Object-oriented metrics for Java programs

Project in Computer Science – EDAN70
January 22, 2015

Olle Tervalampi-Olsson
D11, Lund Institute of Technology, Sweden
dat11ote@student.lu.se

Marcus Lacerda
D11, Lund Institute of Technology, Sweden
dat11mla@student.lu.se

Abstract

A key aspect of software metrics, are dependencies. That is, how different classes and packages make use of each other to construct a program. Object-oriented metrics allow these dependencies to be analyzed and give an overview of how stable and healthy the program is for some aspects. We introduce a new tool that is an extension of the JastAddJ compiler and calculates these metrics on java source code, similar in functionality to *JDepend* and some functions that exist in *CodePro Analytix*.

Keywords Metrics, JastAddJ, CodePro, JDepend,

1. Introduction

The object-oriented software package metrics, as described by Martin in his 1994 paper *OO Design Quality Metrics*[4], are today used in both industry and academia to evaluate some aspects of an object-oriented project. Doing this requires calculating a number of metrics, described later in this paper, where the most important are *efferent* and *afferent* couplings.

Computing these metrics by hand is both time-consuming and error-prone making it advantageous to use a tool. The tool used should, of course, be efficient and error-free. Another desirable quality is to distinguish between standard and user-defined classes, something that the current metric-tools cannot. To know that a package in your project is dependent on e.g. the Java class ArrayList might not be very relevant to the stability of your project.

In this paper we give an overview of Martins metrics, a short introduction to JastAdd and the JastAddJ compiler and their functionality, as well as describing the implementation of a tool that calculates the metrics in JastAdd and give a review of the methods used to solve the problems and analyze the source code. Finally, we present an evaluation of the performance and functionality of the tool as compared to JDepend and the relevant parts of CodePro.

2. Problem description

In order to calculate the different metrics we need to determine the dependencies that exist between Java packages. To do this we need to examine the dependencies that classes have towards each other, specifically classes that are not in the same package. These dependencies are called *couplings*.

While coupling are seemingly easy to define and intuitively understand, it is not immediately clear what kind of code actually creates a coupling. Here, we first present the different types of couplings and the metrics that can be calculated from couplings, and then discuss what to actually include as a coupling.

2.1 Package metrics

What follows is a brief description of the object oriented design quality metrics mentioned in [4]. The paper defines the metrics in terms of *categories* which in Java terms correspond to *packages* which is the term that will be used throughout this paper.

Afferent couplings, C_a, for a package is the number of classes outside a package that depend on classes within this package.

Efferent couplings, C_e, for a package is the number of classes inside a package dependent on classes outside of this package.

Instability, I is a quotient calculated as $C_e/(C_e + C_a)$. This measure has a range from 0 to 1 where $I = 0$ is maximum stability and $I = 1$ is minimum stability.

Abstractness, A, is the ratio between the number of abstract classes through the number of classes in a package.

Furthermore, a package is said to be *balanced* if the sum of its instability and abstractness is 1 ($A + I = 1$). This definition gives us the the *main sequence*, a line from the point $A = 1, I = 0$ to $A = 0, I = 1$. The main sequence allows us to define the last metric.

Distance, D, is the distance from the packages A and I to the main sequence calculated $|(A + I - 1)/2|$. A value close to zero is preferred.

So what kind of code should generate a coupling? The basis for our implementation is that if a class uses another class in any way, for example by accessing their methods or public member variables, that class which makes the call has an efferent coupling to the callee. At the same time, the callee has an afferent coupling to the caller.

As we want any usage of a method to constitute a coupling, simply typing

```
UserCreatedObject uco;
```

will create a coupling. This has the small drawback of potentially creating an extra coupling to an unused class unless the programmer is careful and makes sure to make use of the object, this is something which should be caught at other stages in the process.

If a package implements a class with a public static member variable, and it is accessed with

```
int foo = UserCreatePackage.UserCreatedClass.memberA;
```

this will also create a coupling. Using string literals or primitive java data types will not create a coupling to java.lang. This is done since in order to ensure the calculated metrics (instability and distance from the main sequence) remain as close to the *intent* of the metric as possible. A program will not become more or less stable because it uses integers.

