Package metrics on Java projects

Project in Computer Science - EDAN70
January, 2015

Joel Lindholm
D11, Lund Institute of Technology, Sweden
gda10jli@student.lu.se

Johan Thorsberg
D11, Lund Institute of Technology, Sweden
gda10jth@student.lu.se

Abstract

A tool for calculating metric suits over both java source code and its belonging libraries has been implemented. As of this report the tool is able to use the Chidamber & Kemerer object-oriented metrics suite and the Martin Metric suite. A configuration file exist and is used to change the behavior of the calculations and its output. There also is an ignore file that can be used to ignore specific packages, these will not be included in the calculations.

The tool has been implemented as extension to the JastAddJ compiler, and is therefore easily extended or modified.

Evaluation proved that the tool have an acceptable evaluation time.

Keywords Martin metrics, Java, Compiler, JastAdd, JastAddJ

1. Introduction

Collecting metric data during software development is an important task. It is impossible for every team member in a project to have a deep understanding of the complexity of every part of the software. Metrics can give an indicator of quality of the software or show where resources should be focused next. It is though not feasible to manually calculate metrics by hand, for example lines of code. Tools are needed to collect this data. The goal of this report is to describe a metric suite tool called MetricPCE, which is a acronym for Metric Extension with the first letter of the authors BitBucket usernames included. It is developed for the Java compiler JastAddJ[1]. It is a highly modular compiler due its aspect based programming. Extensions can then easily be added to the compile process.

The main task of this project was to implement the metrics presented by Robert Martin[2], which will be referred to as the *Martin metric*. Its purpose is to measure how well classes or packages in object oriented designs can be modified and extended or reused. The Martin metric analyses the dependencies, abstractness, instability and more to achieve this. Aspects from other metrics, like the Chidamber & Kemerer object-oriented metrics (CK metric)[3], has also been implemented in the tool. These metrics count for example the number of methods in classes.

The metrics work on different levels. Not only can the dependencies between classes be calculated, but also between source and library packages. It is also possible to remove packages from the result of the analysis. This is to remove unnecessary or unwanted clutter, for example dependencies to standard Java libraries. MetricPCE is therefore flexible due to that it allows the user to adjust the tool to its needs.

2. Background

2.1 JastAddJ and JastAdd

JastAddJ is a compiler build with the JastAdd[4] language for research purposes.

JastAdd is declarative, aspect oriented, java like programming language. It is a tool for constructing compilers. The idea is to use only one data structure during the whole compile process. Parsed tokens are put into an AST (Abstract Syntax Tree). Attributes and equations can be added to the different nodes, adding references between nodes are also possible. Java classes representing the nodes are constructed from the language. This makes it easy to add and modify the compiler, because the new code will automatically be put into the right Java class. For example this project is realized by adding a number of new attributes and methods to existing class nodes, which are used in the calculation of the metrics.

2.2 Chidamber & Kemerer object-oriented metrics

The CK metric is an widely used metric suite from 1994[3]. A number of metrics are calculated for each class in a project. In the beginning of this project we were given a older implementation of this metric. We made some alterations to bring this suite up to speed with the current stable implementation of the JastAddJ compiler, namely JastAddJ 7.1[5].

The following metrics, from the CK metric suite, are implemented in MetricPCE:

Weighted methods per class: The count of methods in a class. It indicates how rigid the design is for reuse. A class with a high count of methods is most likely designed for a single purpose, which will lead to limited ability for its reuse. Event handlers and constructors are also included.

Depth of inheritance How deep is the inheritance of a class or how many classes are above the class in its hierarchy.

Number of children How many immediate children the class has in the project, *i.e.* the width of a class hierarchy.

Coupling between object classes How many dependencies to other classes one class has. This set can be divided into two different sets. First the efferent couplings (Ce). This is the set of classes one class is dependent on. Other dependencies which are counted is calls to methods in other classes, inheritance and reference variables. The second set is afferent couplings (Ca), which contain all classes which depend on the one class, basically the opposite couplings of Ce.

Response for a class The number of methods in a class, plus the number of remote methods directly called from within the class.

