Uppsala Master's Theses in
Computer Science 276
2004-06-07
ISSN 1100-1836

Object-Oriented Design Quality Metrics

Magnus Andersson Patrik Vestergren

Information Technology
Computing Science Department
Uppsala University
Box 337
S-751 05 Uppsala
Sweden

This work has been carried out at
Citerus AB
Smedsgränd 2A
SE-753 20 Uppsala
Sweden

Supervisor: Patrik Fredriksson
Examiner: Roland Bol

Passed:

Abstract

The purpose of this report is to evaluate if software metrics can be used to determine the object-oriented design quality of a software system. Several metrics and metric-tools are presented and evaluated. An experimental study was conducted as an attempt to further validate each metric and increase knowledge about them. We present strategies on how analysis of source code with metrics can be integrated in an ongoing software development project and how metrics can be used as a practical aid in code- and architecture investigations on already developed systems.

The conclusion is that metrics do have a practical use and that they to some extent can reflect software systems design quality, such as: complexity of methods/classes, package structure design and the level of abstraction in a system. But object-oriented design is much more than that and the metrics covered by this report do not measure vital design issues such as the use of polymorphism or encapsulation, which are two vital parts of the object-oriented paradigm.

As long as no general design standard exists, general metric threshold values will be difficult to determine. Locally however, rules for writing code can be constructed and metrics can be used to assure that the rules are followed. So metrics do have a future but they will always be limited by the subjectivity of software design.

CONTENTS

1. INTRODUCTION

This report is founded on the following hypotheses:

- We believe that it is possible to find applicable software metrics that reflect the design quality of a software system developed in Java.
- We believe that these software metrics can be used in a software development process to improve the design of the implementation.

The report is divided into three parts: one theoretical and one practical part, that together result in a third concluding part. In the theory part we present what has been done in the area of object-oriented software metrics and investigate and select the metrics that are to be evaluated. The metrics were selected on the basis of their ability to predict different aspects of object-oriented design (e.g. the lines of code metric predicts a modules size). In the second part we collect tools that measure the selected metrics and apply them to a code base with well-known design quality. To get adequate results the code base must consist of both well and badly designed systems. Examples of well-designed systems were relatively easy to find, but when it came to badly designed code it became somewhat difficult. In the code selection process we concentrated on the authors reputation and experience, and the opinion of the general programming community of a code rather than the code itself. This because it is very time consuming to validate the code quality manually and there is no standard on how to perform such validation. To be able to get a good understanding on how to interpret the results of the measures, and evaluate the metrics, one must know what high quality code is, this is discussed later in this chapter.

In the third concluding part we discuss how the results from the practical experiment together with the theoretical part are to be interpreted. We discuss the validity of the metrics and suggest strategies on how they can be used in different stages in a software development process. We also present methods for improving a system's object-oriented design.

1.1 Background

Programmers have measured their code from the day they started to write it. The size of a program can easily be described as the number of lines of code it consists of. But when it comes to measuring the object-oriented design more complex measures are required. The two first object-oriented languages that introduced the key concepts of object-oriented programming were SIMULA 1 (1962) and the Simula 67 (1967) [11].

Two of the pioneers in developing metrics for measuring an object-oriented design were Shyam R. Chidamber and Chris F. Kemerer [3]. In 1991 they proposed a metric suite of six different measurements that together could be used as a design predictor for any object-oriented language. Their original suite has been a subject of discussion for many years and the authors themselves and other researchers has continued to improve or add to the "CK" metric suite. Other language dependent metrics (in this report the Java language is the only language considered) have been developed over the past few years e.g. in [32]; they are products of different programming principles that describes how to write well-designed code. Because the concept of good software design is so subjective, empirical studies are needed to clarify what measures describe good design, examples of such studies are [27], [29], [30], [31], [33], and [36].

1.2 Defining high quality code

It is hard to determine what is good code, but after 20 years of object-oriented programming the developer-community has come up with several criteria that should be fulfilled to make a software code considered to be good. It has to be agile, i.e. written so it can be reused and adapted. If it is not agile, it will become hard to add/remove modules and functionality to the program, and the developer has to concentrate on solving problems that propagates because of the changes rather than to solve new ones. It is important that the code is maintainable, i.e. it is easy to understand and correct, since there are often several developers involved in software development projects. For the code to be maintainable it can't be too complex, i.e. contain too many predicates, too much coupling and cohesion betweens objects and be messily written in general. Good understandable code means that it is well commented and written in a structured way with proper indenturing, these design issues doesn't reflect any part of the object-oriented design in a system and are therefore not covered in this report.

There is a standard quality model, called ISO 9126 (ISO 1991). In the standard, software quality is defined to be:

> The totality of features and characteristics of a software product that bear on its ability to satisfy stated or implied needs [4].

1.2.1 Package Structures in Java

As software applications grow in size and complexity, they require some kind of high-level organisation. Classes, while a very convenient unit for organizing small applications, are too finely grained to be used as the sole organisational unit for large applications [32]. This is where packages come in; the purpose of a package is to increase the design quality of large applications. By grouping classes into packages, one can reason about the design at a higher level of abstraction. The main goal when defining a new package structure of a software system is how to partition classes into packages. There are different ways to do this, in [32] Robert C. Martin propose a number of package cohesion

principles that addresses the goal of package structuring mentioned above. In short Martin argues that classes that are reused together should be in the same package and classes that are being affected by the same changes should also be in the same package. If a class in a package is reusable then all of the classes in that package must be reusable according to Martin.

Another possible approach of package structuring is to group classes into packages in terms of functionality. Take for instance a system that releases one version with a given functionality A and after some time another functionality B is demanded by the customer. In this case each function could be put in a package of it own to simplify the construct and development processes.

Many of the metrics in this report measure different couplings and relations between Java packages. As discussed earlier there are different ways of defining package structures and one must take this into account when interpreting the result of a package-metric. This report and all of the package-metrics in it presumes that a good package structure design is one where every package contains classes that are connected according to Robert C. Martins principles of package cohesion mentioned above. This presumption is made partly because the subject area is too immense to be handled in this report, and partly because the general object-oriented programming community accept these principles as being adequate methods for designing high quality package structures.

2. SELECTION OF METRICS

The selection of metrics investigated and presented in this chapter are collected on the basis of their ability to predict software design quality. The properties that should be fulfilled, so that the code can be considered to be good, are discussed in section 1.2.

2.1 Properties of a Metric

This section covers the definition and properties of metrics as tools to evaluate software. To validate a metric correctness there are two approaches: one is to investigate the metric empirically to see how well the measure correspond with actual data, the other is to use a framework of abstract properties that defines the criteria of a metric, such a framework is proposed in [2] and [6]. In this report metrics validated by either one or both approaches are investigated.

In the process of choosing what metrics are to be used as measurement, the first thing that has to be considered is from what viewpoint the measure is to be evaluated, i.e. what the main goal of the measurement is. As an example consider a metric for evaluating the quality of a text. Some observers might emphasize layout, others might consider language or grammar as quality indicators. Since all of these characteristics give some quality information it is difficult to derive a single value (metric) that describes the quality of a text. The same problem occurs for computer software. This observation indicates that a metric must be as unambiguous and specific as possible in its measure. In this report the goal is to find measures of object-oriented design quality, which is quite a big area. This suggests that many different metrics are needed, where each metric specializes on a specific area of design quality. The result of a metric must stand in proportion to what it measures. Take for instance the example above, a metric for defining the quality of a text, where a low result from the measure indicates low quality of that text. If the text-quality metric is to be of any use it must behave proportional i.e. the higher the result of the text-quality metric the higher quality the text holds. If a measure starts to behave in a random way it has no practical usage since it really doesn't measure anything.

2.2 Size Related

The metrics presented in this chapter are size related, i.e. they are used to give a survey of the scope of a program.

2.2.1 Lines of Code (LOC)

As the name indicates the LOC metric is a measure of a modules size, and it is perhaps the oldest software metric. The most discussed issue of the LOC metric is what to include in the measure. Fenton N. E. [4] examines four aspects of LOC that must be considered:

1. blank lines
2. comment lines
3. data declarations
4. lines that contain several separate instructions

One of the most widely accepted lines of code metrics is the one defined by Hewlett-Packard. In their metric blank and comment lines are removed, it is thereby often called NCLOC (non-commented LOC) or ELOC (effective LOC). In a sense, valuable length information is lost when this definition is used, but on the other hand it gives a much better value of the size of a program, since blank and comment lines isn't used by programs. Fenton recommends that the number of comment lines in a code (CLOC) should be measured and recorded separately. By doing this it is possible to calculate the comment ratio i.e. to which extent a program is self-documented: comment ratio = CLOC ÷ total lines of code. This measure doesn't give any information about the quality of the comments, so a high comment ratio does not imply well-commented code it only shows how much comments a module has.

The practical use of LOC in software engineering can be summarized as follows [24]:

- as a predicator of development or maintenance effort.
- as a covariate for other metrics, "normalizing" them to the same code density.

The most important point of these two usages of LOC, is the fact that it can be used as a covariate adjusting for size when using other metrics. Consider two different classes A and B, where A has a much higher LOC value than B. If one wants to compute the WMC metric (see section 1.4) for both classes, it is most likely that A returns the highest WMC value because of its greater size. This result is too dependent of a class' size and doesn't give a good representation of the WMC value. To correct this, LOC can be used to normalize both WMC values so that they don't depend on size but only on how high WMC both classes actually have.

2.2.2 Halstead Metrics

In 1976 Maurice Halstead made an attempt to capture notions on size and complexity beyond the counting of lines of code [4]. Halstead's metrics are related to the areas that were being advanced in the seventies, mainly psychology literature. Although his metrics are often referenced to in software engineering studies and they have had a great impact in the area, the metrics have been a subject of criticisms over the years, Fenton [4] writes:

> Although his work has had a lasting impact, Halstead's software science measures provide an example of confused and inadequate measurement.

However, other studies have empirically investigated the metrics and found them (or parts of them) to be good maintainability predictors [26] [28]. One metric that uses parts of Halstead's metrics is the maintainability index metric (MI, see section 2.2.3).

Definition

Before defining the metrics Halstead began defining a program P as a collection of tokens, classified as either operators or operands. The basic metrics for these tokens where:

- μ_1 = number of unique operators
- μ_2 = number of unique operands
- N_1 = total occurrences of operators
- N_2 = total occurrences of operands

For example the statement: f(x) = h(y) + 1, has two unique operators (= and +) and two unique operands (f(x) and h(y)).
For a program P, Halstead defined the following metrics:

- The **length** N of P: $N = N_1 + N_2$
- The **vocabulary** μ of P: $\mu = \mu_1 + \mu_2$
- The **volume** V of P: $V = N * \log_2 \mu$
- The **program difficulty** D of P: $D = (\mu_1 \div 2) * (N_2 \div \mu_2)$
- The **effort** E to generate P is calculated as: E = D * V.

In [4] Fenton states that the volume, vocabulary and length value can be viewed as different size measures. Take the vocabulary metric for instance: it calculates the size of a program very different from the LOC metric and in some cases it may be a much better idea to look at how many unique operands and operators a module has than just looking at lines of code. If, lets say a class, consists of four methods (operands): A, B, C and D. The LOC and vocabulary metrics will yield similar results, but if all four operands where implemented as A (four identical methods) the LOC metric would stay unchanged while the vocabulary metric is divided by four. The vocabulary metric shows the fact that: it is easier to understand a class with four identical methods than one with four different.

When it comes to the program difficulty and effort measure, Fenton finds them to be invalidated prediction measures, and that the theory behind them has been questioned repeatedly [4].

2.2.3 Maintainability Index (MI)

In an effort to better quantify software maintainability, Dr. Paul W. Oman, et al., has defined a number of functions that predict software maintainability [28]. In this section the two most common functions will be described. The MI functions are essentially comprised of the following three traditional measures: McCabe's CC metric, Halstead's volume and the LOC metric. The MI metric is a result of a cooperative metric research initiative sponsored by the Idaho National Engineering Laboratory (INEL) and the University of Idaho (UI) Software Engineering Test Laboratory (SETL). A large case study was conducted in collaboration with the U.S. Air Force Information Centre (AFIWC) [28] on their in-house electronic combat modelling system, known as the Improved Many-On Many (IMOM). The study showed a high correlation between a system's maintainability and the MI value. In practice the MI metric can be a good indicator of when an old system might be in need of re-engineering, or when a system under development needs to be re-designed because it is getting unmaintainable.

