Ilona BLÜEMKE*, Rafał ROGUSKI*

SOFTWARE ASSESSMENT USING OBJECT METRICS

Adequate metrics of object-oriented software enable to determine the complexity of a system, estimate the effort needed for testing and even locate some parts in the design that could be error prone or should be redesigned. The measurements and metrics are not widely used in the development of software. Contemporary systems are complex and adequate measuring tool is necessary in the design and evolution process. In this chapter we present how software metrics can be used in assessing software. Some software measuring tools (i.e. JMetrics, Essential metrics, Understand, X-Ray, Eclipse Metrics Plugin, Borland Together and Metrics) are briefly described. A critical comparison of these tools is also given. Next, a case study – an evaluation of three versions of a well known open source software - Junit is presented. To measure this application Metrics 1.3.6, an Eclipse plug-in, was used. The correlation of metrics values and real quality attributes of software is described. On the basis of metrics values some conclusions concerning the quality of the design of each version and some general trends in the evolution of JUnit project evolution are derived.

1. INTRODUCTION

Measurement, metrics, and statistical analysis of data are the basic tools of science and engineering. When the software industry began in the early 1950’s the first metric developed for quantifying the output of a software project was the metric named „lines of code” or LOC. Almost at once some ambiguity occurred, because a „line of code” could be defined either: a physical or logical one. Enough has been said about the inability of LOC to predict the complexity of software even the traditional one. The function point metrics proposed by Albrecht’s team and consisting of five key elements: inputs (screens, signals, etc.), outputs (screens, reports, checks, etc.), inquiries, logical files,
interfaces is not of good use for object-oriented software as well. Other traditional metric the McCabe Cyclomatic Complexity [26] can be used to determine the cyclomatic complexity only of the methods of a class not the whole software. This can be further used to determine a certain level of relative complexity of different classes in a system. The object-oriented literature abounds with descriptions of metrics for measuring multitude aspects of object-oriented designs and software.

With the aid of metrics and heuristics the whole software project or part of it e.g. classes, can be assessed [21, 27]. Design heuristics are rules of thumb that can guide designers as they choose between various alternatives, they capture the experience of skilled designers. A high degree of agreement between different authors in the literature might increase confidence in the validity of their heuristics. The utility of metrics will be questioned until sufficient number of empirical validations will be performed. The experiments should enable to establish a relationship between metrics and real quality attributes of a system, such as reliability, testability, maintainability, etc.

Adequate metrics of object-oriented software enable one to determine the complexity of a system and estimate the effort needed for testing already in the early stage of system development. The metrics values enable to locate parts of the design that could be error prone. Changes in these parts could significantly improve the quality of the final product and decrease testing complexity.

Software metrics research is an important topic, but not yet a well-formed or mature topic. Each of the major software metrics candidates has splintered into a number of competing alternatives, often following national boundaries. There is no true international standard for any of the more widely used software metrics. Further, the adherents of each metric variant claim remarkable virtues for their choice, and often criticise rival metrics. No conclusion has been reached yet about a good set of metrics for object-oriented systems.

Considerably research has been conducted on software metrics, some are described in monographs [13, 20, 31] others in separate papers e.g. [1, 4, 6, 8, 9, 20, 22, 23]. Several authors have shown the usability of object metrics in predicting test efforts or finding error prone parts in a program [4, 5, 6, 8, 10, 15, 19, 21].

There are also several works on assessing programs, especially Java programs, by object metrics. Some authors are assessing small students programs [2, 3], others are applying metrics to complex programs [12, 15, 19]. In [12] the results of measurements of several standard java libraries (J2SE, J2EE, J2ME, JWSDP) are presented.

According to the authors knowledge, recently few works has been done on the assessment of software evolution [14].

The goal of this chapter is to present how software metrics can be used in assessing
software. Metrics for object programs are briefly described in section 2. Seven software measuring tools (i.e. JMetrics [17], Essential metrics [11], Understand [29], X-Ray [33], Eclipse Metrics Plugin [24], Borland Together [7] and Metrics [25]) are presented in section 3. The advantages and disadvantages of each of these tools are pointed out. A critical comparison of these tools is also given. Next, in section 4, a case study an evaluation of three versions of a well known open source software - JUnit [18] is presented. To measure this application Metrics 1.3.6, an Eclipse plug-in, was used. The correlation of metrics values and real quality attributes of a system is described. On the basis of metrics values conclusions concerning the quality of the design of each version and some general trends in JUnit project evolution are derived. Some conclusions are given in section 5.

