Inter-Project Dependencies in Java Software Ecosystems

Antonín Procházka[1], Mircea Lungu[2], Karel Richta[3]

[1]Czech Technical University in Prague, [2]University of Bern,[3]Charles University in Prague

Abstract Understanding the legacy of code in a software ecosystem is critical for the organization that is the owner of the ecosystem as well as for individual developers that work on particular systems in the ecosystem. Model driven development (MDD) and model driven architecture (MDA) techniques for describing inter-project dependencies are rarely used or they're not updated by anyone during software evolution process. Describing the dependencies by hand can be painful and error prone process. Another solution is recovering the dependencies using some reverse-engineering process. There are some existing technologies today. One of them is an Ecco model of inter-project dependencies with a set of methods for recovering the dependencies from Smalltalk based software ecosystems developed by Lungu et al. Aim of our research is applying this model with its methods on Java based software ecosystem.

Keywords

Model Driven Development, Software Ecosystems, Inter-Project Dependencies, Java, Reverse Engineering

1 Introduction

Software engineering is concentrated mostly on individual projects nowadays. We've got sophisticated methods and tools for project management, version management, refactoring, testing, deployment and so on. But projects are rarely developed individually. They coexist together, evolve together and benefit from each other. We call these systems of projects *software ecosystems*. Like other terms connected to computers, the term ecosystem comes from biology. In nature we define an ecosystem as *the complex of a community of organisms and its environment functioning as an ecological unit*[1]. In the context of software engineering the ecosystem is defined as *a collection of software projects which are developed and co-evolve in the same environment [2]*. Example of such software ecosystem can be a company developing software, an open-source community or a research group. As every project is located in its version control repository, we define

[1] Webster's Dictionary definition.

J. Pokorný, V. Snášel, K. Richta (Eds.): Dateso 2012, pp. 135–142, ISBN 978-80-7378-171-2.

a *super-repositories* as *a collection of all the version. control repositories for multiple software projects [3].*

Looking at the software from a point of view of software ecosystems uncovers wide range of important information which help managers to manage their teams and projects and also help individual developers to better understand their work. Analysis of software at the abstraction level of software ecosystems can be either focused on the projects or on the developers in the ecosystem. Our work is currently focused on projects and their relationships inside a software ecosystem. We extend previous work of Lungu et al. [4] focused on recovering inter-project dependencies in Smalltalk ecosystems. In their work they argued for importance of raising abstraction of view on software products from individual projects to whole software ecosystems. They presented several viewpoints at this abstraction level including the inter-project dependency viewpoint. Each viewpoint, including this one, provides two areas of research. One is own visualization Having an interesting information is not enough - we also need to know how to present it to the user. The second area is information retrieval. Before we can present some information, we need to get it by some technique from some source. At first we focus on inter-project information retrieval from java based software ecosystems.

Structure of this paper is following: In section 2 we describe a model used to store retrieved information. Section 3 summarizes information specific about inter-project dependencies specific for Java base software ecosystems. Evaluation of different methods for dependency information retrieval is described in section 4. In section 5 we discuss contribution of this work and outline our further research to be performed on this topic.

2 Ecco model

Lungu et. al presented in their work a lightweight model describing inter-project dependencies called *Ecco*. They defined the model and filled it up with information about inter-project dependencies present in selected Smalltalk based software ecosystems.

The Ecco model consist of four main elements.

Ecosystem. In relation to the Ecco model the ecosystem means a set of software projects and dependencies between them.

Project. Every software ecosystem consists of one or more projects. Modules of each project call some methods and define another. A project can call a method which is defined in another project. Methods like this are called requirements.

Dependency. When one project require some method and another defines it, we call this relationship a dependency. The dependency consists of a client project, which requires the methods, and of a provider project, which provides the required methods. The methods making the dependency between two projects are called elements of dependency.

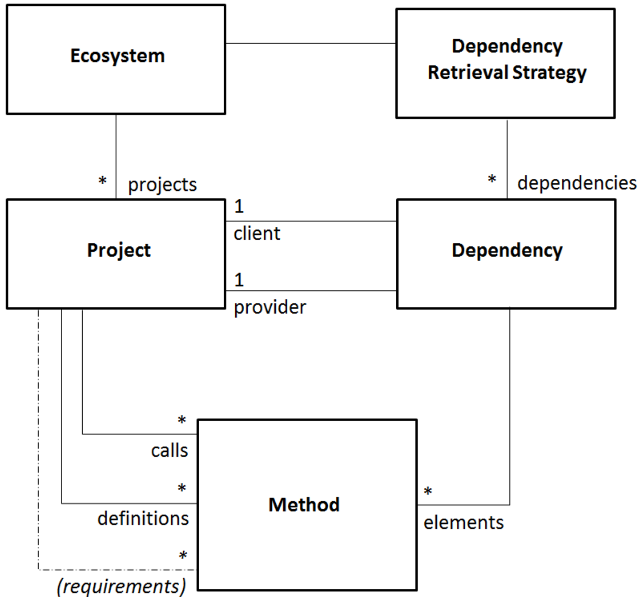

Fig. 1. Ecco is a very lightweight model aimed at extracting dependencies between projects in an ecosystem [4]

Dependency Extraction Strategy. There are several existing techniques for gathering information about inter-project dependencies and others can be defined in future. Techniques like this are called dependency extraction strategies. We include them in the model to be able to compare them during our research process.