Different threshold values of MI have been proposed, and the most commonly referenced is the ones derived from a major research effort performed by Hewlett-Packard [14]. The first versions of the MI metric did include Halstead's effort metric but was later changed to only include the volume metric [28]. The following threshold values for MI in a module was proposed (derived from Fortran, C and Ada code):

- MI < 65 => poor maintainability
- 65 =< MI < 85 => fair maintainability
- MI >= 85 => excellent maintainability

Definition

3 metric MI equation, referred to as Maintainability Index Non-Commented (NC):

> MI = 171 – 5.2 * ln(aveV) – 0.23 * aveV(g') – 16.2 * ln(aveLOC)
>
> Where aveV is the average Halstead Volume per module (see section 2.2.2), aveV(g') is the average extended cyclomatic complexity (see section 2.3.2) per module and aveLOC is the average lines of code per module.

4 metric MI equation:

> MI = 171 – 5.2 * ln(aveV) – 0.23 * aveV(g') – 16.2 * ln(aveLOC) + 50 * sin(sqrt(2.4 * perCM))

Where aveV is the average Halstead Volume per module (see section 2.2.2), aveV(g') is the average extended cyclomatic complexity (see section 2.3.2) per module, aveLOC is the average lines of code per module and perCM is the average percent of lines of comments per module.

The *50 * sin(sqrt(2.4 * perCM))* expression in the 4 metric MI equation is noteworthy and it is there because an earlier version of MI included another expression (*0.99*aveCM*, where aveCM is the average lines of comments in a module) instead which caused the metric to be overly sensitive to the comment ratio of a module. By using the *50 * sin(sqrt(2.4 * perCM))* expression the comment ratio is given a maximum additive value to the overall MI.

The only difference between the two equations is the perCM parameter, and to determine which of these two equations is the most appropriate to measure the MI on given software system, some human analysis of the comments in the code must be performed. Oman et al. [28] makes the following observations on comments in a code:

- The comments do not accurately match the code. It has been said, "the truth is in the code." Unless considerable attention is paid to comments, they can become out of sync with the code and thereby make the code less maintainable. The comments could be so far off as to be of dubious value.

- There are large, company-standard comment header blocks, copyrights, and disclaimers. These types of comments provide minimal benefit to software maintainability. As such, the 4 metric MI will be skewed and will provide an overly optimistic maintainability picture.

- There are large sections of code that have been commented out. Code that has been commented out creates maintenance difficulties.

Generally speaking, if it is believed that the comments in the code significantly contribute to maintainability, the 4 metric MI is the best choice. Otherwise, the 3 metric MI will be more appropriate.

2.3 Method Level

The metrics presented in this chapter measure properties on method-level. These metrics doesn't say anything about the object-oriented design quality, instead it gives a survey of the complexity of methods, and thereof the understandability of the code.

2.3.1 Cyclomatic Complexity (CC)

Cyclomatic Complexity [10] was first proposed as a measurement of a modules logical complexity by T.J McCabe [8] in 1976. The primary purpose of the metric is to evaluate the test- and maintainability of software modules, and it has therefore been widely used in research areas concerned with maintenance. In practice the metric is often used to calculate a lower bound on the number of tests that must be designed and executed to guarantee coverage of all program statements in a software module. Another practical use of the metric is that it can be used as an indicator of reliability in a software system. Experimental studies indicate a strong correlation between the McCabe metric and the number of errors existing in source code, as well as the time required to find and correct such errors [12]. It is more difficult to find a practical use in terms of object-oriented design, but the measure can be a good indicator of the complexity of a method, which indirectly can be used to calculate the complexity of a class (see section 2.4.1).

Since the CC metric is a measure of complexity it is desirable to keep it as low as possible. The upper limit of CC has been a subject of discussion for many years and McCabe himself suggests, on the basis of empirical evidence, a value of 10 as a practical upper limit. He claims that if a module exceeds this value it becomes difficult to adequately test it [12]. Other empiric studies reach similar results and values between 10 and 20 are mentioned as an upper limit of CC [4].

Definition

McCabe's CC metric is defined as:

$v(G) = e - n + 2$

where v(G) = the cyclomatic complexity of the flow graph G of the module (method) in which we are interested,
e = the number of edges in G and n = the number of nodes in G [8].

To illustrate this definition consider this pseudo code example of a sorting algorithm:

```
   bubbleSort(array A)
1:          do while A not sorted
              set A as sorted
2:              for i=1 until i<A.size do
3:                if A.[i] < A.[i-1]
4:                  swap(A[i-1],A[i])
                    set A as unsorted
5:                end_if
6:              end_for
7:          end_while
8: end_bubbleSort
```

Now all we have to do is convert this code segment into a flow graph in order to calculate the cyclomatic complexity. This is done by using the numbers that indicate the statements in the code above, as nodes in our flow graph:

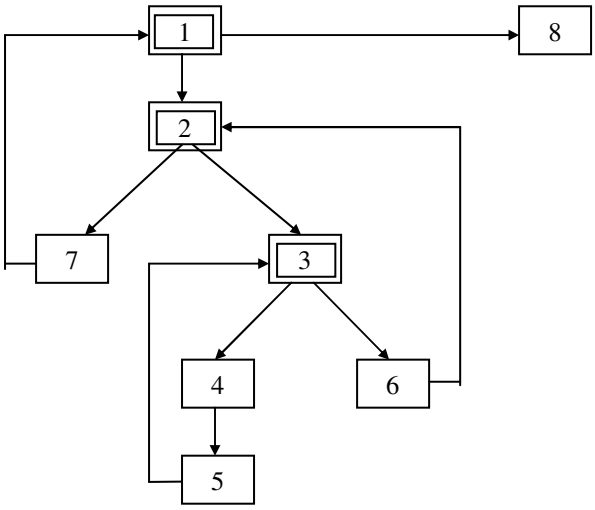

Figure 2.1: flow graph representation.

The number of edges = 10 and number of nodes = 8 this yields a CC value v(G) = 10 – 8 + 2 = 4.
One more way to calculate v(G) is:

v(G) = P+1

where P is the number of predicate nodes* found in G. In figure 2.1 nodes number 1,2 and 3 (the double squared nodes) are predicate nodes and thus gives a value of v(G) = 3+1 = 4.

* A predicate node is a node that represents a Boolean statement in the code, thus giving it two outgoing edges instead of one.

2.3.2 Extended Cyclomatic Complexity Metric (ECC)

The ECC metric measures the same thing as the CC metric but it also takes into account compound decisions and loops (AND/OR). Take for instance the following code examples:

Example 1

```
if( y < 5)
  //do something…
```

Example 2

```
if( y < 5 AND y > 1 )
  //do something…
```

Example 1 yields a complexity value of 1+1 = 2 for both the CC and ECC metrics. In example 2 the CC metric will still result in 2 but the ECC value will be 3 because of the AND-statement within the if control structure.

Definition

The ECC metric is very similar to the CC metric with the only difference that it's complexity increases with one for each compound statement within a control structure.

$$ECC = eV(G) = P_e + 1.$$

Where P_e is the number of predicate nodes in the flow graph G (see section 2.3.1) weighted by (add one for each statement) the number of compound statements (AND/OR) it has.

2.4 Class Level

The metrics presented in this chapter measure systems at class-level; i.e. the classes within a package. Some of the metrics could be used on package level, for instance: the depths of inheritance tree, but those are not covered in this report.

2.4.1 Weighted Methods per Class (WMC)

S.R Chidamber and C.F Kemerer first proposed the WMC metric in 1991 [3] and it relates directly to the definition of complexity of an object. The number of methods and the complexity of methods involved are indicators of how much time and effort is required to develop and maintain the object. The larger the number of methods in an object, the greater the potential impact on the children, since, children will inherit all the methods in the object. A large number of methods can result in a too application specific object, thus limiting the possibility of reuse [1].

Since WMC can be described as an extension of the CC metric (if CC is used to calculate WMC) that applies to objects, its recommended threshold value can be compared with the upper limit of the CC metric (see section 2.3.1). Take the calculated WMC value and divide it with the number of methods, this value can then be compared with the upper limit of CC. One disadvantage of using CC in order to measure an objects complexity is that the WMC value cannot be collected in early design stages, e.g. when the methods in a class has been defined but not implemented. To be able to measure WMC as early as possible one could use the number of methods in a class as a complexity value, but then the WMC metric is no longer a complexity measure but a size measure, also known as the number of methods metric (NM) [16].

Definition

Consider a class C_1, with methods $M_1, M_2, \ldots M_n$. Let $c_1, c_2, \ldots c_n$ be the static complexity of the methods. Then:

WMC = $\sum_{i}^{n} c_i$, where n is the number of methods in the class.

If all static complexities are considered to be unity, WMC = n, the number of methods [7].

The static complexity of a method can be measured in many ways (e.g. cyclomatic complexity) and the developers of this metric leave this to be an implementation decision.

2.4.2 Lack of Cohesion in Methods (LCOM)

The cohesion of a class is characterized by how closely the local methods are related to the local instance variables in a class [4]. S.R Chidamber and C.F Kemerer first defined the LCOM metric in 1991 [3]. Since then several more definitions of LCOM have been proposed and there is still research conducted in this area. In [18] the authors compare and evaluate different LCOM definitions and reach the conclusion that Li and Henry's definition [20] is more rigorous than the CK LCOM metric. Both these definitions plus another proposed by Henderson-Sellers [17] will be covered in this section. The LCOM metric is a value of the dissimilarity of the methods in a class. A high LCOM value in a class indicates that it might be a good idea to split the class into two or more sub-classes. Since the class might have too many different tasks to perform, it is better (design-wise) to use more specific objects. Because LCOM is a value of dissimilarity of methods, it helps to identify flaws in the design of classes [3]. Cohesiveness of methods within a class is desirable, since it promotes encapsulation and decreases complexity of objects.

It is hard to give any exact threshold values of the LCOM metric since the result can be viewed from different perspectives, such as reusability, complexity, and maintainability of a class. Studies of the LCOM metric have shown that high values of LCOM were associated with: lower productivity, greater rework and greater design effort. Studies also show that LCOM can be used as a maintenance effort predictor [20], [22].

Chidamber-Kemerer Definition

Consider a class C_1 with n methods $M_1, M_2, \ldots, M_n$. Let $\{I_j\}$ = set of instance variables used by method M_i. There are n such sets $\{I_1\}, \ldots, \{I_n\}$. Let $P = \{(I_i, I_j) \mid I_i \cap I_j = \emptyset\}$ and $Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \emptyset\}$. If all n sets $\{I_1\}, \ldots, \{I_n\}$ are $\emptyset$ then let $P = \emptyset$.

LCOM = $|P| - |Q|$, if $|P| > |Q|$
= 0 otherwise [7].

The definition of disjointness of sets given in the Chidamber-Kemerer definition is somewhat ambiguous, and was further defined by Li and Henry [18].

Li-Henry Definition

> LCOM* = number of disjoint sets of local methods; no two sets intersect; any two methods in the same set share at least one local instance variable; ranging from 0 to N; where N is a positive integer.

To illustrate these two different definitions consider the following example:

Given a class C and its variables a, b, c and d the following methods are present:
Method W accesses variables {a, b}.
Method X accesses variable {c}.
Method Y accesses no variables.
Method Z accesses {b, d}.
Using the Li and Henry definition of the LCOM metric (LCOM*), the disjoint sets of methods, where any two methods in the same set share at least one local instance variable, would be:
{W, Z}, {X}, {Y}
The result is three different sets and thus the LCOM value is 3.
Using the definition proposed by Chidamber and Kemerer to calculate LCOM:

$W \cap X = \emptyset$

$W \cap Y = \emptyset$

$W \cap Z = \{b\}$

$X \cap Y = \emptyset$

$X \cap Z = \emptyset$

$Y \cap Z = \emptyset$

P = 5, the intersections whose result is ø. Q = 1, the intersections whose result is not ø.
LCOM = |P| - |Q| = 5 – 1 = 4.

Another example where these two definitions return different values is in the case of a perfectly cohesive class (all methods are related to one and other), where measured by the Li and Henry metric, LCOM would have a value of 1 (one set of methods), whereas the same class would have a value of 0 when measured with Chidamber and Kemerer's metric (|Q| > |P|). In the case of a perfect un-cohesive class the value of the Li and Henry metric would equal the number of methods in the class. The same class measured with the CK metric would yield a higher value since the metric would equal to n taken two at a time ((½ n)(n-1)), n>1), where n is the number of methods in the class. Take for instance three classes: A, B and C containing 3, 4, respective 5 methods that doesn't intersect one and another, the CK metric will return the values: 3, 6 respective 10 (all methods taken two at a time), while LH's metric results in the values: 3, 4 respective 5 (the number of different sets of non-intersected methods).

Both metrics behave in the same way i.e. results in higher LCOM values for classes with higher lack of cohesion in its methods, but the LCOM* metric is the more un-ambiguous of the two. Consider yet another example that illustrates the ambiguousness in the Chidamber and Kemerer definition:
In the special case of a class where none of the methods access any of the class' member variables the CK definition says that if all n sets $\{I_1\}, \ldots\{I_n\}$ are null sets then $|P| = 0$, and thus LCOM = 0. So far ok but if only one of the methods changes its behaviour and accesses one or more attributes in the class, the LCOM value for that class becomes equal to the number of methods. In both of these cases the class is still completely un-cohesive because it has no methods that share any member variables, but the difference in the LCOM value indicates otherwise. With the LCOM* definition both these cases would yield the same LCOM value.