2. OBJECT METRICS

In recent years many researchers and practitioners have proposed metrics for object oriented software [1, 4, 6, 8, 9, 20, 22, 23]. Almost all research has been dedicated towards associating a complexity number either to a class or to the whole system. In subsection 1 some “class-level”, and in sub-section 2 “system-level” metrics are described. Some of the proposed metrics are based on the general object-oriented features, so they can be applied to any object-oriented software, others are dedicated to only one programming language like C++ [22]. The language dependent metrics comprise class-level and system-level metrics.

2.1. CLASS LEVEL METRICS

Programmers can use class-level metrics to identify error prone classes. The class-level metrics can be also used to estimate testing effort, the possibility of code reuse, and to improve the quality of the class code as well. Some metrics were introduced for measuring different aspects of classes. Widely known set of such object-oriented metrics was introduced by Chidamber and Kemerer [9] in 1994 and is presented below – CK metrics. CK metrics contain:

1. WMC - weighted methods per class,
2. DIT - depth of inheritance tree,
3. NOC - number of children,
4. CBO - coupling between objects,
5. RFC - response for a class,
6. LCOM - lack of cohesion in methods.

In their papers [9,10] Chidamber-Kemerer also provide an analytical confrontation of their proposed metrics with Weyuker’s [30] list of measurement principles.

2.2. SYSTEM LEVEL METRICS

The system-level metrics may be used by project managers to reduce the complexity of the design at early stages and to increase the quality of the design as well. Many metrics have been proposed to measure project characteristic. Below system-level metrics introduced by Brito et al. [8], called MOOD are presented. The metrics are evaluated in [16] and contain following metrics:
1. MHF – Method Hiding Factor,
2. AHF – Attribute Hiding Factor,
3. MIF - Method Inheritance Factor,
4. AIF – Attribute Inheritance Factor,
5. CF – Coupling Factor,
6. PF – Polymorphism Factor.

Other widely known set of system level metrics was proposed by Martin [23]. Martin’s metrics measure the quality of object design in terms of interdependence between subsystems. Designs which are highly interdependent tend to be hard to maintain and difficult to reuse. Interdependence is necessary if subsystems collaborate so some forms of dependency are desirable. Martin proposed following metrics:
1. Ca – Afferent couplings – the number of classes outside this category that depend upon classes within this category.
2. Ce – Efferent couplings – the number of classes inside this category that depend upon classes outside this categories.
3. I – Instability: \(C_e / (C_e+Ca)\), range \([0,1]\), \(I=0\) indicates a maximally stable category while \(I=1\) maximally instable one.
4. A – Abstractness: number of abstract classes in category / total number of classes in this category. \(A=0\) means concrete and 1 completely abstract class category.

To estimate and compare software projects “size” metrics are also used e.g.:
- TLOC - total lines of not commented code,
- NOCL – number of classes,
- NOI - number of interfaces,
- NOP - number of packages,
- NOA - number of all attributes,
- NSA – number of static attributes,
- NOM – number of methods,
- NORM - number of overridden methods,
- NSM - number of static methods,
- MLOC – method lines of code, not commented,
- NPAR – number of parameters in methods,
- NBD – nested block depth in a method.

3. METRICS MEASURING TOOLS

There are many programs calculating object metrics, some of them are stand alone e.g.: jMetrics [17], Essential metrics [11], Understand 2.0 [29], other are Eclipse plug-in e.g: XRay [33], Eclipse Metrics Plugin [24], Borland Together 2008 [7], Metrics 1.3.6 [25]. All these programs are intuitive and easy to use.

**jMetrics** [17] calculates only five metrics and files containing code must be manually added so this program may be used only for a very small application.

However **Essential metrics** [11] is a commercial program, and able to calculate much more metrics (twenty) than jMetrics, contains some errors. The unregistered version is able only to calculate metrics in five files only. We didn’t succeed in obtaining the registered version. Essential can be run from command line only. To calculate metrics for the whole project a file containing the list of all source files should be prepared. Essential is also able to generate some reports, such feature is not common in other metric measuring tools.

Program **Understand 2.0** [29] calculates more than twenty metrics and an educational licence can be obtained for one year. The implemented metrics are variants of basics ones e.g. for a metric $M$, the metric maximum of $M$ is also provided. Understand 2.0 has nice graphic interface presenting source file and metrics calculated in this file. For complex application, containing many files, the usage of Understand 2.0 is inconvenient. Each application file should be marked in browser window to see the values of its metrics. There are no tools supporting the comparison of metrics. Currently new version of Understand appeared. So far we are not aware about the functionalities of the version 2.5.
There are also several Eclipse plug-ins calculating metrics. **X-Ray** [33] was developed in Switzerland in 2007 by a student. Unfortunately the installation manual is imprecise and we are not able to run X-Ray. **Eclipse Metrics Plugin** [24] is able to calculate nine metrics also some CK, but the interface is not user friendly. All metrics values are presented as "warnings". In "property window" the metrics limits may be set. It is not possible to see the metric values in special window.

