiteral`. Such nodes are local variables of type `ObjectLiteral`.

    - the children nodes of a given object, reporting the children nodes (if any) that are of type `VariableDeclarationExpression`. Such nodes are local variables of type `VariableDeclarationExpression`.

3.3. Conclusion

This document overviewed the approaches used by the Code Analyser to extract dependencies between source code components. While they represent the foundation on which we are building our dependency analysis, as in all research projects, they might be subject to small changes aimed at further improving the quantity/quality of extracted information. Specifically, we apply two kind of analysis: the analysis of Java code and the analysis of the Web application components (JSP, HTML and JavaScript code). While the analysis of Java source code has been consolidated during the last year, the analysis of Web applications components is more susceptible to changes. Moreover, we do not capture all the kind of dependencies that characterize Javascript code. For example, we do not analyze the inclusions of other JavaScript files yet. In addition, we cannot analyse: (i) dynamic 'injection' in JSP pages (code created in the JSP scriptlets), (ii) Javascript import that cannot be solved whenever the URI are absolute URI.