In [17] B. Henderson-Sellers discuss one serious problems with the CK definition of LCOM: there are a large number of dissimilar examples, all of which will give a value of LCOM = 0. Hence, while a large value of LCOM suggests poor cohesion; a zero value does not necessarily indicate good cohesion (since there are no negative LCOM values in the CK definition). Henderson-Sellers propose a new definition of LCOM, called LCOM**.

Henderson-Sellers Definition

$$LCOM^{**} = (mean(p(f)) - m) \div (1 - m)$$

Where m is the numbers of methods defined in a class A. Let F be the set of attributes defined by A. Then let p(f) be the number of methods that access attribute f, where f is a member of F.

In the case of perfect cohesion (all methods access all attributes) the value of LCOM** = 0. A totally un-cohesive class (where each method only accesses a single variable) results in LCOM** = 1.

2.4.3 Depth of Inheritance Tree (DIT)

Inheritance is when a class share the same structure or behaviour defined in another class. When a subclass inherits from one superclass it's called a single inheritance and when a subclass inherits from more than one superclass it's called multiple inheritance. Inheritance through classes increases its efficiency by reducing the redundancy [5]. But the deeper the inheritance hierarchy is, the greater the probability is that it gets complicated and hard to predict its behaviour.
To give a measure of the depth in the hierarchy Shyam R. Chidamber and Chris F. Kemerer in 1991 [3] proposed the depth of inheritance tree metric.

In [25] J. Bloch points out some dangers with using inheritance across packages, since packages often are written by different developers. One problem is that the subclass depends on its superclass, and if a change is made in the superclass's implementation it may cause the subclass to break. According to

Bloch, inheritance should be used only when completely certainty exists that a class A that extends a class B really is a specialization of class B. If not completely certain about this, then composition should be used instead, where composition is when the new class is given a private field that references an instance of the existing class, then the existing class becomes a component of the new one. When extending classes is specifically designed and documented for extension or the classes is in the same package under control of the same programmers, J. Bloch thinks inheritance is a powerful way to achieve code reuse [25].

There have been several studies on what threshold value the DIT metric should have in a system, but no clear one has been found or even a direction thereof. With a high value of the DIT there will be good reusability but on the other hand the design will be more complex and hard to understand. It is up to the developer to set an appropriate threshold depending on the current properties of the system.

Chidamber-Kemerer Definition

> Depth of inheritance of the class is the DIT metric for the class. In cases involving multiple inheritances, the DIT will be the maximum length from the node to the root of the tree [7].

In [9] W. Li founds two ambiguous problems with this definition. The first is that when multiple inheritance and multiple roots are present at the same time. The second is a conflict between the definition, theoretical basis and the viewpoints stated in [7]. Both the theoretical basis and the viewpoints indicated that the DIT metric should measure the number of ancestor classes of a class, but the definition of DIT stated that it should measure the length of the path in the inheritance. The difference is illustrated in the example below.

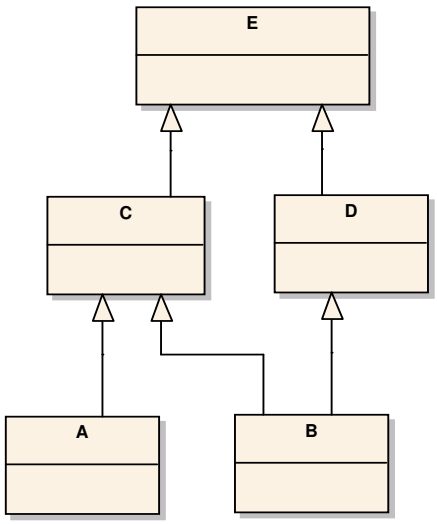

Figure 2.2: Inheritance tree

According to [7] the same maximum length from the root of the tree to the nodes, thus DIT(A) = DIT(B) = 2. However, class B inherits from more classes than class A does and according to the theoretical basis and the viewpoints, classes A and B should have different DIT values, DIT(A) = 2, and DIT(B) = 3 [9].

Because of these ambiguous problems W. Li introduced a new metric called Number of Ancestor Classes (NAC). It is an alternative for the DIT with the same theoretical basis and viewpoints but with the definition.

Wei Li Definition

> NAC measures the total number of ancestor classes from which a class inherits in the class inheritance hierarchy [9].

2.4.4 Number of Children (NOC)

The number of children metric was proposed by Shyam R. Chidamber and Chris F. Kemerer in 1991 [3] to indicate the level of reuse in a system, and hence indicate the level of testing required. The greater the number of children is in an inheritance hierarchy the greater the reuse, since inheritance is a form of reuse. But if a class, or package, has a large number of children it may be a case of misuse of sub-classing, because of the likelihood of improper abstraction of the parent class. In the case of a class, or package, with a large number of children, the class, or package, may require more testing of the methods [1].

In [3] Chidamber and Kemerer proposed that it is better to have depth than breadth in the inheritance hierarchy, i.e. high DIT and low NOC. In [13] Sheldon, Jerath and Chung had the opinion:

> Our primary premise argues that the deeper the hierarchy of an inheritance tree, the better it is for reusability, but the worse for maintenance. The shallower the hierarchy, the less the abstraction, but the better it is for understanding and modifying. Taken the maintenance point of view, it is recommended that a deep inheritance tree should be split into a shallow inheritance tree.

It is up to the developer to find a proper threshold value of the current system, since there are no empirically or theoretically found one. The developer have to take in consider the specific properties of the system.

Definition

> NOC = number of immediate subclasses subordinated to a class in the class hierarchy [7].

According to the definition of NOC only the immediate subclasses are counted, but a class has influence over all its subclasses. This is pointed out in [9] there W. Li propose a new metric, Number of Descendent Classes (NDC) that has the same properties as NOC but with another definition:

> The NDC metric is the total number of descendent classes (subclasses) of a class [9].

2.4.5 Response For a Class (RFC)

If a class consist of a large number of methods it is likely that the complexity of the class is high. And if a large number of methods can be invoked in response to a message received by an object of that class it is likely that the maintenance and testing becomes complicated. Shyam R. Chidamber and Chris F. Kemerer proposed in 1991 [3] the response for a class metric to measure the number of local methods and the number of methods called by the local methods.

There is no specific threshold value on the RFC metric evolved. But in [3] Chidamber and Kemerer suggest that the greater the value of the RFC, the greater the level of understanding required on the part of the tester.

Definition

RFC = |RS| where RS is the response set for the class, given by

RS = $\{M\} \cup_{\text{all } i} \{R_i\}$

where $\{R_i\}$ = set of methods called by method i and

{M} = set of all methods in the class [7].

To illustrate this definition consider the following example:
A::f1() calls B::f2()
A::f2() calls C::f1()
A::f3() calls A::f4()
A::f4() calls no method
Then RS = {A::f1, A::f2, A::f3, A::f4} U {B::f2} U {C::f1} U {A::f4}
= {A::f1, A::f2, A::f3, A::f4, B::f2, C::f1}
and RFC = 6

2.4.6 Coupling Between Objects (CBO)

A class is coupled to another if methods of one class use methods or attributes of the other, or vice versa. Coupling between objects for a class is the number of other classes to which it is coupled.
In [19] R. Marinescu lists some impacts on having high coupling on quality attributes in a system.

- The reusability of classes and/or subsystems is low when the couplings between these are high, since a strong dependency of an entity on the context where it is used makes the entity hard to reuse in a different context.

- Normally a module should have a low coupling to the rest of the modules. A high coupling between the different parts of a system has a negative impact on the modularity of the system and it is a sign of a poor design, in which the responsibilities of each part are not clearly defined.

- A low self-sufficiency of classes makes a system harder to understand. When the control-flow of a class depends on a large number of other classes, it is much harder to follow the logic of the class because the understanding of that class requires a recursive understanding of all the external pieces of functionality on which that class relies. It is therefore preferable to have classes that are coupled to a small number of other classes [19].

In [17] Henderson-Sellers et al. states that class coupling should be minimized, in the sense of constructing autonomous modules, though a tension exists between this aim of a weakly coupled system and the very close coupling evident in the class/superclass relationship. Without any coupling the rationale is that the system is useless, so consequently, for any software solution there is a baseline or necessary coupling level, it is the elimination of irrelevant coupling that is the developer's goal. Such unnecessary coupling does indeed needlessly decrease the reusability of classes [17].

There are several different definitions of coupling, depending on the purpose, e.g. internal data-, global data-, sequence-, parameter-, inheritance coupling, package coupling etc. There can be a high coupling value between classes in a package, but the package can have low coupling between other packages in a system. This report covers the basic, wide used CBO metric proposed by Shyam R. Chidamber and Chris F. Kemerer in 1991 [3]. Their viewpoint where that excessive coupling prevents reuse, since the more independent a class is, the easier it is to reuse it in another application. They also claimed that to improve modularity and promote encapsulation, inter-object class couples should be kept to minimum. Thus the larger the number of couples the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult. They also meant that a measure of coupling is useful to determine how complex the testing of different parts of the design is likely to be, thus the higher the inter-object class coupling, the more accurate the testing needs to be [7].

Definition

CBO for a class is a count of the number of other classes to which it is coupled [7].

2.5 Package Level

The metrics presented in this chapter are measured on package-level; some measures properties between packages and some measures the individual properties on the packages.

2.5.1 Instability and Abstraction

Stable-Abstraction Principle (SAP)

> A package should be as abstract as it is stable, R. C. Martin [32].

This principle states that a stable package should also be abstract so that its stability does not prevent it from being extended. A stable package is a package that has no dependencies with other packages, i.e. it is completely independent. If a package for example only contains shapes like a square, circle and a triangle the package is concrete, i.e. it is hard to add functionality to the package. But if the package only contains an interface or abstract class called shape, then it will be easier to add new functionalities, e.g. it is easier to create one print method that prints a shape than to add a new print method for every geometric figure. The SAP sets up a relationship between stability and abstractness. It also says that an instable package should be concrete since its instability allows the concrete code to easily be changed. If a package is to be stable, it should also consist of abstract classes so that it can be extended. Packages that are stable and that are extensible are flexible and do not overly constrain the design [32]. To be able to measure the SAP, R. C. Martin defined an instability metric (I) and an abstractness metric (A), with these metrics he could define a metric D, that will represent the SAP. These metrics are presented below.

The instability metric was proposed by R. C. Martin in [21] to give a measure of the stability of a class. The definition is $I = (Ce \div (Ca + Ce))$, with the range [0,1]. I=0 indicates a maximally stable package. I=1 indicates a maximally instable package. Ca is the afferent coupling, which is the number of classes outside the package that depend upon classes inside the package, (i.e. incoming dependencies). Ce is the efferent coupling, which is the number of classes outside the package that classes inside the package depend upon, (i.e. outgoing dependencies) [23].

A system with packages that are maximally stable would be unchangeable, therefore it is better if some portions of the design are flexible enough to withstand significant amount of change. There is one sort of class that can be maximally stable and flexible enough to be extended without requiring modification namely abstract classes. Thus, if a package is to be stable, it should also consist of abstract classes so that it can be extended, since stable packages that are extensible are flexible and do not constrain the design [21]. There must be classes, outside the abstract package, that inherit from it and implement the missing pure interfaces, and therefore the abstract package must have dependents. There should not be any dependencies upon instable packages, thus instable packages should not be abstract they should be concrete [21]. For measuring the abstractness in a package R. Martin defined a metric $A = Na \div Nc$ in [23]. The A metric has a range of [0,1], like the I metric. A=0 means that the package contains no abstract types. A=1 means that the package contains nothing but abstract types. Na is the number of abstract types in the package and Nc is the total number of classes in the package.

To define the relationship between stability (I) and abstractness (A) Martin created a graph with A on the vertical axis, and I on the horizontal axis. The maximally stable and abstract packages are at the upper left corner (0,1) and the maximally instable and concrete are at the lower right corner (1,0). To represent a package that is partially extensible, e.g. I=0.5 and A=0.5 Martin draw a line from (0,1) to (1,0) that he call the main sequence. The graph is illustrated below [21]:

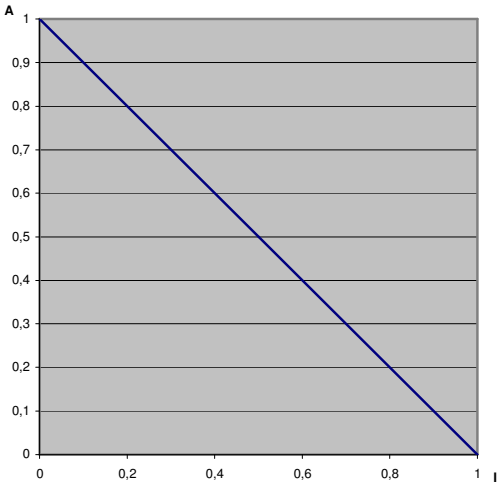

Figure 2.3: The Main Sequence

According to Martins hypothesis the packages that sits on the main sequence is not too abstract, nor too instable, instead it has the right number of concrete and abstract classes in proportion to its efferent and afferent dependencies. The most optimal position for a package is on the two endpoints (0,1) or (1,0) on the main sequence.