A third note on couplings is that we do not implement them as a transitive relation. This is again a choice made to stay as close to what the metrics represent as possible. If a package Foo has an efferent coupling to a package Bar, which in turn has an efferent coupling to a package Qux, we do not know if any of Foos usage of Bar will actually change or in any way affect the relationship between Bar and Qux.[6]

3. JastAdd and JastAddJ

Our tool is constructed as an extension to the JastAddJ compiler[2]. JastAddJ is a modular Java compiler that is implemented in JastAdd, a declarative, aspect-oriented language. JastAdd is designed to support compiler implementation and related extensions such as analysis tools[5].

To add the tool to the JastAddJ compiler we extend the frontend of the JastAddJ compiler. As previously stated, we want to perform the computations on the source code and not on byte code and it is therefore necessary to add attributes and equations to the nodes in the abstract syntax tree that JastAddJ generates, before the generation of bytecode.

When considering how to find different couplings between classes in different packages, things get a bit complicated. In the AST that JastAddJ constructs, the root node is a program and a program node in turn consists of zero or more *compilation units*. These compilation units correspond to a .java file, and as such we see that there is no handling of packages at the AST-level (Each compilation unit has a *packageDecl* attribute that is a string, but there is no representation of packages in the AST).

As classes can be nested inside a .java file it is incorrect to think of a compilation unit as representing a single java class. Instead, a compilation unit has, among other things, a list of *TypeDecl* nodes. Theses nodes can then be a *ReferenceType*, which in turn can be a *ClassDecl* or an *InterfaceDecl*.

Each call to a method is conveniently found inside a node MethodAccess in the tree that represents the call. These nodes have a *decl* attribute that gives us a corresponding definition of the call as a MethodDecl node.

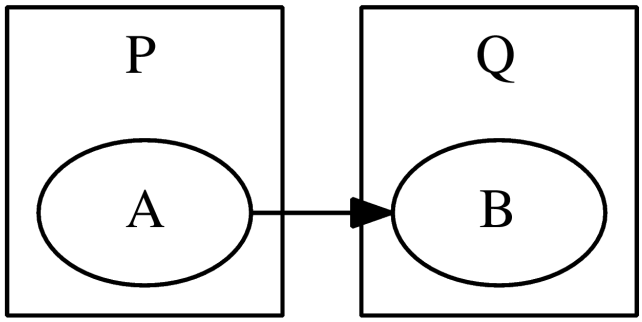

Figure 1. A graph showing the dependencies between two classes, A and B, where A uses B.

4. Implementation

What we will be primarily interested in is to create two different JastAdd collection attributes for each package in a Java project. One to keep track of the efferent couplings and the other to track afferent couplings. Once this is done, calculating the instability, abstractness and distance becomes quite easy. Thus, when a class A in a package P creates an object of type B that is in a package Q, an efferent coupling from P.A to Q.B is added to P, and an afferent coupling is added to Q.

One could imagine a coupling for every import, however, a coupling should only be created if the value is later used. For example, if a package P has five classes, and a class outside of it does `import P.*`, we do not want to create five couplings unless all five classes in P are actually used.

The process is further complicated by the possible use of static calls, for example it is possible to do `foo = P.bar.baz()` without importing P or `bar`, provided `bar` and `baz` are declared to be static. These method calls also need to be found to be a coupling and added to the collection attributes if the classes accessed are outside the current package scope.

For implementing the tool we needed to find a good way to solve several problems, including:

- The ability to find the compilation unit for MethodAccess and MethodDecl objects
- The ability to find the class or interface a MethodAccess or MethodDecl is in.
- A way to iterate through all compilation units in a package.

The first two problems are rather trivially solved. We added the preexisting inherited attribute compilationUnit() to the MethodAccess and MethodDecl nodes. These nodes already had an attribute *HostType* corresponding to the enclosing class or interface. We then created two sets of strings for each compilation unit using the JastAdd collection attribute. When a method access is found, we look at the compilation unit for the MethodAccess and the corresponding MethodDecl. If these compilation units have different packages we have found a coupling. We then add a string containing the necessary coupling info to the efferent set for the caller, and the afferent set for the callee.

In order to present this information to the user in a nice way, we would like to present the couplings on a package basis. In order to do this we iterate through all compilation units, adding the couplings present to a HashMap, where the key is the name of the package and the value is a set containing all couplings for the package. Then we iterate through this map and present the couplings for that package, as well as additional info on stability and abstractness of the package.