3 Java Dependencies

In general we have two types of dependency extraction strategies. The first type reuses information existing explicitly in software super-repositories. The disadvantages of such sources are limited availability in different ecosystems and error-prone and time-wasting maintenance. On the other hand, this source is very important during research because it tells us what results to expect during evolution of the second type of dependency extraction strategies.

The second type is base on reverse-engineering of source code. In contrast to the first type, this one can be used on any kind of super-repository and doesn't need any maintenance at all. However it is harder to retrieve the information this way.

3.1 Project Object Model

If we'd like to find some reverse-engineering strategy for recovering inter-project dependencies in Java based software ecosystems, we first need to find proper source of data. We need to have a super-repository which will provide us both the explicit data and source code which we'll reverse-engineer.

Looking for such super-repository we found Apache Maven best suits our needs. Maven is a project-centric tool for software development. Its data structures contain different information about each project enabling to manage project's build, reporting and documentation. Whole Maven stands on technology called Project Object Model (POM) [1]. Every project has its own so-called POM-file, which is an XML file containing all the information relevant to this project like the developers working on it, the path of its sources, required binaries, the builder, the documentation manager, the bug tracking system and much more. It includes the explicit information about the inter-project dependencies. This information has to be compounded from four inter-project relationships described in the POM: dependencies, exclusions, inheritance and aggregation. There's also a file called Super-POM which defines value common for all project in the Maven repository unless they are redefined. A simple POM with one dependency can look like this:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>cz.cvut.fit.swing</groupId>
  <artifactId>my-project</artifactId>
  <version>1.0</version>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.0</version>
      <type>jar</type>
      <scope>test</scope>
      <optional>true</optional>
    </dependency>
  </dependencies>
</project>
```

Dependencies. If one project depends directly on another then the information is described in a dependencies section. This section is located in POM file of the project which requires these dependencies - the Client Project from the Ecco's point of view. These dependencies can also be transitive. Transitive dependency means that if a client project A requires a project B which requires a provider project C, C becomes common requirement for both A and B. Dependencies here are divided into 5 scopes:

A Compile Scope is a default scope representing group of regular projects which are available with their source code and are necessary for successful build of a Client Project. The Compile Scope dependencies *are transitive*.

A Provided Scope represents a group of precompiled projects expected to be given at compile time by Software Development Kit (SDK), container or another way. The Provided Scope dependencies *are not transitive*.

A Runtime Scope is much like the Provided Scope but represents projects expected to be given at runtime. The Runtime Scope dependencies *are not transitive* as well.

A Test Scope is like the Compile Scope but represents projects needed for testing purposes. The Test Scope dependencies *are transitive* as well as the Runtime Scope.

A System Scope is similar to the Provided Scope but requires a developer to provide its dependencies explicitly. The System Scope dependencies *are not transitive* as well as the Provided Scope.

As we'll be examining only projects contained in a given ecosystem, we are interested only in the Compile Scope dependencies. Possibly we can be also interested in the Test Scope dependencies if we'll extend our analysis to project's used for testing purposes.

Exclusions. Transitive dependencies can produce unwanted behavior. If a developer needs to exclude some project from the dependency list she includes it into the exclusions section of the dependency which causes the problem. The meaning of the exclusions during populating the Ecco model is obvious. We should respect these exclusions and throw away dependencies excluded by them.

Inheritance. The Project Object Model brings a feature which enables us to make an inheritance tree of projects. From the view of POM this means that if we define something in an ancestor project's POM file, all its child project inherit these definitions unless they are redefined in a child project's POM files. There are two points important for us. First, the inheritance relationship itself represents a dependency and we have to to think about it this way. Second, dependencies of ancestor client projects become dependencies of child client projects since these two projects are in inheritance relationship.

Aggregation. If a project is made of a modules, Maven thinks about the modules as about separated projects which are aggregated into another project called multi-module project. This relationship is described in the multi-module project's POM file in a modules section. As the modules are expected to belong to the same group as their multi-module project, they are defined only by their project names. From our point of view, the aggregation relationship represents another way to express the inter-project dependencies between the modules and the multi-module project.

3.2 Java Bytecode

When we think about a reverse-engineering of a Java software, we are not limited only to a Java language. We can think of any language which can be compiled to a Java Bytecode. The original information can be simply disassembled from the byte-code [6]. Consider this simple class definition written in the Java language:

```
import java.awt.*;
import java.applet.*;

public class DocFooter extends Applet {
  String date;
  String email;

  public void init() {
    resize(500,100);
    date = getParameter("LAST_UPDATED");
    email = getParameter("EMAIL");
  }

  public void paint(Graphics g) {
    g.drawString(date + " by ",100, 15);
    g.drawString(email,290,15);
  }
}
```

If we call javap DocFooter to disassemble a DocFooter.class, we get this output:

```
Compiled from DocFooter.java
public class DocFooter
    extends java.applet.Applet {
  java.lang.String date;
  java.lang.String email;
  public DocFooter();
  public void init();
  public void paint(java.awt.Graphics);
}
```

Passing some arguments will give us also a disassembly of a behavior, but this interface declaration is all what we need. We've got fully qualified name of every class and method used in the compiled code.

This is how our reverse-engineering dependency extraction strategies will look like. At first we take a Java Archive. Every java project is distributed as a Java Archive. The archive is a regular compressed package of data containing a Class Files. Every Class File contains a byte-code of one Java class. We open the archive, disassemble every class file and see which methods are called and which are defined. We fill this information into the Ecco model. Information

gathered this way needs some more processing before we'll get reliable result. This post-processing is topic of our further research.

4 Evaluation of Results

To let us compare different inter-project dependency retrieval techniques we need to have a measuring method to let us assign a value to each technique. For this purpose we'll use well-known information retrieval metrics - *a precision*, *a recall* and *an F-measure* [5] adopted for our case by Lungu et al. [4]. To use them we first need a "golden standard" or an oracle. This is the information we retrieve from Maven's POM. Thanks to this information we are able to distinguish *a Relevant* dependencies which are present in the oracle and *a Nonrelevant* which are not present in the oracle. Besides this we can divide the dependencies to those which *have* or *have not been* retrieved by a concrete reverse-engineering technique. In common we get four different statistical sets of dependencies which can be seen in table 1.

Table 1. Statistical sets of retrieved inter-project dependencies [5]

	Relevant ($TP \cup FN$)	Nonrelevant ($FP \cup TN$)
Retrieved ($TP \cup FP$)	True Positives (TP)	False Positives (FP)
Not Retrieved ($FN \cup TN$)	False Negatives (FN)	True Negatives (TN)

The metrics are then defined as follows. The Precision (P) is a fraction of retrieved dependencies that are relevant. The Recall (R) is a fraction of relevant documents that are retrieved. The F-measure (F) is the weighted harmonic mean of precision and recall. The F-measure represents a single measure that trades off the precision versus the recall and thus indicates an overall accuracy of the measured technique.

$$P = \frac{|TP|}{|TP \cup FP|} \qquad R = \frac{|TP|}{|TP \cup FN|} \qquad F_1 = \frac{2PR}{P+R}$$

We use a default balance F-measure (F_1) which equally weights the precision and the recall because we don't want to emphasize the recall nor the precision.

During evaluation of our reverse-engineering techniques we'll calculate these values for each technique and compare them. This comparison will give us the required information about the technique's effectivity.

5 Conclusion

The information summarized in this paper gives us excellent base for our further research aimed on different reverse-engineering techniques for retrieval of inter-project dependencies in the Java based software ecosystems. We have an excellent source of data which will help us with a development of the techniques. Using the explicitly given information about the dependencies and using the mentioned metrics we are able to compare every techniques and tell which one better suits our needs. We found a way which lets us to retrieve the dependencies from any language which can be compiled to the Java byte-code. In connection with the work done by Lungu et al. on the Smalltalk based software ecosystem we'll be also able to summarize differences between a dependency retrieval from statically and dynamically typed languages.

6 Acknowledgments

We would like to thank for financial support of Student Grant Competition of CTU in Prague, grant number SGS12/093/OHK3/1T/18.

References

1. Apache. Maven project, 2002.
2. Lungu, M. *Reverse Engineering Software Ecosystems.* PhD thesis, University of Lugano, 2009.
3. Lungu, M., Lanza, M., Girba, T., and Heeck, R. Reverse engineering super-repositories. In *Proceedings of the 14th Working Conference on Reverse Engineering* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 120–129.
4. Lungu, M., Robbes, R., and Lanza, M. Recovering inter-project dependencies in software ecosystems. In *Proceedings of the IEEE/ACM international conference on Automated software engineering* (New York, NY, USA, 2010), ASE '10, ACM, pp. 309–312. ACM ID: 1859058.
5. Manning, C., Raghavan, P., and Schtze, H. *Introduction to Information Retrieval.* Cambridge University Press New York, NY, USA, 2008.
6. Oracle. Java se documentation, February 2010.