Since the most desirable is when a package sits on the main sequence, it would be good if the distance to the main sequence could be measured, R. C. Martin came up with the distance $D = |(A+I-1)| \div \sqrt{2}$ [21], this metric ranges from [0, ~0.707]. To normalize it, the distance is defined as $Dn = |(A+I-1)|$, this is more convenient since it ranges from [0,1]. Dn=0 means that the package is directly on the main sequence. Dn=1 means that it is as far away as possible from the main sequence [21].

2.5.2 Dependency Inversion Principle Metric (DIP)

This DIP metric is based on a software design principle developed by R. C Martin [32], called the dependency inversion principle:

- High-level modules should not depend on low-level modules. Both should depend on abstractions.

- Abstractions should not depend on details. Details should depend on abstractions.

In short the principle states that objects (classes or packages in Java) should depend on abstract entities (high-level modules), not concrete ones (low-level modules). The reason for this is that concrete things change a lot; abstract things change much less frequently. Abstractions represents places were the design can be bend or extended without themselves being modified, which gives high reusability and good design. To illustrate the use of the dependency inversion principle, consider the following example:

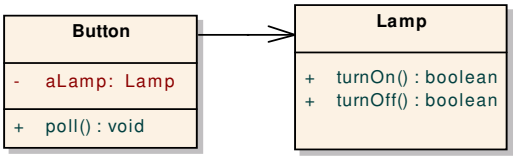

Figure 2.4: Naive model of a button and a lamp (before applying DIP).

Figure 2.4 displays a system consisting of a button that controls a lamp. The `Button` object receives `poll` messages from an outside user object and then sends a `turnOn` or `turnOff` message to the `Lamp`. If this system is to be coded in Java the `Button` class will depend directly on the `Lamp` class. This dependency implies that the `Button` class will be affected by changes to the `Lamp` class. Another disadvantage with this system is that the `Button` class cannot be reused to control something other than `Lamp` objects. To make it more clear the Java code for the `Button` class is provided:

Button.java

```
public class Button{

  private Lamp aLamp;
  public void poll(){
     if(/* some condition */)
       aLamp.turnOn();
     }
  }
}
```

This solution to the `Button` and `Lamp` system violates the dependency principle. The abstraction has not been separated from the details i.e. the `Button` object depends on the `Lamp` object. To fix the system so that it correspond to the dependency inversion principle the button object must depend on an abstraction, and thus the `Button` doesn't need to know that it is a `Lamp` that's being turned on, it just needs to send `turnOn` or `turnOff` messages to some object without knowing anything about it. By introducing an abstract object that serves as a contract between objects so that they don't need to know any details of one another, the DIP definition can be fulfilled. To illustrate this improved version of the `Button` and `Lamp` system an abstract object, in this case an interface called `ButtonServer`, is introduced. `ButtonServer` provides abstract methods that `Button` can use to turn something on or off. The `Lamp` object implements the `ButtonServer` interface and thus, `Lamp` is now doing the depending, rather than being depended on.

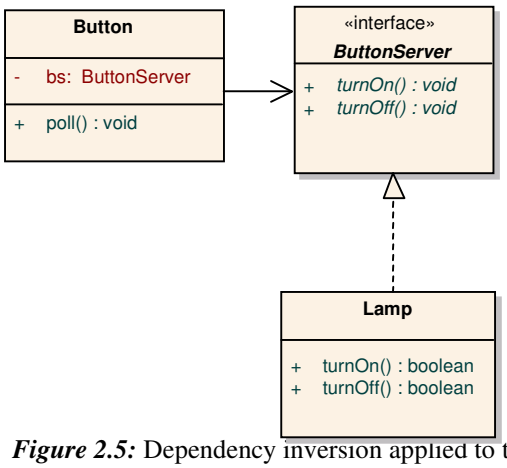

Figure 2.5: Dependency inversion applied to the `Lamp`.

The design in figure 2.5 allows button to control any device that is willing to implement the `ButtonServer` interface. This results in good flexibility and it also implies that `Button` objects will be able to control objects that have not yet been invented. R. C. Martin states that inversion of dependencies is one of the main characteristics of a good object-oriented design and if dependencies are not inverted the system has a procedural design [32].

Definition

The DIP metric is defined as:

> DIP = the percentage of dependencies in a package or class that has an interface or an abstract class as a target. A DIP of 100 % indicates that all dependencies in a module are based on interfaces or abstract classes.

2.5.3 Acyclic Dependency Principle Metric (ADP)

The ADP metric is derived from another software design principle defined by R. C. Martin [32], the acyclic dependency principle:

> The dependencies between packages must form no cycles.

The purpose of this principle is to make all packages in a system to be as self-contained as possible. Cycles between packages makes packages depend on each other, which causes compatibility problems between the packages. One can say that a cyclic dependency between two packages in a way makes them so dependent of each other that they can be considered to be one large package, since none of them can be used without affecting on the other.

Definition

> The ADP metric gives the percentage of dependencies that form no cycles. If the metric is 100 %, it means that there are no cycles in the package structure. If the metric is 90 %, it means that 10 % of the dependencies need to be fixed minimally to get the package structure to conform to the ADP.

The ADP metric analyses the package structure on a single level i.e. it doesn't consider how sub-packages are structured. There is a variant of the ADP metric called the ADP Recursive (ADPR) that analyses dependencies in the complete package tree.

2.5.4 Encapsulation Principle Metric (EP)

In a good modular design, each of the modules encapsulates implementation details that are not visible to the user of the module. The encapsulation principle is defined as:

> A substantial part of a package should not be used outside of the package.

This means that a package within a package (the child to that package) is not to be used, as much as possible, by other modules. Take the example package structure below:

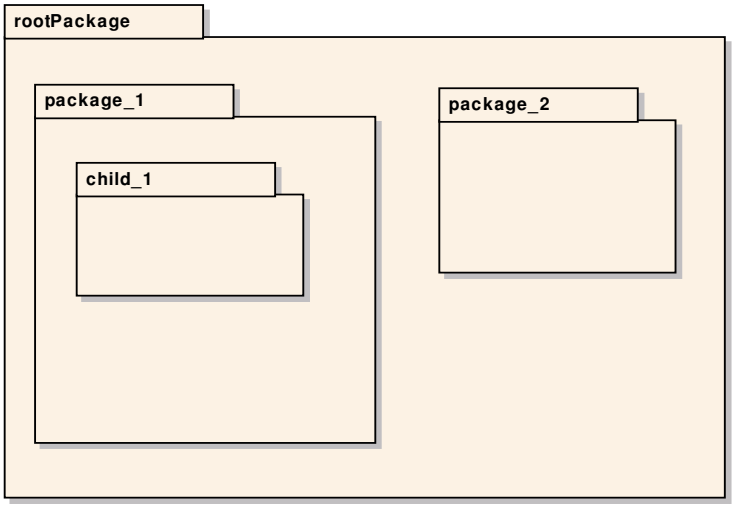

Figure 2.6: Example of a Package structure.

If the EP value of the `rootPackage` in figure 2.6 is to be 100 % there cannot exist any dependencies between `child_1` and the packages `package_2` and `rootPackage`.
A low EP value (many child-packages are used outside the package) implies that the package in question might have to be split in two or more packages.

Definition

The EP metric is the percentage of children that are not used outside of the package. If EP is 100 %, it means that none of the child-packages is used outside the package. And if EP is 0 %, it means that all children are used outside the package in question.

3. SELECTION OF SOURCE CODE

To be able to evaluate metrics and metric tools on a given code base, the code has to be evaluated beforehand, otherwise the measurements won't give significant results. In the introduction part good and bad code is defined, and this definition was held in mind when the selection of source code was made. The good quality code was selected on the basis of the author who wrote it. Authors that were primary selected were developers that are recognized, by the object-oriented community, as good quality code designers. The search for bad quality code was concentrated on open source and school projects where no or little effort were made in terms of code design.

3.1 Good Code

3.1.1 The Java Collections API

This source is found in J2SDK 1.4.2 in java.util and is written by Joshua Bloch. Bloch is a senior staff engineer at Sun Microsystems Java Software Division, where he works as an architect in the Java language group. Earlier Bloch worked as a Senior Systems Designer at Transarc Corporation in Pittsburgh Pennsylvania, where he designed and implemented many parts of the Encina distributed transaction-processing system [35]. He is also the author of *Effective Java*. As he is a recognized good programmer, writing quality-designed code, the collections API will be a good source code for measures.

3.1.2 Fitnesse

Fitnesse is a fully integrated standalone wiki, and acceptance-testing framework. It is an open-source project administrated by Robert C. Martin and Micah D. Martin. Robert C. Martin has been a software professional since 1970. He is CEO, president, and founder of Object Mentor Inc., a firm of highly experienced software professionals that offers process improvement consulting, object-oriented software design consulting, training, and development services to major corporations around the world. He has published a number of books in the area of OO-design, e.g. [32], and he writes the *Craftsman* column for *Software Development* Magazine [http://www.objectmentor.com/].

The source code for Fitnesse is available at: *https://sourceforge.net/projects/fitnesse/*

3.1.3 Taming Java Threads

This is an open-source threading library that expands Java's threading capabilities written by Allen I. Holub. Holub is a computer professional with 24 years experience in all aspects of computer technology and software engineering, ranging from OO design and Java programming, to hardware design, operating system and compiler design and construction, application programming, and UI design. Allen's primary focus in the last few years has been on helping companies be successful in their software-development efforts.

The source code is included in the book *Taming Java Threads* written by Allen Holub and can be downloaded at: *http://www.holub.com*

3.1.4 JDOM

JDOM is an API that provides a way to represent an XML document for easy and efficient reading, manipulation, and writing. Jason Hunter author of "Java Servlet Programming" (O'Reilly) who in 2003 received the Oracle Magazine author of the year award manages the JDOM project together with Brett McLaughlin. Brett is the author of "Java and XML" (O'Reilly) and he works at Lutris technologies where he specializes in distributed systems architecture. In 2001 JDOM was accepted as a Java Specification Request (JSR-102) by the Java Community Process (JCP) witch means that JDOM will be included in the Java Platform. Members of the JCP Executive Committee supporting the effort include Apache, Borland, Caldera, Cisco, HP, IBM, and Sun.

The source code for JDOM is available at: *http://www.jdom.org/*

3.2 Bad Code

3.2.1 BonForum

BonForum is an open-source chat application. It is described fully in the book, "XML, XSLT, Java and JSP" written by Westy Rockwell. The book is supposed to be a case study in XML, XSLT and JSP and the author wrote BonForum for this purpose. The book does not cover object-oriented design and the program is only written to try out different technologies, there is not a single UML diagram that explains how the program is designed. The author admits that it is not the purpose of the book or the program to explain the design and he also admits that huge chunks of code need to be refactored.

The source code can be downloaded at: *http://sourceforge.net/projects/bonforum/*

3.2.2 Student Project

This code was collected from a course at Uppsala University. The program consists of a user interface that could be applied to a chat program. The chat code is not included in this system since it was given to the students. Three different students with limited experience of Java implemented this code, and they admit that they did not consider object-oriented design when constructing the program. The source code was mailed to us and it's not available for download.

4. SELECTION OF METRIC TOOLS

This section covers the selection of software metric tools that is used in the experimental part of the project. Every tool has a short description on what they are measuring and where they can be collected; there is no description on how to use them included in this thesis, that is up to the readers to learn on their own. Both commercial and open source tools were tested and evaluated.

4.1 Tools Evaluated

The tools were tested on a simple program so that there metric calculations easily could be verified. The tools also measure some other metrics, but only the metrics selected and defined in chapter 2 are taken up here.

4.1.1 Metric 1.3.4

Metric 1.3.4 is a plug-in for Eclipse; Eclipse is an open source IDE developed by IBM, Borland, Red hat, Sybase among others and can be found at http://eclipse.org. To collect the metrics the analysed program does not have to be compiled since the tool measures on java files. Along with the tool comes a graphical dependency analyser that shows a dynamic hyperbolic graph of the package dependencies. After some testing several bugs in the tool were detected. For instance, when the LCOM where calculated for several classes the tool reported wrong value, sometimes it became larger than 1. This is an acknowledged bug, discussed on the tool homepage. If this bug is fixed in future version, the tool can be a potent metric collector. The tool can be collected at *http://metrics.sourceforge.net/*.