During the implementation and design we were faced with a number of important design decisions. For instance, determining whether interfaces should contribute to a packages abstractness or not. We decided that interfaces should be counted in the same way as abstract classes when computing abstractness, since an interface is basically an abstract class with only abstract methods. The main difference between the two is that multiple interfaces can be implemented by the same class but only one class can be extended. Furthermore, no logic can be held in the methods[1] of an interface which is not the case for abstract methods. The only relevant aspect in terms of abstractness is that both interfaces and abstract classes need other classes that implement and extend them.[4] Thus they should both be counted equally. Also, both CodePro[7] and JDepend[1] include interfaces when calculating abstractness.

We also had to make a decision regarding how to distinguish between couplings between classes created by the program and couplings from the program to standard libraries. Here, we chose to include both types of dependencies but separate them. This so that user-defined package couplings could be easily determined, while at the same time not losing information regarding standard packages which other pre-existing tools provide. Another good aspect of including the user-defined packages is that the tool becomes more easy to use compared to other tools, e.g. for testing purposes.

5. Testing

For testing, we set up an automated framework that looks at a directory containing several test directories, where each directory is a specific test case. For each test case, all the java files inside the directory are compiled with the JastAddJ compiler with our tool extension, and the output is written to an outfile. We then compare this output to a manually written file containing the expected output for each test case. This works well for smaller cases when the metrics are easily calculated by hand.

An example of the smaller test cases we used is:

- Two classes in different packages calling each other.
- Three classes in different packages calling each other in a cycle.
- Making a call to a method in another package from a nested class, ensuring the tool correctly reports a coupling from the nested class.
- Testing to ensure that interfaces are correctly counted as abstract classes.
- Classes that make use of standard java library classes and ensuring the afferent couplings gets added to the packages for these classes.
- Testing the option to not count standard library functions when calculating metrics.
- Ensuring all interfaces gets added as couplings for classes that implement several interfaces from different packages.
- Checking the instability output when a package has no efferent or afferent couplings (Ensuring the tool doesn't crash because of a division by zero)

For the larger test cases, we use the same approach but instead of calculating the expected output by hand it is instead manually compared with the output of two other available tools, JDepend and CopePro and determined to be correct.

6. Evaluation

Once our tool passed the small and large test cases, we started comparing the output of the tool with the output from CodePro and JDepend, and measuring the speed of the tool as compared with the speed of JDepend (CodePro can only be run via Eclipse, and we have not found a suitable way to measure the speed of the analysis).

All of the tools require some degree of compilation before the analysis is run. JDepend works on .class files and as such needs all java files to be compiled before analysis, our tool takes .java files as input and proceeds to carry out calculations after the analysis phase of compiling. The tool we built skips the code generation step completely, saving time. It would be possible to provide it as an optional extension to the full JastAddJ compiler to allow for complete compilation while optionally running the analysis.

6.1 Functionality

Our tool, CodePro and JDepend have all taken different approaches to the software package metrics, and we will list the primary differences and provide brief discussion of them and why they differ.

The first thing we need to look at is what constitutes a coupling. JDepend considers every use of standard java objects and primitive types to be a coupling. For example, the following would be considered having an efferent coupling to java.lang as it uses ints.

```
public class Foo {
    public int bar() {
        return 17;
    }
}
```

CodePro takes another approach in which ints and other primitive data types do not constitute a coupling, however, creating a String would be considered one. While string is not strictly defined as a primitive data type, it is very widely used and has lots of special support from the java language. For this reason, our tool also does not consider the creation or use of a String object to be a coupling. Our decision is based on the opinion that knowing a program uses Strings or ints doesn't say enough about the software package to

[1] Java 8 default methods

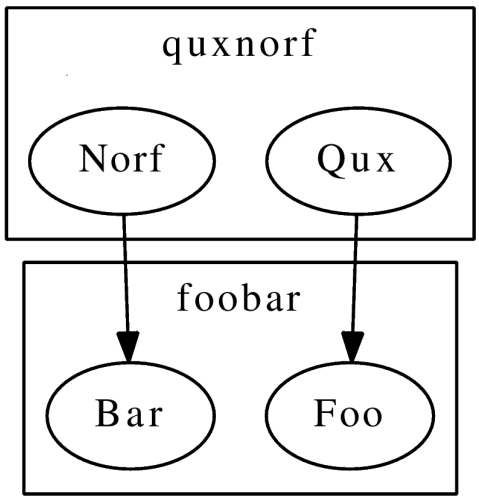

Figure 2. A graph showing the dependencies between four classes, two in each package.

warrant inclusion in the analysis, as that would only ensure almost every class has a coupling to `java.lang`, but not be very interesting. Our tool primarily analyses method calls and use of objects, and invoking `String.substring()` or similar methods would result in a coupling to `java.lang`, but simply creating or using a String would not. However, when a program creates a user-defined object, a coupling to that package is added, as the constructors for those classes might affect the state of the program.

