Prediction of code evolution

Viktor Karabut

November 14, 2014

Contents

Abstract

Goal of this work is develop computer system that predicts possible improvements for specific Java code fragment. System parses Java method into ACT (approximate code trees) fingerprints and searches for similar code from database of fingerprinted, versioned and indexed OSS (open source software) projects. During this work I plan to obtains OSS code from Maven Central and construct chains of versioned ACT fingerprints. After that develop search engine for finding similar code in fingerprint database.

Introduction

Currently there is a lot of code written in Java. Often developers waste time on writing code which is already exists in another project or library. Our goal is develop a tool, which would hint developers, where is already exist a simillar code and how to improve it's quality. Often code reuse is limited by legal issues, such as patents, copyrights. Hovewer there is a wide variety of an open source projects, from where code can be used as an inspiration or even be directly copied.

In the first chapter we consider questions about code similarity. To detect similar code fragmenst we will parse code into Abstract Syntax Trees (AST). After that we can compute edit distance between AST's. Edit distance shows how many nodes from original AST need to remove or add to get target AST. This is analogue of Levenshteins distance for trees.

In second chapter we will consider search engine for code fragments. AST each-to-each comparison complexity is $O(n^2 \cdot m^2)$ where n is number of trees and m is size of each treee. For large code fragment database speed of quadratic alghorithms is unacceptable. To solve this issue we use a hashing mechanism. Unfortunately hashing of raw ASTs cannot help in finding similar code fragment. For this we introduce Approximate Code Trees (ACT) fingerprints. ACT is generalized version of AST, where some nodes replaced by patterns. Every ACT represent wide class of ASTs. Now we can in constant time find matching ACT and get some relatively small set of matching AST from where we can find more similar code fragments using a slow quadratic alghorithms.

Finally in third chapter we describe experiments with finding code fragments in open source projects fetched from Maven Central database.

1 Similar code detection

In this work we use pure syntactic methods to determine 'equivalent' or 'similar' code. Simplest possible way to do this would be line-by-line comparison of source lines. However code can differ by an indentation, comments, variable names and etc. Closer to full semantics but sill practical possibility would be to compare program representation in which control and data flows are explicit.

At first step of similar code detection we parse code into Abstract Syntax trees. Rather than comparing trees for exact equality, we compare by similarity scoring function[1]:

$Similarity = \frac{2 \cdot S}{2 \cdot S + L + R}$

S is a number of shared nodes. L is a number of different nodes in left subtree, and R - in right subtree.

2 AST search engine

2.1 Hashing of code fragments

The one to one AST comparison method is slow and isn't applicable in practice. We need hashing mechanism for AST where similar AST have same hash codes. For this we introduce our Approximate Code Tree (ACT) structure. ACT is a method fingerprint, which we use for our hashing function. We produce ACT from single method AST by throwing away irrelevant information, generalizing types, normalizing expressions and replacing some construction with wildcards. We work at method level, so as a disvantage we cannot handle refactoring such as a method inlining or method extraction.

First in our hashing function we can ignore variable names. The primitive types we preserve as is. In Java there are 8 primitive data types which we should preserve in our hashing mechanism: boolean type, Unicode character type, 8-, 16-, 32-, 64-bit signed integers, IEEE 754 double- and single-precision floating numbers. As an option we can generalize all integer and floating types which are semantically similar.

Class type handling is different. To properly handle the class types we need to have information about full project class hierarchy. It means that we should parse not only whole project but all project dependencies. Moreover similar classes in different project can have different names, so considering class types would significantly limit variety of similar code. By considered above reason we throw away object information in our search engine. But there is one exclusion: we preserve Java standard library classes, such as 'java.lang.String' which are the same for every project.

Some expressions is normalized, for example chain of assignments, if assignment are not related with each other we can safely reorder them. We normalize assignment chain by reordering them in alphabetical order. Also we normalize constant-variable boolean comparisons. In normalized comparison variable always on left side.

Introduced in Java 6 foreach cycle can be replaced with old for cycle. Lambda expressions in Java introduced only in Java 8. In Java lambdas semantically are equal to callback-classes with single method. Even developer can use lambda in every place where callback-method is accepted. So in our search engine we do not distinguish lambdas from callback-classes.

Some kinds of expression are removed while ACT processing. This is a simple log method calls, if statements which checks does logging is enabled or nor, assertions and class casts. To detect logging we just have base of known logging frameworks. Most of Java projects use SL4J, java.logging, commons-logging etc.

2.2 Database and Maven

Apache Maven is a build automation tool used primarily for Java projects. Maven addresses two aspects of building software: First, it describes how software is built, and second, it describes its dependencies. Contrary to preceding tools like Apache Ant it uses conventions for the build procedure, and only exceptions need to be written down. An XML file describes the software project being built, its dependencies on other external modules and components, the build order, directories, and required plug-ins. Maven dynamically downloads Java libraries and Maven plug-ins from one or more repositories such as the Maven Central Repository, and stores them in a local cache.

Build artifacts in maven repository is versioned and often provided with source code. It allows as to trace evolution of project. In this paper we user artifacts from Maven Central Repository. Maven Central Repository is the default repository for Apache Maven and it serves only Open Source projects. According Maven Central Repository statistics in November 2014 there were over 800 thousands artifacts.

For accessing Maven repository there is well documented Maven Plugin API for fetching and managing artifacts.

3 Experiments

About experiments, benchmarks, examples etc.

4 Conclusion and Future Work

How does class file look like? Check out JVMS [2].

References

[1] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 368–377. IEEE Computer Society Press, 1998.

[2] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java^{TM} Virtual Machine Specification*. Oracle America, Inc., java se 7 edition edition, February 2012.

[3] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

Appendices

Appendix A: A very important messy details

Text of the appendix A.