When it measures the extended cyclomatic complexity it ignores nested methods and it counts the ?: ternary operator when it is used in a return statement and in expressions, but not when it is used in an assignment expression. Examples:
`return (t==1 ? t : 0)`: counts.
`if( (t==1 ? t : 0)>=1 )`: counts.
`S += (t==1 ? t : 0)`: does not count.

The metrics measured by Metric:

- Number of Classes
- Number of Interfaces
- Number of Packages
- Afferent Coupling

- Efferent Coupling
- Instability
- Abstractness
- Normalized Distance
- Depth of Inheritance Tree
- Number of Methods
- Lines of Code
- Weighted Methods per Class
- Lack of Cohesion of Methods
- Number of Children
- Extended Cyclomatic Complexity

4.1.2 Optimal Advisor

Optimal Advisor is a package structure analysis tool developed by Compuware and the version used in this experiment is a free evaluation copy downloaded at *http://javacentral.compuware.com/pasta/.* Optimal Advisor gives a graphical view of the dependencies in a system from multiple levels of abstraction e.g. package or class level. This gives a good perspective of the dependencies in a small system but if a large system is examined the graphical view can get pretty messy and hard to interpret. But Optimal Advisor also has a dependency view that's strictly text based and that also displays the code fragments that are causing the dependency. The Java Virtual Machine (JVM) memory consumption is very high and for large systems the tool must be started with extra-allocated memory. In the full version metrics can be collected at byte code level (.class files).

The free evaluation version calculates the following metrics on java code:

- Number of Classes
- Number of Interfaces
- Number of Packages
- Number of Abstract Classes
- Efferent coupling
- Afferent coupling
- Instability
- Abstractness
- Normalized Distance
- Dependency Inversion Principle Metric
- Acyclic Dependency Principle Metric
- Encapsulation Principle Metric

4.1.3 JHawk

The JHawk metric tool is supplied by *Virtual Machinery* and was purchased from their homepage at *www.virtualmachinery.com/jhawkprod.htm*. It measures metrics at a system's package, class or method level on Java code. It doesn't have any graphical view of the metrics, only text, but it is possible to export the results of an analysed system to .xml or .csv files.

It measures the following metrics at a systems package, method or class level:

- Halstead Volume
- Halstead Length

- Halstead Vocabulary
- Halstead Difficulty
- Halstead Effort
- Maintainability Index
- Response For a Class
- Depth of Inheritance Tree
- Number of Descendant Children
- Lack Of Cohesion in Methods
- Cyclomatic Complexity

4.1.4 Team In A Box

This tool is also a plug-in to Eclipse. It collects metrics during program build and show warnings in the task list if threshold values are exceeded. It is also possible to export the metrics to html format for public display. The metrics are then showed both in tables and as histograms. The metrics can also be exported as csv format for further analysis.
The tool can be collected at *http://www.teaminabox.co.uk/downloads/metrics*.

The metrics measured by Team In A Box:

- Number of Statements
- Efferent Coupling
- Weighted Methods per Class
- Lack of Cohesion in Methods
- Cyclomatic Complexity
- Lines of Code in Methods

4.1.5 JDepend

This is an open source project that comes in an independent version and as a plug-in to Eclipse. The plug-in was tested here. The tool calculated the metrics on Java files and other than the calculated metrics it shows dependencies between packages and also shows a graphical view of the distance from main sequence. It can be downloaded at http://andrei.gmxhome.de/jdepend4eclipse/index.html.

The metrics measured by JDepend:

- Number of classes and interfaces
- Efferent Coupling
- Afferent Coupling
- Instability
- Abstractness
- Distance from Main Sequence
- Package Dependency Cycles

4.2 Metric Assignment

The next step after evaluating the tools is to decide what tool should measure what metric. In the evaluation process the tools were validated and compared (if they measured the same metric) to each other. This section is the result of these validations and comparisons and each metric covered by this

report is assigned a tool that measures it. A more comprehensive description of how the tools measure its metrics is also covered in this section. In table 1 below each metric is assigned a tool followed by a description.

Metric	**Product**
Lines Of Code (LOC)	**Metric**
Total lines of code in the selected scope. Counts only non-blank and non-comment lines inside method bodies.	
Halstead Metrics	**JHawk**
Measures all five metrics defined in section 2.2.2.	
Maintainability Index (MI)	**JHawk**
Measures both definitions presented in section 2.2.3 but it only uses cyclomatic complexity (CC) not the extended CC metric (ECC). JHawks CC metric doesn't consider switch-case statements or the ternary operator.	
Cyclomatic Complexity (CC)	**Team In a Box**
As the definition from 2.3.1, does not take in account the && and || conditional logic operators in expressions, counts the ?: ternary operator and switch-case statements. It also takes nested methods into account. Measured at method level.	
Extended Cyclomatic Complexity (ECC)	**Metric**
As the definition from 2.3.3, also count the && and || conditional logic operators in expressions and switch-case statements.	
Weighted Methods per Class (WMC)	**Team In a Box**
The sums of the cyclomatic complexity of the methods of the class being measured, where Team In a Box also measures the cyclomatic complexity, see section 2.4.1.	
Lack of Cohesion in Methods (LCOM)	**Team In a Box**
Measured with the Hendersson-Sellers definition. Only includes methods that access at least one field, and only includes a field if it is being accessed for at least one method, see section 2.4.2.	
Depth of Inheritance Tree (DIT)	**JHawk**
Measures the number of superclasses in the class hierarchy see section 2.4.3.	
Number of Descendant Children (NDC)	**JHawk**
This is a variant of the number of children metric (NOC) discussed in section 2.4.4 it calculates all children recursively not just the ones directly below a parent class.	
Response For a Class (RFC)	**JHawk**
Measures the response set for a class as defined in section 2.4.5, it treats constructors as methods.	
Coupling Between Objects (CBO)	**Team In a Box / Optimal Advisor**
The afferent and efferent coupling metrics **Ca** and **Ce** can be summarized as the CBO metric (see section 2.4.6. Optimal Advisor measures Ca and Ce at package level. The couplings (Ca and Ce) are not calculated based on the number of ingoing and outgoing classes, but rather on the weight of the dependencies. By using the dependency weight one takes into consideration that some classes are more	

tightly coupled than others. The dependency weight between two elements (packages or classes) is the amount of references from one to the other. Team In a Box measures the Ce at class level. The metric is a measure of the number of types the class that is being measured "knows" about, e.g. inheritance, interface implementation, parameter types, variable types etc.

Instability (I)	**Optimal Advisor**
Calculates I according to the definition in section 2.5.1, where Ca and Ce are also calculated with Optimal Advisor.	
Abstractness (A)	**Optimal Advisor**
See definition in section 2.5.1. The number of abstract classes, classes and interfaces are also calculated with Optimal Advisor.	
Normalized Distance (Dn)	**Optimal Advisor**
Measures Dn according to the definition in section 2.5.1, where I and A are measured by Optimal Advisor.	
Dependency Inversion Principle Metric (DIP)	**Optimal Advisor**
Collected at package level, see section 2.5.2 For definition.	
Acyclic Dependency Principle Metric (ADP)	**Optimal Advisor**
Measured as defined in section 2.5.3.	
Acyclic Dependency Principle Metric Recursive (ADPR)	**Optimal Advisor**
Measures the same thing as the ADP metric but it also takes sub-packages into consideration.	
Encapsulation Principle Metric (EP)	**Optimal Advisor**
Measured as defined in section 2.5.4.	

Table 1: Metric assignments.

5. EXPERIMENTAL STUDY

In this part of the report an experimental study was conducted, where the selected tools and source codes presented in sections 4 respective 3 were used. For each and every one of the metrics the minimum, maximum, mean, median and standard deviation were calculated on every source code.

Parts of the results of the experimental study are presented in tables and diagrams in section 5.1 (the complete results are presented in appendix B) followed by an interpretation and review of the results in sections 5.2 and 5.3. In section 5.4 a correlation analysis between several metrics is conducted, the results are presented in tables with an interpretation of the results in section 5.4.2.

5.1 Results

5.1.1 Size Related

Metric	Fitnesse	JDom	Collection API	Taming Java Threads	BonForum	Student Project
Lines of Code	9 189	7 012	830	1 602	3 451	549
Number of Packages	18	7	1	6	2	1
Number of Abstract Classes	17	2	4	0	0	0
Number of Interfaces	13	3	4	2	0	0
Number of Classes	339	45	10	26	18	15
Number of Methods	1 993	711	175	180	170	33
Maintainability Index	129	164	182	176	114	143
Maintainability Index (NC)	129	114	132	126	93	103
Halstead Length	47 563	23 886	3 728	4 562	13 665	2 693
Halstead Vocabulary	24 638	9 998	2 205	2 438	4 002	944
Halstead Volume	197 216	124 291	15 506	19 428	83 158	16 542
Halstead Difficulty	5 476	3 273	765	819	1 184	159
Halstead Effort	851 189	1 390 683	128 019	178 621	2 416 912	193 990

Table 5.1: Metrics measured on system-level

Halstead Metrics

The only Halstead metric that separates the badly designed systems from the good is the Halstead Effort metric. Although BonForum doesn't have the highest LOC value it has the highest effort value and the Student Project has higher effort than other systems with higher LOC values (Collection API, Taming Java Threads). An interesting observation is that the Halstead effort metric gives more information about a system when it's combined with the LOC metric, for instance Halstead Effort ÷ LOC.

Maintainability Index (MI)

The MI metric was collected at system-level and all the systems in the experimental study had values that indicated excellent maintainability according to the threshold values in section 2.2.3. The threshold values for the MI metric were not set to measure on Java source code, so if the MI metric is to be used in measuring Java systems new threshold values must be set. The lowest value however did correspond to the worst designed system (BonForum) despite that it was not the largest system examined

5.1.2 Method Level

Cyclomatic Complexity

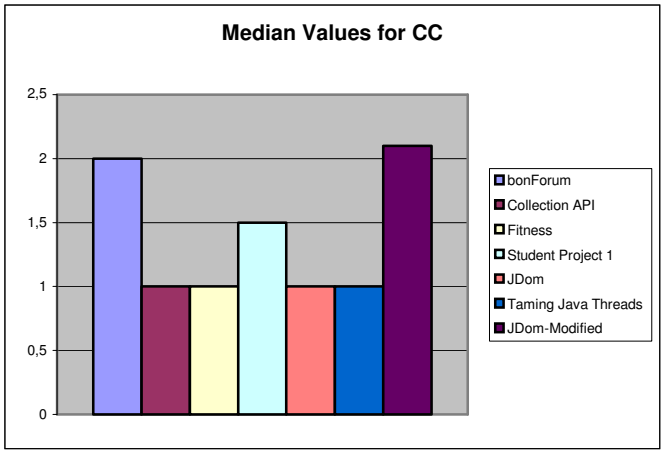

Diagram 5.3: Median of CC on all systems.

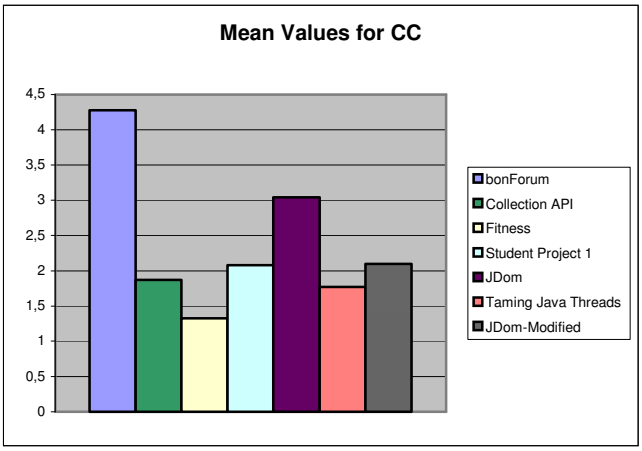

Diagram 5.4: Mean of CC on all systems.

This metric shows interesting result when looking at the different systems. In the Student project the mean of the CC is 2,08, the median is 1,50 and the standard deviation is 2,59. When comparing these results with the other projects, this is quite high. BonForum got the highest values of all, with a median of 2.0. The well-designed systems got a median of 1.0.