![Fig.1. Screen from Metrics calculating metrics for Junit 4.4](image)

**Borland Together 2008** [7] is integrated with Eclipse but can be also added as a plug-in into other Eclipse installation. It is a commercial product and to be able to use metric calculation the licence must be bought. Borland is able to calculate many metrics (all listed in section 2, more than twenty).
**Metrics 1.3.6** [25] is an open source Eclipse plug-in, with nice graphic interface and able to calculate many metrics. In figure 1 the screen with metrics for Junit 4.4 [18] is shown. The MOOD set of metrics is not available. The calculation of metrics can be switched on or off for the whole project. The time to calculate metrics depends on the number of files in a project and is acceptable even for complex application such as Junit or Vuze [32]. The values of metrics are stored in a file and the results can be presented in a tree. The user is able to specify lower and upper limits for each metric. Packages with metric values beyond limits, are displayed in red. Martin metrics $A$ and $I$ are calculated for packages only. We discovered some inconveniences in the calculation of CK metrics DIT and NOC. In the calculation of DIT metric, the inheritance of classes in standard Java libraries is also considered. An example is shown in figure 2. Calculating NOC (number of children) Metrics 1.3.6 counts all instances.

Despite some inconveniences we decided to use Metrics 1.3.6 for the assessment of evolution of several complex, open source, applications. Below, some results obtained for three versions of Junit program are presented. Measurements of several versions of programs Jedit and Vuze can be found in [28].

4. JUNIT EVALUATION ON THE BASIS OF METRICS VALUES

Three versions of Junit program i.e. 4.0, 4.4 and 4.8.1 were measured with Metrics1.3.6 plug-in. Some metrics values are given in table 1.

<table>
<thead>
<tr>
<th>Junit</th>
<th>TLOC</th>
<th>NOCL</th>
<th>NOP</th>
<th>NOA</th>
<th>NSA</th>
<th>NOM</th>
<th>NORM</th>
<th>NSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0</td>
<td>3081</td>
<td>58</td>
<td>11</td>
<td>84</td>
<td>26</td>
<td>353</td>
<td>14</td>
<td>101</td>
</tr>
<tr>
<td>4.4</td>
<td>4567</td>
<td>103</td>
<td>20</td>
<td>104</td>
<td>61</td>
<td>505</td>
<td>22</td>
<td>152</td>
</tr>
<tr>
<td>4.81</td>
<td>6149</td>
<td>130</td>
<td>27</td>
<td>146</td>
<td>38</td>
<td>691</td>
<td>25</td>
<td>176</td>
</tr>
</tbody>
</table>

Table 1. Size metrics of Junit program

The values of metrics in table 1 show, that the size of examined versions increases. From version 4.0 to 4.4 the number of classes, code lines increased almost 50%, from version 4.4 to 4.8.1 about 30%. Calculating metrics values the Metrics plug-in does not take into account libraries. The Junit project increases significantly. It can be noticed that the number of packages (NOP) increases in new version and is almost 100% greater in version 4.4 than in 4.0. The names of packages suggest, that new (4.4 and 4.8.1) versions contain experimental packages. Analyzing metrics calculated for packages with
name containing word “experimental” e.g. number of classes in a package, average number of code lines in a class it can be deduced that version 4.4 contains only skeletons of experimental packages which are fully implemented in version 4.8.1. In figure 2 the average number of lines of code per package is presented. The average number of lines of code in experimental packages increased significantly in version 4.8.1 comparing to version 4.4.

![Fig. 2 Average number of lines of code per package in Junit](image)