Lack of cohesion in methods Lack of cohesion is a value representing the cohesiveness of a class, *i.e.* how well its methods functionality are related. A non-cohesive class is a class that preforms multiple functions with no relation between some of its internal attributes. Classes with a high lack cohesion can and should, according to this metric, be split into multiple smaller classes. A class with the value higher than one is considered to be poorly designed[6].

Number of public methods The total number of public methods and constructors in respective class.

2.3 The Martin metrics

Martin metrics was released the year 1994, the same year as the CK metric was released. The goal of the suite is to calculate the balance between instability and abstractness in packages. The Martin metric calculates dependencies, like the CK-metric dependencies between classes, between packages. These dependencies plus the abstractness of a package are used to calculate this balance, which is called Distance from the Main Sequence.

2.3.1 Implemented metrics

The following metrics are implemented in MetricPCE:

Efferent Couplings (Ce) The number of classes in other packages that this package has a dependency on.

Afferent Couplings (Ca) Opposite to efferent couplings, *i.e.* the number of classes in other packages that has a dependency on this package.

Number of Classes and Interfaces (NPI) As the name indicates, the total number of classes and interfaces in a package.

Abstractness (A) A ratio between the number of abstract classes and the number of concrete classes in a package. Ranges from 0 to 1, 0 indicates that the package is completely concrete and 1 is a completely abstract package. The access modifiers of the classes and interfaces will not affect the calculation of this metric attribute.

Instability (I) A ratio between efferent couplings and the total number of couplings in a package, this is given by the following equation.

$$\frac{Ce}{Ce + Ca} \tag{1}$$

It ranges from 0 to 1, 0 indicates that the package is completely independent. In short a package will be more stable if it has few efferent couplings as possible and as many afferent couplings as possible.

Distance from the Main Sequence (D) An ideal package, according to Martin, have a distance of close to zero to the main sequence line which is the line between the points (0, 1) and (1, 0), see Figure 1. The distance from this line is calculated with the equation.

$$D = |A + I - 1| \tag{2}$$

The most desirable spot for a package is on the end nodes, to be both completely abstract and stable or to be completely concrete and instable. In MetricPCE, this metric can range from [0,˜0.707] or it can be normalized to [0, 1]. This can be configured by the users.

2.3.2 Example

Here is a small example of Martin metrics. Two classes, Class1 and Class2, from two different packages, P1 and P2:

```
package P1;
import P2.Class2;
public class Class1{
   public void run(){
      Class2 c2 = new Class2();
      c2.run();
   }
}
package P2;
public class Class2{
   public void run(){
      System.out.print("Hello World");
   }
}
```

P1 has an efferent coupling to P2 and no afferent couplings. This will makes P1 fully unstable. Also it is completely concrete. P1 is right on the node (1, 0) in Figure 1. The relationship between abstractness and instability discussed above in this section results in a distance, D, of 0. This is an ideal package according to Martins metric. A concrete package can't be extended without alterations to its existing classes. Therefor the class must be instable, which means that the class can be changed without forcing changes in other classes.

P2 on the other hand have only one Afferent coupling which means the package has an instability of 0. This package is though fully concrete. This results in the maximum D of ˜0.707, and ends up on (0, 0) in the graph. We have a package that cannot be extended but is also completely stable, because changes can have an effect on dependent packages. This package is too rigid, we cannot do any alterations without expecting remote effects. This package should be rewritten or removed. For example Class2 could be added to P1 and P2 removed completely.

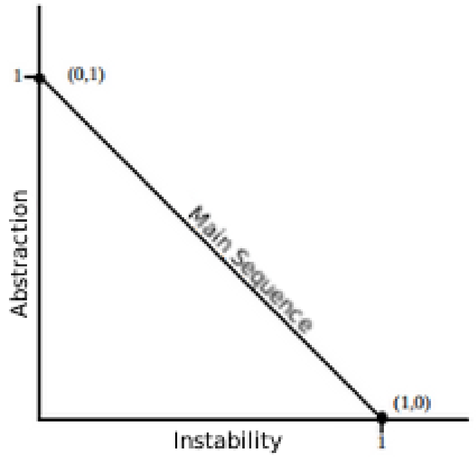

Figure 1. the main sequence line, $A + I = 1$[2].