5.1.3 Class Level

Weighted Methods per Class (WMC)

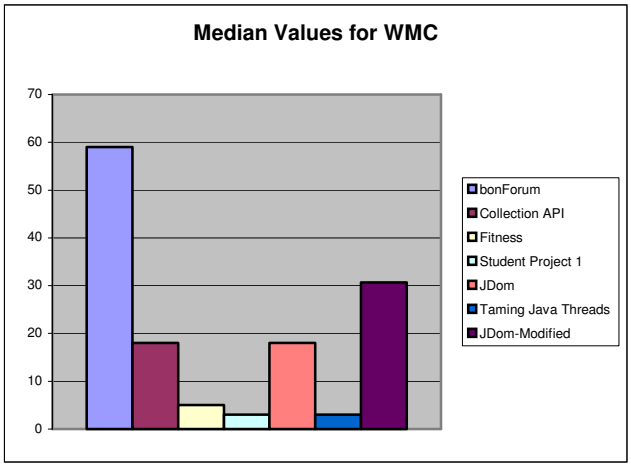

Diagram 5.1: Median of WMC on all systems.

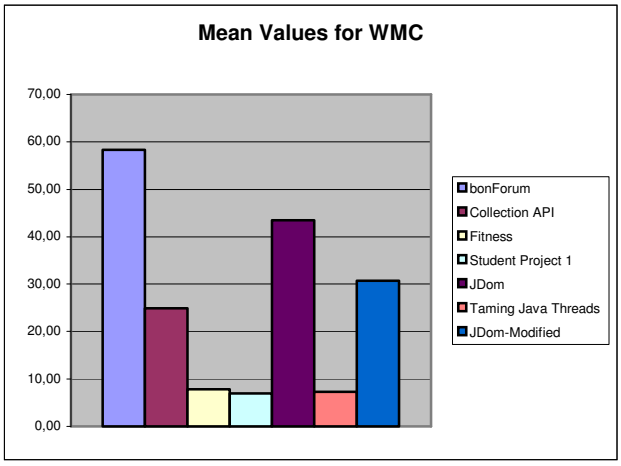

Diagram 5.2: Mean of WMC on all systems.

The BonForum system got the highest values of all and with a mean of 58,35. One of the JDom versions also got very high values, but as explained in section 5.2 this does not make the code more complex.

Lack of Cohesion in Methods (LCOM)

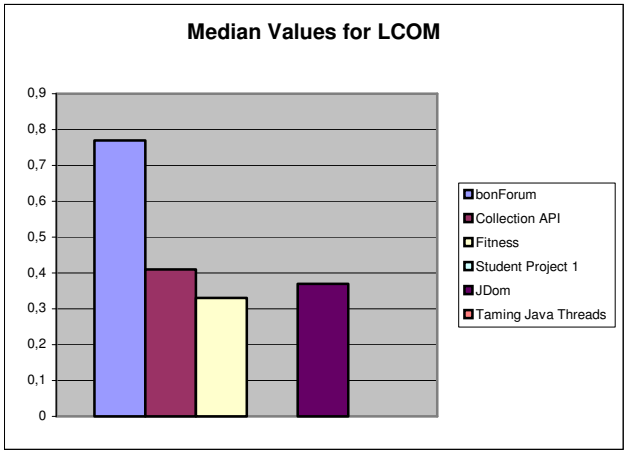

Diagram 5.5: Median of LCOM on all systems.

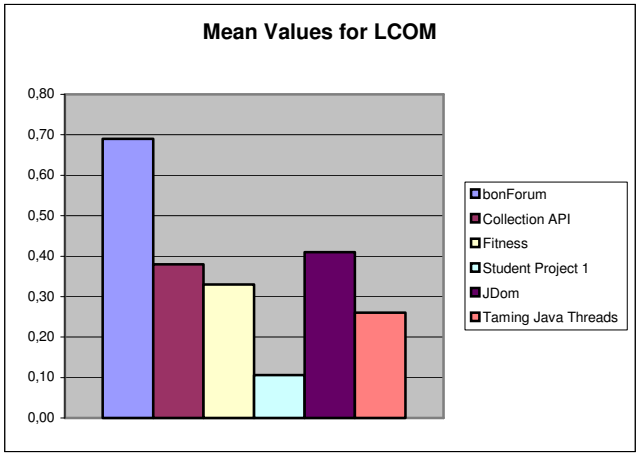

Diagram 5.6: Mean of LCOM on all systems.

In the well-designed systems the mean value are low, it is never over 0,5. In the BonForum system the mean is 0,69 and the median is 0,77.

Response For a Class (RFC)

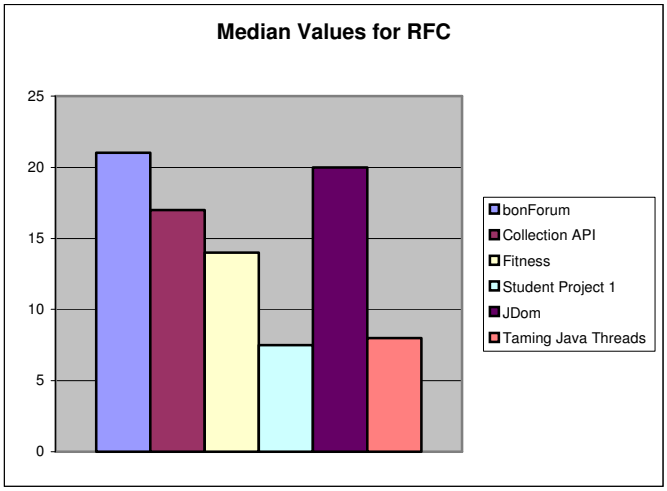

Diagram 5.7: Median of RFC on all systems.

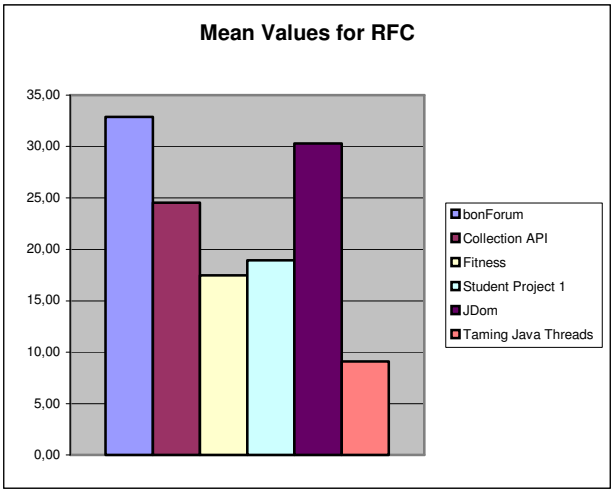

Diagram 5.8: Mean of RFC on all systems.

The BonForum system got the highest mean of 32,88. The Student project system got quite low values and this since the Student project only has 33 methods.

Depth of Inheritance Tree (DIT)

This metric is low overall when looking at the results. The well-designed systems got a mean that is over 0,0 and the badly designed systems got a mean of 0,0. The Collection API system got the highest values with a median of 1,0 and a mean of 1,23.

Number of Descendants Children (NDC)

Like the DIT metric, the NDC metric shows low results. The well-designed systems got a mean value that is higher than 0,0 and the badly designed systems got a mean of 0,0. Also here the Collection API system got the highest values.

5.1.4 Package Level

Afferent- and Efferent Coupling (Ca, Ce)

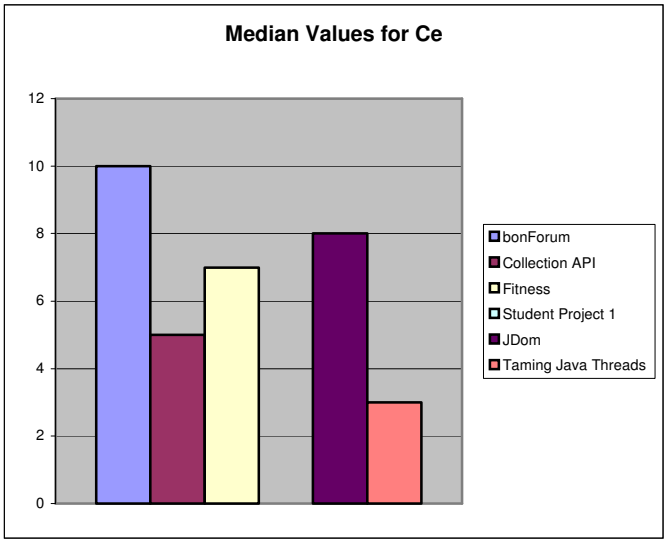

Diagram 5.9: Median of Ce on all systems.

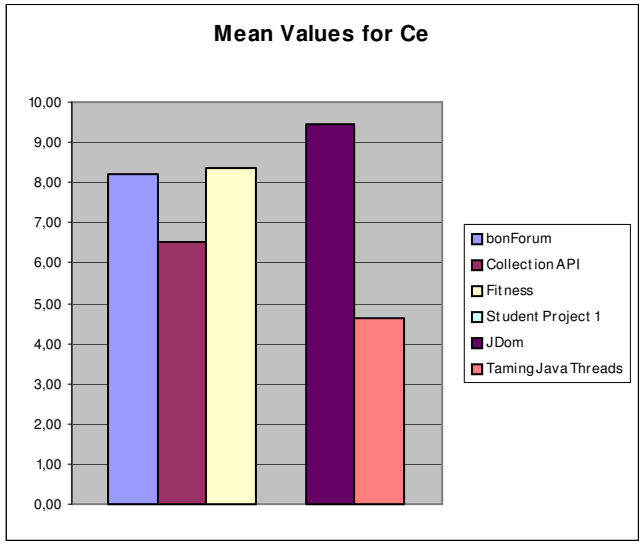

Diagram 5.10: Mean of Ce on all systems.

None of the Ca or Ce metrics serve as strong indicators of badly designed systems.

Dependency Inversion Principle Metric (DIP)

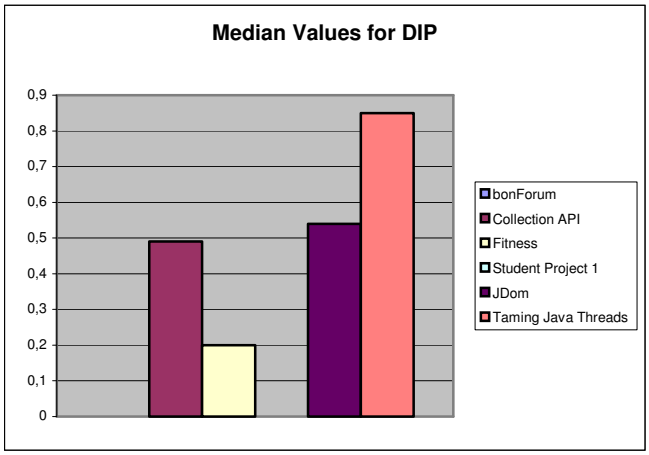

Diagram 5.11: Median of DIP on all systems.

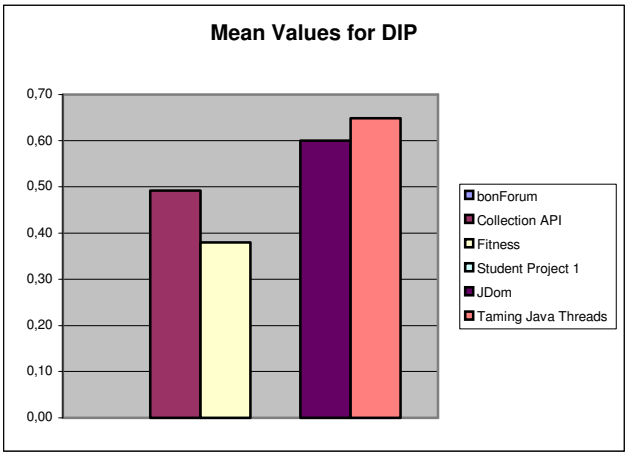

Diagram 5.12: Mean of DIP on all systems.

The DIP values in the badly designed systems are zero, in contrary to the good designed systems where the mean value of all the DIP metrics is above 0,3 (see diagram 5.12).

Instability (I), Abstractness (A) and Normalized Distance (Nd)

The badly designed systems consisted of too few packages to be measurable. Of the well-designed systems the Fitnesse, JDom and Taming Java Threads gave measurable results, and the Taming Java Threads got the highest values, with the mean of Normalized Distance at 0,32.

Acyclic Dependency Principle Metric (ADP)

This metric could not be collected from the badly designed systems. It was only measured on three of the good designed systems with the following median result: 0,67, 0,93 and 0,96.

Encapsulation Principle Metric (EP)

This metric could not be collected from the badly designed systems. It was only measured on three of the good designed systems with the following median result: 0,51, 0,88 and 0,88.

5.2 Interpretation of the Results

When looking at the cyclomatic complexity on the method level in the result tables (Appendix B) and in the diagrams, there is a difference between the well-written systems and the systems that's not so well written. The mean value is higher in the BonForum code than the others and with higher standard deviation.