The tools also differ in presenting the dependencies between classes and packages. For example, consider the diagram in figure 2. As we can see, we have two different classes in one package using two different classes in another package. JDepend and CodePro will treat both these uses as one single coupling from one package to the other, whereas our tool reports that there exists two couplings from one package to another. We feel this allows the stability measure to be of greater use. If we consider the stability measurement as defined by Martin [4], stability is a measurement of how likely a class is to change, and one reason a class is unlikely to change is that it is depended on by many other entities. Thus, it makes more sense that a package that is being depended on by ten different classes in two different packages should be more stable than a package that is depended on by three different classes in three different packages.

JDepend and CodePro also only reports at a package level, telling us that one package is depended on by another package but not providing any information about the relation between the classes inside of the packages. This could also lead to confusion. For example, consider the case with a package with an instability measure that is close to zero. If we were following object oriented practices, we would try to come up with a lot of other solutions before we attempted to change any one class inside that package. However, suppose there is a class inside of the package that is not being used by any class outside the package. We see that we are free to change the class (and probably move it to another package). However, with CodePro or JDepend, these kinds of situations will not show up, while our tool provides information about which classes inside a package that are being depended on.

Both JDepend and CodePro offer more functionality than our tool. CodePro has a lot of functions for generating a lot of things apart from SPM, and JDepend has several built-in features that our tool does not, for example it allows the user to ignore arbitrary packages in the analysis, while our program only provides functionality for excluding java standard packages. On the other hand, no functionality they offer provides a way to change their way of generating couplings to ours.

6.2 Performance

In order to get accurate results, we adapted a methodology based on [3]. We will not provide the statistical background here but instead provide some reasons for our choice of methodology. We choose to measure startup performance as this would be the closest to the real world usage of ours and other tools as you only need to run them once on a project to collect the desired metrics. We ran the tests 50 times in order to get a large enough sample size that the mean value obtained approximates a normal distribution.

For running the tests we set up a script that would first run the test once and discard the result in order to minimize variance because of objects and code being loaded into memory.

When running the tests we used the `time` command, which produces an output consisting of three lines, real time, user time and sys time. Real tells us how much time elapsed, that is, how many seconds did it take from the command being put in to the command exiting. This is not very useful for our purposes as it includes time the thread spent blocked, interrupted and time-slices used by other processes. User is the amount of CPU time spent outside the kernel but within the called process, giving us actual CPU time used by the process. The last value, sys, is the amount of CPU time spent in system calls from the process within the kernel. This is still a part of CPU time used by the process (it doesn't include time spent in the kernel from calls outside the process). Thus, the total time used by each call is the sys+real value, which is what we later use to compute mean values and confidence intervals.

As previously noted JDepend needs to operate on class files, and for this reason we included the time to compile the program using javac in our results as a separate measurement which more accurately shows the actual time to get the measurements. The commands used to run the the tests were:

```
/usr/bin/time -p javac @filelist
/usr/bin/time -p java jdepend.textui.JDepend
tests/junit4.0/
```

for JDepend with javac,

```
/usr/bin/time -p java jdepend.textui.JDepend
%testdir%/
```

for just JDepend and

```
/usr/bin/time -p java -jar software-metrics.jar
-nostandard %testdir%/*
```

	JDep	JDep & javac	Our Tool
SmallTest			
MeanUser	0.10	0.84	0.65
MinUser	0.08	0.79	0.63
MaxUser	0.11	0.87	0.67
StddevUser	0.007	0.015	0.009
IntervalUser	0.081, 0.111	0.81, 0.87	0.635, 0.671
MeanTot	0.11	0.88	0.67
MinTot	0.08	0.82	0.64
MaxTot	0.12	0.91	0.69
StddevTot	0.008	0.016	0.011
IntervalTot	0.09, 0.12	0.85, 0.91	0.64, 0.69
LargeTest			
MeanUser	0.56	4.01	3.46
MinUser	0.49	3.97	3.30
MaxUser	0.61	4.27	3.79
StddevUser	0.03	0.06	0.13
IntervalUser	0.51, 0.61	3.96, 4.21	3.21, 3.72
MeanTot	0.68	4.34	3.56
MinTot	0.58	4.21	3.39
MaxTot	0.74	4.53	3.89
StddevTot	0.04	0.07	0.13
IntervalTot	0.61, 0.75	4.21, 4.47	3.30, 3.81

Table 1. Results of running our performance measurements. User corresponds to the "User" value given by the time command, and Tot is the User value plus the "Sys" value

for our own tool.