### Table 2 CK metrics of Junit programs

<table>
<thead>
<tr>
<th>Junit</th>
<th>NBD av.</th>
<th>NBD max</th>
<th>LCOM av.</th>
<th>LCOM max</th>
<th>WMC av.</th>
<th>WMC max</th>
<th>DIT av.</th>
<th>DIT max</th>
<th>MCCC av.</th>
<th>MCCC max</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0</td>
<td>6.09</td>
<td>3.81</td>
<td>0.26</td>
<td>0.52</td>
<td>1.15</td>
<td>3</td>
<td>1.43</td>
<td>10</td>
<td>3.81</td>
<td>10</td>
</tr>
<tr>
<td>4.4</td>
<td>6.55</td>
<td>3.65</td>
<td>0.21</td>
<td>0.51</td>
<td>1.15</td>
<td>4</td>
<td>1.38</td>
<td>11</td>
<td>3.81</td>
<td>10</td>
</tr>
<tr>
<td>4.81</td>
<td>7.55</td>
<td>4</td>
<td>0.22</td>
<td>0.56</td>
<td>1.15</td>
<td>4</td>
<td>1.4</td>
<td>10</td>
<td>3.81</td>
<td>10</td>
</tr>
</tbody>
</table>

### Table 3 Other metrics of Junit programs

<table>
<thead>
<tr>
<th>Junit</th>
<th>Ca</th>
<th>Ce</th>
<th>A</th>
<th>I</th>
<th>NBD av.</th>
<th>NBD max</th>
<th>MCCC av.</th>
<th>MCCC max</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0</td>
<td>6.09</td>
<td>3.81</td>
<td>0.26</td>
<td>0.52</td>
<td>1.15</td>
<td>3</td>
<td>1.43</td>
<td>10</td>
</tr>
<tr>
<td>4.4</td>
<td>6.55</td>
<td>3.65</td>
<td>0.21</td>
<td>0.51</td>
<td>1.15</td>
<td>4</td>
<td>1.38</td>
<td>11</td>
</tr>
<tr>
<td>4.81</td>
<td>7.55</td>
<td>4</td>
<td>0.22</td>
<td>0.56</td>
<td>1.15</td>
<td>4</td>
<td>1.4</td>
<td>10</td>
</tr>
</tbody>
</table>
Considering the values of metrics shown in table 2 and 3 it can be noticed, that the design of Junit is good. The number of methods per class is not very high, so the class behaviour can be understand and the code could be maintained properly. The DIT depth of inheritance tree has maximal value 5, the average is less than two. In calculating DIT Metrics counts also standard Java class inheritance. An example of inheritance tree is shown in figure 3. Unfortunately it is not possible to change in Metrics the way DIT is calculated, to block the inheritance of standard classes.

The analysis of size metrics helped in discovering a redundancy problem. The problem concerns classes `org.junit.Assert` and `org.framework.Assert`. The name suggests similar functionality, but after thorough examination of the code in all versions of Junit it turned out, that both classes have almost the same implementation. All of the methods have the same names and differences occur in few places in the program and mainly involve a greater number of conditional instructions in one of the methods. In all of the versions of Junit `org.framework.Assert` contains 38 static methods, while `org.junit.Assert` has 26 in version 4.0, 45 in version 4.4 and in the newest release class `org.junit.Assert` includes 49 static methods. It is most likely that in the future the class from package `org.junit` will replace the one from package `org.framework`. The above described problem can lead to a false interpretation of the flow of the application, especially because the methods are not called by the full name including the package, but after importing those packages and addressing those methods only by calling `Assert`. This strongly hampers the analysis of the source code as well as its maintenance. Without documentation or precise comments it will be very difficult to
provide any kind of changes regarding both classes org.junit.Assert and org.framework.Assert.

5. CONCLUSIONS

In this chapter a metric based analysis of three versions of an open source software – program Junit is presented. Availability of a suitable and adequate measuring tool at the early stage of a program development enables early prediction of the system complexity, thus reducing the cost of making necessary changes. Metrics calculation should also accompany the evolution of a project, for the same reasons as in the design. Software architects may optimize their design for better quality and easier maintenance. Open source systems are widely used and often they are developed differently than industrial ones. The code of new version should be measured by object metrics to observe the changes in the design. The correlation of metrics and real quality attributes of a system, such as reliability, testability, maintainability was studied by several authors but still new experiments and research should be conducted.

REFERENCES

[3] AL-JA’AFER J., SABRI K., Metrics for object oriented design (MOOD) to assess Java programs, King Abdullah II School for Information Technology, University of Jordan, Jordan, 2004
[19] JURECZKO M., SPINELLIS D., Using object-oriented design metrics to predict software defects, Models and methodology of system dependability, Wydawnictwo Politechniki Wroclawskiej, chapter 6, 69-81, 2010
[21] KIRSOPP C., SHEPPER M., WEBSTER S., An empirical study into the use of measurement to support OO design evaluation, ESERG TR99-01
[27] RIEL A.J., Object-oriented design heuristic, Addison Wesley: Reading, Massachusetts, USA, 1996
[29] Understand: http://www.scitools.com
[33] XRay: http://xray.inf.usi.ch