There are two versions of the JDom system; the first one includes two methods with CC values of 358 and 162, which resulted in a mean of 3,04, and a standard deviation of 14,58. Even though the CC values were extremely high, the methods weren't hard to understand since they only checked an input name for XML compliance, so these two data weren't significant for the whole system. In the second version the implementation of the two methods were removed to get a more proper survey. The mean was now changed to 2,10 and the standard deviation was changed to 2,25, this is a big difference when considering that the system has 711 methods. Overall when looking at the systems with good design the CC is quite low, even if they got some methods with a high CC value the mean and standard deviation stays low that indicates that the most values are low. In the Collection API result table there is an interesting observation, the max of CC is higher than the max of ECC, and this deflection is explained by that the implementation of ECC, in the tool used, doesn't count Boolean statements within a return-statement.

At the class level, the depth of inheritance tree metric shows interesting results. In almost every system there exists no inheritance between classes, and the number of descendants children is also very low. This is very remarkable since this decreases the reusability in the systems. When looking at the other class level metrics there is a tendency of higher values in the BonForum project than in the other well-written systems.

It is interesting to look at the results from the Fitnesse system that is written by R. C. Martin himself to see if he has followed his own principles. As the results show he has done that. The Instability metric shows quite good result, the Abstractness metric is a bit low. The mean value of Normalized Distance is 0,14, which means that the most of the packages sits near the main sequence, the minimum is 0,01 and the maximum is 0,33, which is very good. Also the Dependency Inversion-, Acyclic Dependency- and Encapsulation Principles show good results.

5.2.1 A Closer Look at BonForum

The BonForum system, which in beforehand was known to have lower design quality shows interesting results. The WMC metrics has a mean of 58,35 with a standard deviation of 56,55, which is notable. After a closer look at the code these results can be explained. The system has 3451 LOC and the three biggest classes has 1128, 910 and 581 LOC which means that the system is based around only these three classes, the classes also includes very large methods. With these three big classes the understanability, maintainability and reusability decreases and it's clear that to improve the metric-results and hence increase the quality of the design, the system is in need of refactoring or some other changes in the code.

The MI metric in BonForum is the lowest one of all the systems; it shows a result of 114 for commented code. The Fitnesse system has an MI of 129 which is better and this though it consists of 9189 LOC, i.e. it is almost three times bigger than BonForum.

5.3 Experiment Review

The experiment consisted of too few systems to measure on, especially there were to few badly designed systems, in this one there were only two. To get more accurate results the code base has to consist of several more systems, and the systems has to be large since this report investigates metrics on object-oriented designed codes.

The depth of inheritance tree metric didn't get satisfactory measures since almost every system showed a lack of inheritance hierarchy, the same with number of descendant children.

Some of the source codes were implemented in just one package and therefore the package-level metrics on those sources didn't give any adequate information. Because of this, comparison between the package metrics is superfluous.

Since Halstead's and the MI metrics are not really Object-Oriented design metrics and the code base examined in this report only consists of known system-level designs i.e. there is no information of the

design quality on specific classes or methods, more empirical data must be collected on classes and methods with known design quality to get a better validation on the metrics.

5.4 Correlation Analysis

The reason for conducting a correlation analysis is to examine the relationship (correlation) between, in this case, software metrics. One simple way to check for correlation between two sets of data is to just draw a scatter plot and visually try to determine the relationship between the two sets. A more rigorous approach is to use statistical methods, this can be done in two ways: by generating measures of association that indicate the similarity of two data sets, or by generating an equation that describes the relationship. The scatter plot and the generating measures of association approaches will be used in this analysis. There are different ways to calculate the measure of association, in the case of software metrics the *spearman rank correlation coefficient* (SRCC) is a good method to use since it handles non-normally distributed data [4]. A correlation calculated with this method results in a linear correlation, i.e. it indicates if two sets of data follow each other in a linear manner. In the scatter plot approach two data sets are drawn in the same diagram with one set at the x-axis and the other at the y-axis. All points in the plot will now show if there is a clear linear trend or pattern between the two sets.

5.4.1 Spearman's rank correlation coefficient

The SRCC value is always between 1 and –1, where values near 1 indicate a strong positive correlation and values near –1 indicate a strong negative correlation. A value near zero indicates weak or no correlation. The SRCC (r_s) equation is as follows:

$$r_s = 1 - \frac{6\sum d^2}{n^3 - n}$$

Where d and n are the only unknowns. To get n just count the number of paired items in the two sets. To get d the two data sets must be ranked separately in either ascending or descending order (from 1 to n or from n to 1). Take the data from table 5.2 as an example of how the ranking procedure and the calculation of d are done. The two data column shows the relationship between the WMC metric and the LOC metric collected from ten different classes.

WMC (x)	Rank (Rx)	LOC (y)	Rank Ry	d (Rx-Ry)	d^2
1	1	31	1	0	0
50	7	184	8	-1	1
32	5	90	4	1	1
51	8	137	6	2	4
245	10	1128	10	0	0
34	6	142	7	-1	1
73	9	549	9	0	0
19	2	63	2	0	0
31	4	108	5	-1	1
21	3	86	3	0	0

Table 5.2: Two sets of data, their rankings and d values.

This yields an r_s value of 0,9515 (n=10 and d^2=8, taken from table 5.2). This indicates a strong positive correlation between the lines of code and the WMC metric of a class. Before the calculation of r_s is performed, a null hypothesis must be defined. The null hypothesis must be rejected or else the correlation is not significant. In this section the following null hypothesis is used:

There exists no correlation between the two data sets (r_s =0).

Performing a test of the level of significance on the spearman coefficient can reject the null hypothesis. This is done by inserting the r_s and n values into the following equation:

$$t = r\sqrt{\left(\frac{n-2}{1-r^2}\right)}$$

The t value acquired must be checked against a critical value found in a probability table of the t-distribution (see Appendix C) In the table t is checked against the level of significance and if t is larger than the value in the table the null hypothesis can be rejected at a level of significance found in the table.

5.4.2 Results of Correlation Analysis

The analysed metrics can be found in tables 5.3 and 5.4, where the value of the spearman's coefficient is given. Each metric was collected from 32 different classes in 6 different systems. Since all metrics measure something related to program code and its components, it's likely to expect that some correlation exists. Taking this statement into account when setting the limit of when to investigate or comment a correlation closer, a limit of a coefficient value higher than 0,75 is set.

Spearman's Coefficient: r_s-value						
	WMC	**RFC**	**LCOM**	**Ce**	**LOC**	**MI**
WMC	1	**0,7714***	0,4005	0,4536**	**0,9432***	-0,3405
RFC	-	1	0,4359	0,6504*	**0,7711***	-0,2982
LCOM	-	-	1	0,4111	0,3694	-0,0675
Ce	-	-	-	1	0,4601**	-0,0383
LOC	-	-	-	-	1	-0,4692
MI	-	-	-	-	-	1

Table 5.3: Spearman coefficient values for 6 different class-level metrics.

*Significant at the 0,05 % level

**Significant at the 0,5 % level

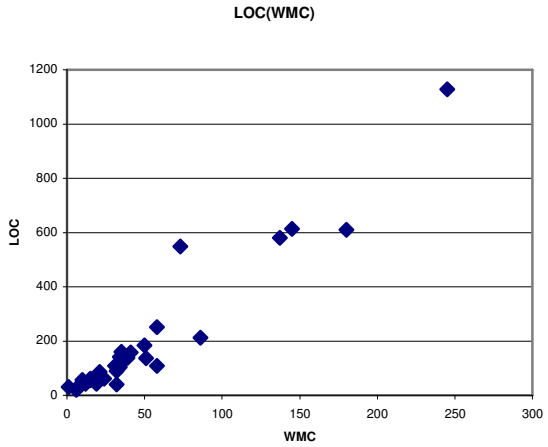

Diagram 5.13: Scatter plot of LOC and WMC.

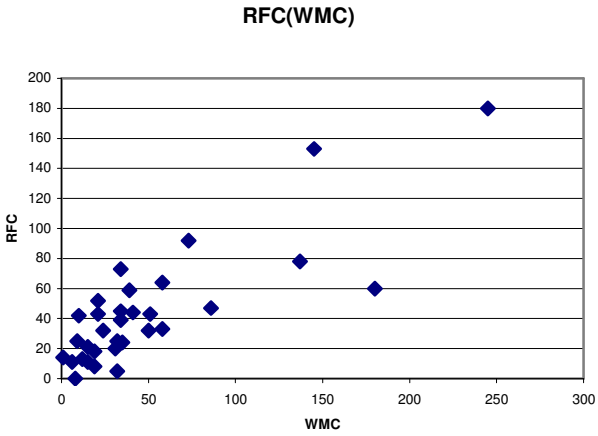

Diagram 5.14: Scatter plot of RFC and WMC.

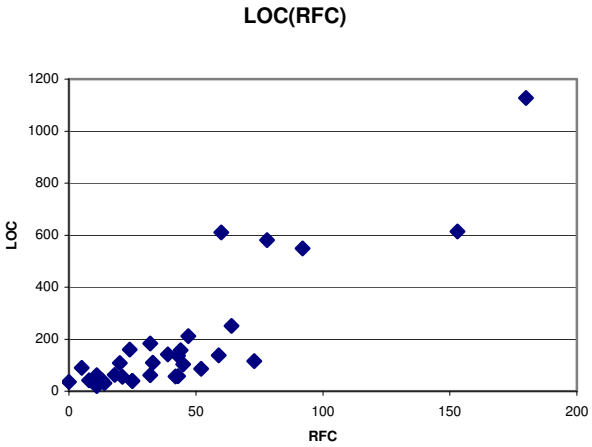

Diagram 5.15: Scatter plot of LOC and RFC.

The data in table 5.3 and the scatter plots in diagrams 5.13-5.15 show a strong positive correlation between the WMC, RFC and the LOC metrics. It also shows that the LOC metric does not correlate well with the Ce or the LCOM metrics.

Spearman's Coefficient: r_s-value							
	MI	**LOC**	**H-length**	**H-vocabulary**	**H-volume**	**H-difficulty**	**H-effort**
MI	1	-0,4692**	-0,5252**	-0,4043	-0,6005*	-0,3075	-0,7053*
LOC	-	1	0,7343*	0,7257*	0,7258*	0,5899*	0,6242*
Halstead length	-	-	1	**0,974***	**0,9885***	**0,7795***	**0,9006***
Halstead vocabulary	-	-	-	1	**0,9468***	0,7382*	**0,8359***
Halstead volume	-	-	-	-	1	0,7382*	**0,9245***
Halstead difficulty	-	-	-	-	-	1	**0,7528***
Halstead effort	-	-	-	-	-	-	1

Table 5.4: Spearman coefficient values for 7 different metrics calculated at class-level.

*Significant at the 0,05 % level

**Significant at the 0,5 % level

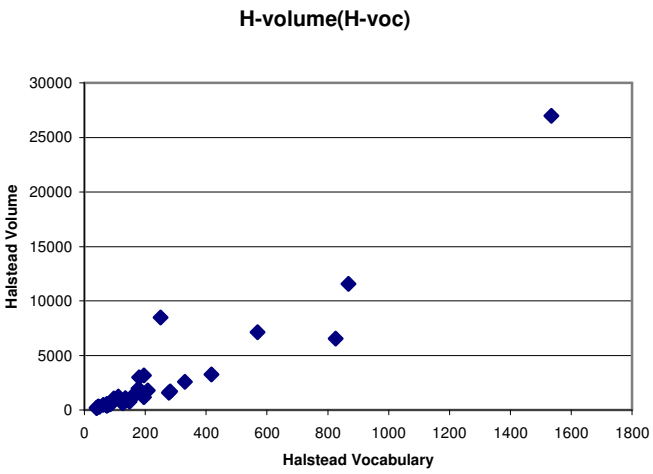

Diagram 5.16: Scatter plot of H-volume and H-vocabulary.

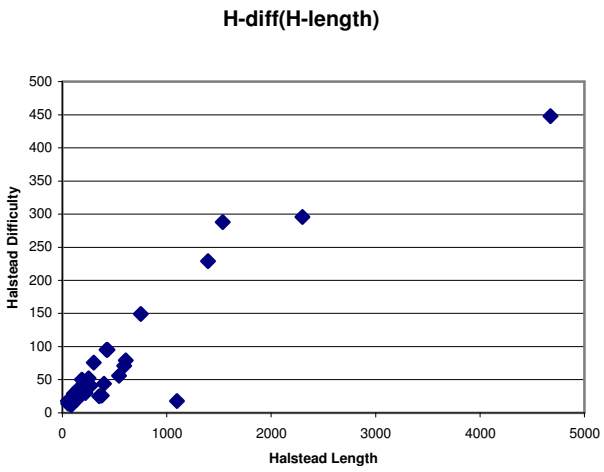

Diagram 5.17: Scatter plot of H-difficulty and H-length.