Not shown here are the printed results from the removal of the generated class files after each run of the javac/JDepend test in order to provide the same input state each time. Once we had the results we calculated the mean, min and max values as well as the standard deviation and a 95% confidence interval for each test.

As can be seen from table 1, none of the confidence intervals overlap and we can be reasonably (95%!) sure that there is a difference between all running times. The small suite consists of ~200 LOC (Measured using *cloc*). For the larger test suite we analyzed JUnit 4.0, which has 7400 lines of Java code.

The times included in the table are first the reported user time, and then the total (user + sys) time reported. We make this distinction since running javac and then JDepend means having to write .class files to disk in the javac stage and then reading them again when running JDepend, which could potentially skew the results. We still include the total sum in order to report on how long time actual use takes (since if one of the tools for some reason write a lot of data then that should still be included in the running time). As can be seen, the same general pattern emerges regardless of which one of the results we look at, which is that just running JDepend on precompiled files is a lot faster than running our tool and running JDepend with javac first, while our tool is a bit faster than running JDepend with Javac. We interpret these results to mean our tool more closely corresponds to real world usage. When measuring results with JDepend, it is necessary to have a compiled program, and once the metrics are calculated they will not actually change until the program is compiled again, meaning further running of JDepend, while fast, will not provide new results.

6.3 Other considerations

The handwritten parts of the tool is relatively small and efficient, the tool itself is at 80 lines, the printing and analyzing of results is 180 lines and the graph generation 160 lines. The final size of the generated jar file clocks in at 1.5 megabytes since it includes a complete compiler front-end (The generated java code from the compiler is 94K lines).

Our tool is very easy to use. It is contained in a jar file and the only input required to run it is the .java files of the project one wants to analyze. The jar file is approximately 1.5 megabytes, which might seem big for such a small program, but it includes a lot of generated code necessary for constructing the AST. As the tool compiles the program in order to get a functional syntax tree, all .java files necessary for running the program needs to be provided. The compiler will warn if something goes wrong, but still perform the analysis. It is not possible to use it with eclipse or other IDEs since it is built as a modification of a custom compiler.

7. Conclusion

We have created a tool that can calculate object-oriented metrics for java projects and which displays the data both as text and as graphs. It can compute afferent- and efferent couplings, abstractness and instability as well as the distance from the main sequence. Our tool can distinguish between packages from the Java standard library and user-defined packages.

In comparison to CodePro and JDepend our tool has less functionality. Although, since our scope was only to create a good tool for specifically OOM it is not that relevant to compare other things. Regarding the OOM feature of CodePro and JDepend our tool actually competes quite well. The information produced by our tool is more detailed and due to it being a command line tool it can easily be modified by scripts.

A drawback might be that it cannot be directly included in integrated development environments since it uses a custom compiler, as mentioned before. It can however be run on any project if used outside of the IDE. Also, the graphs produced start to get incomprehensible when the projects start to get large, specially the class dependency graph. Because of this it might be a good idea to divide the analysis of large project into smaller parts.

Acknowledgments

We would like to thank Jesper Öqvist for supervising this project.

References

[1] Inc. Clarkware Consulting. Jdepend, 2009. http://clarkware.com/software/JDepend.html [Online; accessed 21-January-2015].

[2] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Sys-*

tems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada, pages 1–18, 2007.

[3] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 57–76, 2007.

[4] R. Martin. Oo design quality metrics : an analysis of dependencies. *ROAD 1995*, 2(3), 1995.

[5] Jesper Öqvist and Görel Hedin. Extending the jastadd extensible java compiler to java 7. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Stuttgart, Germany, September 11-13, 2013*, pages 147–152, 2013.

[6] Robert S. Koss Robert C. Martin, James W. Newkirk. *Agile Software Development: Principles, Patterns and Practices*, pages 261–268. Pearson Education, 2012.

[7] CodePro: Java Developer Tools, 2012. https://developers.google.com/java-dev-tools/codepro/doc/features/metrics/metrics [Online; accessed 21-January-2015].