3. MetricPCE

The result of this project is prototype tool called MetricPCE. MetricPCE calculates a number of metric suites over the source code of individual java projects. The tool is built as an extension of the JastAddJ compiler and will be using the compilers internal code structure to evaluate a given project. When this report is written, two sets of metrics have been implemented, the Martin metrics and the CK metric suite.

3.1 Capabilities

The capabilities of the tool will be explained in this section. There are three main features of the tool: Configure what will be calculated when the tool is used, filter out packages from the calculations and the different output files.

3.1.1 Configure the result

With the tool comes a configuration file. This file will be read by MetricPCE before it starts its calculations. The format of this file is a standard text file (.txt), and can be read and edited like one. Look at this example.

```
# This is a comment
configuration_name1 = 1
configuration_name2 = hello
```

Each row in the file is read as a new configuration in the tool. The name of the configuration and the value is split by an equal sign (=). Rows can be ignored by adding a number sign (#) in the beginning. Internally the tool can handle missing configurations and unused ones without crashing.

Examples of configuration are: what specific metrics should be calculated, what metrics should be included in the final result files, which kind of output files should be created and design options for the .dot files.

The format of the .dot files are of a simple text language used GraphViz[7], which is an open source graph visualization software. With this format nodes and their relations can be described. In the resulting .dot files nodes are represented as either classes or packages and their relations as the different couplings.

3.1.2 Ignore packages

With the tool comes an ignore file. In this file the user can specify which packages should be ignored by MetricPCE, when preforming its calculations. This can be used as an example to ignore standard libraries, which will otherwise alter the outcome of the calculations and clutter the dependency graphs.

3.1.3 Result files

As of this report three output files are available after the calculations. First is a text file, which contains the result in a simple text format. Each row represent a metric and its result, or to which package or class the following metrics belong to.

The second file is a .dot graph of dependencies between packages and its classes. Each node represent a package, and each package contains the classes which have dependencies outside of the package, see Figure 2. It will not be possible to decipher any information from theses graphs on larger projects without configuring the ignore file. This graph has been mainly used during the development of the tool itself. If the ignore file is used to filter out large part of a project, this type of graph can be useful to inspect specific parts of a project.

The third file is yet another graph, see Figure 3. This graph shows the dependencies between packages plus the result from martin metric. This type of graph gives the user a better overview of larger projects, where it might be more interesting to inspect the couplings between packages instead of individual classes. These graphs will not be as clustered as the class graphs when it comes to larger projects.

3.2 Internal structure

Figure 4 shows the modified internal AST structure of a project compiled with JastAddJ with the MetricPCE extension. The top

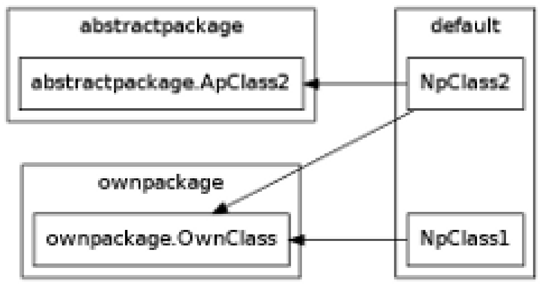

Figure 2. Packages with external dependent classes.

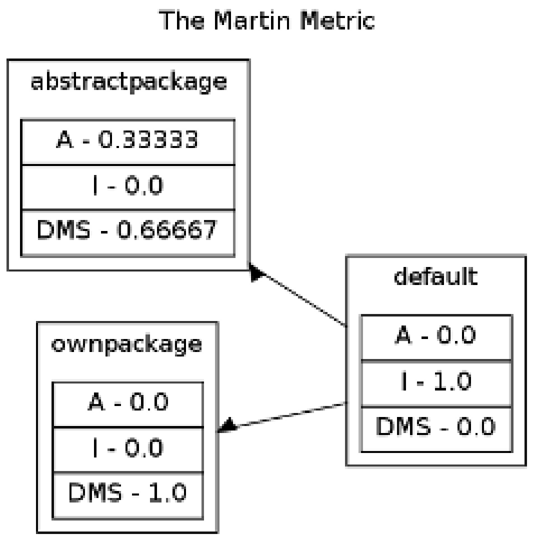

Figure 3. Packages external dependencies and martin metric result

node, Program, consist of a list with all the source files. These nodes are created during the parsing phase. Later the library files are added to the top node as NTA:s (Non-Terminal Attributes), *i.e.* they are created after the parsing.

In the AST library and source files are both of the type CompilationUnit.

There existed no node for packages in the structure of JastAddJ:s AST. The packages are only represented as strings in each source or library file. This was not deemed enough to represent the packages. Therefore a new parameterized NTA was added to the AST to represent the packages. Figure 4 shows how the structure looks like with the MetricPCE extension enabled. The new NTA:s are connected to the Program node and a reference to each class and Interface deceleration is added to the package nodes.

4. Evaluation

Obviously there are other tools that calculate the Martin metrics on Java object oriented projects, one of these tools are JDepend[9]. Most of these tools, JDepend included, analyses the generated '.class'-files instead of the source '.java'-files. This might result in deviations from the original definition of the Martin metric, and inaccurate calculations. For example different compilers might optimize some code by creating different .class files for a given object. It might add a constant and a nullable type of that object. This will result in a higher number of classes in the package which

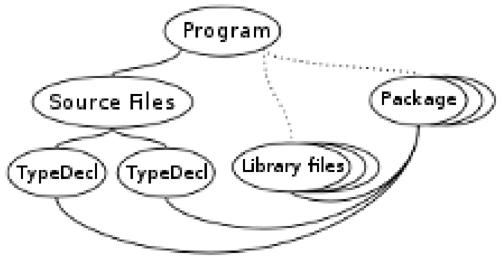

Figure 4. The internal representation of the project in JastAddJ, with the MetricPCE extension.

will alter the result of abstractness and instability. Another problem is that these type of tools can not differentiate between source classes and library classes. In the following section MetricPCE will be compared to JDepend.

4.1 Differences

At first a ratio of correctness between the two tools was supposed to be evaluated. This proved to be a problem, because the tools have different interpretation of how the Martin metric is evaluated. MetricPCE follow the original definition on couplings that calculate on classes to classes, see section 2.3.1 and 2.3.2. JDepend calculates this on a package to package level instead.

4.2 Correctness

The correctness of MetricPCE has been tested during the whole development process with a number of test cases. Some minor bugs have been noticed, but these do not alter the results at large. In some cases some library classes do not place themselves in their package. This has a small impact on afferent couplings on package level, which are in MetricPCE calculated based on the efferent couplings to a package. This occurs only for library packages and is not very common, and is therefore not considered a major problem. After these conclusions it was decided to do the evaluation based only on time.

Some errors in number of classes where noted in JDepend, but only with one or two to many.

4.3 Calculation time

To figure out how well the tool perform time wise, seven project of different sizes where selected, all taken from open repositories on GitHub. The total number of source lines of code (SLOC) where calculated. Despite their differences, the tools where configured to calculate the whole Martin metric suite. No packages or libraries where ignored. They were forced to produce one text file with the result. Error checking was enabled on MetricPCE, which might have slowed the process down, but not significantly. Both tools calculated the Martin metrics 10 times for each project, and each time was recorded. A confidence interval was calculated from the results.

4.4 Result

In Fig 5 the results of the calculation time have been plotted. The confidence interval shown in the graph is on 95%.

5. Discussion

Let's start with only looking at MetricPCE. From the graph in Figure 5, MetricPCE seems to grow linearly with the SLOC. The confidence interval for each project is almost invisible in the figure

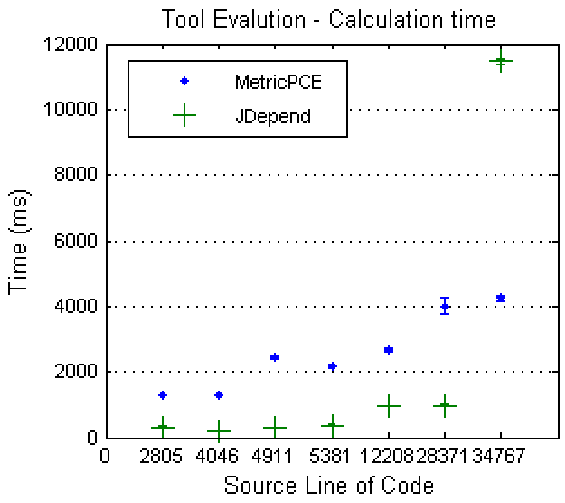

Figure 5. Graph of calculation time as function of SLOC.

due to the scale of the axes. Although one interval stands out, project number 6. The interval is slightly larger and we do not know why. The interval is 235 ms which is quite small, considering it takes around four seconds for the calculations, so the tool can still be considered stable.

The most interesting piece of the result is the last data-node for JDepend. JDepend's evaluation time is significantly less than MetricPCE:s, for most projects except for last. This project was reevaluated again, to make sure that the data was correct. The problem did not seem to be the source code of this particular project, but the extensive library that it used. As been said before, JDepend preforms its calculations on the java byte code, and therefore do not differentiate between library and source code. This forces the tool to evaluate the entire library. MetricPCE only calculates on classes that have dependencies to the source code, and can ignore large unused parts of libraries. This is believed to be the cause of this result.

6. Conclusion

We strongly believe that the tool, MetricPCE, is a good complement to the set of metric tools available. There still some minor bugs in the implementation, but they do not have a significant impact on the results. The time it takes to run the tool is within an acceptable time frame, around 4 seconds on 35000 SLOC, with an extensive library.

Compared to other metric tools like JDepend, the fact that MetricPCE can differentiate between source files and class files which gives it a slight advantage. It will only load library files that are used by the project. Execution time will not increase significantly when extensive libraries are used by projects. One drawback is that MetricPCE can only be used on source files.

6.1 Future improvements

Further extensions to MetricPCE can be added and tweaking the already existing features would increase the overall usability.

We tried to follow design principles to make the implementation of new metric suites as effortless as possible. It was not entirely possible mainly due to a lack of time. When extending MetricPCE with new metrics, the ideal would be that the old code could be left alone. Though not extensive, a few alterations are needed when adding a new metric suite.

The ignore feature has great use, but cannot do anything other than ignore packages. It would be useful to extend it so that it is to ignore specific classes in a package. Also adding the possibility to exclude a sub-package from the ignore file could be useful feature, *e.g* ignore java.* but still keep java.util.* in the evaluation.

When a package is ignored, it is ignored completely. That is, it will not be included in the calculations, and this will alter the results of the packages who have dependencies on ignored packages. An added configuration to only ignore files in the graph output but still keeping them in the calculations would be an interesting feature.

The output text file format should follow a well-known standard, like XML or JSON.

The result on the large file was interesting. We believe the result is because of the big library used in that specific project but there could be something else. More tests on other big projects should be done.

Acknowledgment

We would like to thank out supervisor Görel Hedin, who have been a great support throughout the whole project. We would also like to thank Jesper Öqvist, who has been a big help when we needed him.

References

[1] T. Ekman and G. Hedin: The JastAddJ extensible Java compiler, OOPSLA 2007. ACM, New York 2007

[2] R. Martin: OO Design Quality Metrics. An analysis of dependencies. ROAD 1995, vol 2. No 3. 1995

[3] R. Shyam Chidamber and Chris F. Kemerer. A Metrics Suite for Object Oriented Design. Transactions of software engineering, vol 20. No 6 june 1994.

[4] G. Hedin. Tutorial: An Introductory Tutorial on JastAdd Attribute Grammars GTTSE III, 166-200, LNCS 6491, 2011.

[5] J. Öqvist and G. Hedin, Extending the JastAdd extensible Java compiler to Java 7. ;In Proceedings of PPPJ. 2013.

[6] Tuomas Salste, Cohesion metrics, aivosto, read 17 dec 2014, `http://www.aivosto.com/project/help/pm-oo-cohesion.html#LCOM1`, read 17/12-2014.

[7] GraphViz. `http://www.graphviz.org/About.php`, read 19/1-2015.

[8] JastAdd Team, JastAdd Concept Overview, read 19 jan 2015, `http://jastadd.org/web/documentation/concept-overview.php`

[9] JDepend. `http://clarkware.com/software/JDepend.html`, read 13/12-2014.