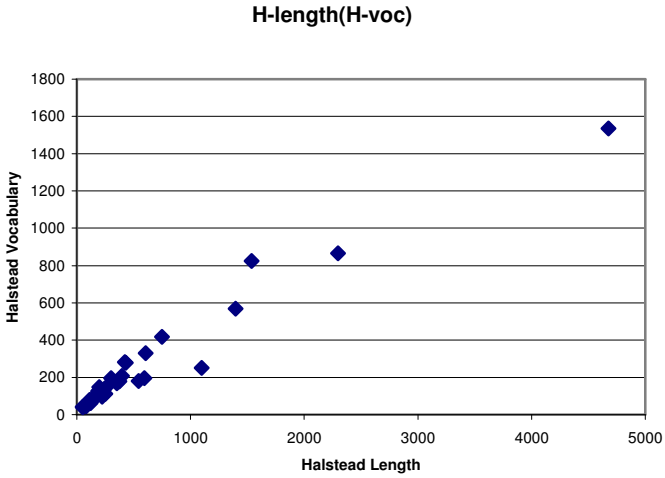

Diagram 5.18: Scatter plot of H-length and H-vocabulary.

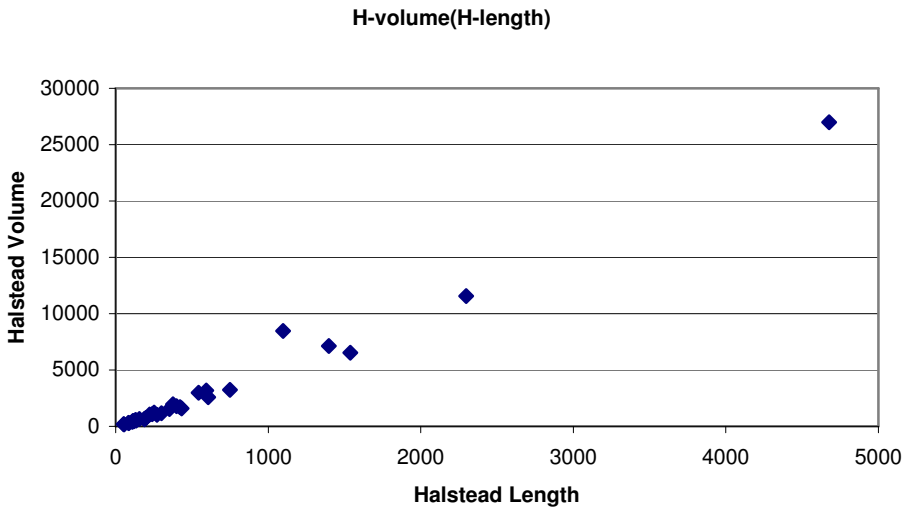

Diagram 5.19: Scatter plot of H-volume and H-length.

Very high positive correlations between all the Halstead Metrics are shown in table 5.4 and diagrams 5.16-5.19. This can to a degree be explained by looking at the definition of the Halstead Metrics in section 2.2.2, which shows that many of the metric-equations have very similar parameters.

Another interesting outcome of this analysis is that the MI metric doesn't seem to correlate significantly to any of the metrics that's it actually derived from (see section 2.2.3 for definition).

5.4.3 Correlation Analysis Review

Not all metrics covered by this report has been a subject of a correlation analysis. This is due to the lack of data collected for these metrics. Every combination of correlated metrics has not been calculated in this analysis, this is partly because of lack of time and partly because in some cases it is obvious that two metrics don't have anything to do with each other. It is important to take into account that this analysis was performed on arbitrary selected classes and methods, and if a similar analysis is performed on a different set of methods and classes different correlation values may appear. To further validate the different correlations between software metrics discussed in this report more empirical data may be needed.

6. SOFTWARE METRICS IN PRACTICE

In this section the practical usage of software metrics will be discussed by presenting examples and strategies. There are several issues that must be confronted before deciding to use software metrics in a system development process. The first things that must be clear are what attributes a specific metric has and what they tell a developer, some issues that need to be confronted are:

- Know what the metric measures and how it is implemented. Sometimes the definition of the metric differs greatly from how it is actually implemented.
- Learn what values indicate good design and what values don't. Beware of metrics that gives similar results from very different designed components; such metrics cannot be used easily because every new measure must be validated manually.
- Gather empirical experience about the metric to get an even better understanding of what values usually indicates good respective bad design. This way a developer can better separate bad values from good ones. It can be dangerous to look at other companies/developers result in this matter because the development process and/or software techniques can vary very much.

Although it is good to have experience with metrics it is not always necessary to know exactly how they are interpreted. This is a different approach, which can be useful if an empirically validated metric that indicates some aspect of good/bad design is applied on a software system. This way the metric is used to only indicates some quality aspect of the system and not provide any help on how this design issue is improved. This way the developers of a system must find the design problem and improve it somehow, and then check the metric value again to see if they succeeded to improve the design. This approach is typical for composite metrics (e.g. the Maintainability Metric (MI), see section 2.2.3), which can be hard to understand but can still serve as design quality indicators if they are properly validated.

6.1 Metric Evaluation

This section is a summary of the overall opinion about what metrics actually are important to use in practice. The metrics covered in section 6.1.1 are considered to be useful in practice because they are easy to collect and understand. In section 6.1.2 the metrics that are rejected as design predictors are discussed and section 6.1.3 presents metrics that might be of some use.

When can these metrics be collected? Well, the metrics covered by this report can be divided into two groups: during and before coding. The “during-coding” metrics require actual source code before they can be collected and the “before-coding” metrics can be collected on other bases, for instance UML diagrams. All metrics presented in section 2 can be collected during coding and they ones that doesn’t require source code are: Acyclic Dependency Principle, Dependency Inversion Principle, Encapsulation Principle, Depth of Inheritance Tree, Number of Children, Instability, Abstraction, and Coupling Between Objects.

6.1.1 Useful Metrics

These metrics are considered to be easy to understand and to give good directions on what needs to be done to fix a deviating value. Fixing problems pointed out by these metrics are often worth doing in a design perspective (not necessarily the same as a cost perspective).

Cyclomatic Complexity

This metric should be used as a tool for exposing too complex methods in a system. The most important value that needs to be checked is the highest in the whole system. When that value is reduced to an acceptable level, if it’s necessary, the second highest CC value can be fixed and so on. This way one can keep a low complexity in every method in a system, giving much higher understand- and maintainability. A very good technique for fixing too high CC values is refactoring [34]. This metric is useful during software development to warn when some method gets too complicated.

Halstead Vocabulary

This metric should be collected at method-level and it can be used for detecting methods that contain too many different unique operators and operands, which increases the complexity and maintainability of a method. This metric can be checked much like the CC metric discussed above and refactoring [34] is also a good technique for fixing high vocabulary values.

Lines Of Code

The Lines of Code metric gives a quick and simple view on how extensive a method, class or system is, and can be useful for scaling other metrics, see discussion in section 2.2.1.

Acyclic Dependency Principle Metric

Low ADP values in a system with many packages can be an indication on maintain- and extend-ability difficulties. Low values are fixed by changing the package structure e.g. changing place of packages or merge packages.

Dependency Inversion Principle Metric

Too low DIP values indicate lack of abstract types in a system, which could present difficulties to extend and reuse parts of the system and increased maintainability. A low DIP value is fixed by introducing more abstract types.

Encapsulation Principle Metric

If the package principles discussed in section 1.1.2 are followed by the developers this metric should be kept high at all times. Low EP values indicates bad package structures in a system and is fixed by either merging some packages into one single package or that some classes should be moved to a different package.

6.1.2 Rejected Metrics

Lack Of Cohesion of Methods

According to the definition a low value is good and a high value is bad, it ranges from 0 to 1. The Henderson-Sellers definition should be used instead of Chidamber and Kemerer's, see section 2.4.2. Even if there is a high value of LCOM in a class (the methods affect different fields) it isn't necessarily bad. For instance, if there is a abstract class called `geometricFigure` with three fields called position, colour and name and the class has setter and getter methods for these fields, then the LCOM will be one but the class is not badly constructed, instead this is a good class that could be extended and hence increase the reusability in the system. The LCOM metric gives too little information to be useful in a design quality investigation.

Depth of Inheritance Tree

This metric doesn't help developers to write well designed systems. It only says how deep the inheritance hierarchy is in the written project. It doesn't matter if you collect a number of 3 for instance; it just tells you that the length from this node to the root is 3. When developers write systems they don't have any need of very deep hierarchy, often the only inheritance is from Java libraries and those inheritance depths is not counted.

Number of Descendants Children

As the DIT metric this is not a helpful tool when developing software systems. The results collected from this metric can only state how many children a class have and that does not give any useful indication regarding object-oriented design quality.

Halstead Difficulty

Even if in the experimental study there were some differences between well-written code and bad code this metric is rejected because it is too hard to understand. If high values are collected as was done with the bad code, you don't know what to do with that. There is no obvious way to redesign the source code to improve the metric and hopefully the design.

Instability and Abstraction

These metrics are too complex for use in a developing process, if they are to be used the developers has to follow R. C. Martins Stable Abstraction Principle. There is no easy way to redesign the system so that the metrics give proper results.

6.1.3 Possibly Useful Metrics

Halstead Effort

In the experimental study there were notable differences between the well-written codes and the badly written ones, but the metric is hard to interpret and it doesn't say much about the design quality. But it can be useful when combined with LOC, i.e. Halstead Effort ÷ LOC, then it will be easier to interpret. One possible drawback with the metric is that it doesn't point out where in the system the flaw is.

Response for A class

According to the definition it doesn't count methods recursively, i.e. it only counts the methods that are directly called. A recursively version of this metric may be a better indicator of the understandability and complexity in the system since it would yield a more complete survey of the class. When collecting high values, i.e. the methods in the class calls a lot of other methods outside the class, it may be an indicator of misplaced methods.

Weighted Methods per Class

The CC metric is better to use than this one. If a high WMC is collected it says that either the class consists of many methods with low CC or few methods with high CC. The CC metric is used to locate which methods that cause high WMC values. The WMC metric can also be used as a size measure, but the LOC metric does that work better.

Coupling Between Objects

The CBO metric gives a good insight in where the in- and outgoing coupling resides in a system, and it is a metric that doesn't indicate that there is a design flaw but it shows where it might be difficult to

alter the code due to dependencies. Too high CBO values in a class can be an indication of misplaced methods, and can be reduced by moving the misplaced method to the class it mostly relates to.

Maintainable Index

This metric can be used in an ongoing software development effort if the developers understand what the metric measure really says, i.e. if a low value is collected the developers has to know what is wrong in the design and how to change it. The metric involves Halstead Volume, CC and LOC, which mean that the developer has to understand all of these metrics. It may be used best in situations where a company is about to buy a software system from another company, to see how maintainable the code is. As stated earlier new threshold values, than those in section 2.2.3, are needed that relates to Java-code.

Halstead Length and Volume

These metrics are size measures, but there is no point in using them instead of LOC. The LOC is much easier to understand and measure.

6.2 Integration Strategies

If a developer team has decided to use metrics it is important to start using them as early as possible in the software development process. It can be very costly (time-wise) if metrics are introduced too late and show that the existing code violates some metric threshold values. These design issues could have been avoided if the metrics where there from the start.

Many software developers today use some kind of software testing, e.g. unit testing. Tests are often part of a system's build process; this way testing for bugs in the source code is performed continuously throughout the whole development process. The same methodology can be used for testing/measuring the quality of a source code, by using metric threshold values as test directives. This way every metric is checked continuously and a developer gets instant feedback about any design critic situation that emerges in the code. Since these "metric tests" are not tests for error in the code they must be evaluated a bit different than bug related tests. An error in the code must of course be fixed as soon as possible while a design problem is nothing else but an indication that something should be fixed to improve the quality of the code. An evaluation of the results is needed: should this design flaw be fixed or can the construction of the code continue without fixing it? If the design flaw is considered to be trivial and is not to be fixed another problem occurs: the next time the metric test is performed on the code the same test will still point out the same design issue. To get rid of this behaviour some kind of filter must be applied to the metric tests. One place where such filter could be provided is in the metric tool that calculates the value of the metrics. This is a feature that most of the tools evaluated in this report lacked. A filter should at least provide functionalities such as: removing certain modules from a

specific metric test and changing the implementation of metric calculations (e.g. removing switch-case statements from the CC metric calculation).

Another approach on how to check for metric threshold values is if the programmer has a metric tool running in the background of his development environment, this way the tool can give a warning if he just violated any threshold value. This approach differs from the “testing approach” discussed above in the way that now the responsibility of checking for design flaws with metrics is in the hands of every developer not in the system build process.

6.3 Practical Aid Strategies

A real world situation when metrics can be useful is when a company is about to buy a source code from another company. If the system consists of hundred of thousands of LOC it will become very hard to determine the systems status, i.e. is the system maintainable and in good condition? Or is it hard to maintain and in need of redesign? If the latter is true it may lead to significant of costs for the buying company in the future. Here a measure of the system can give an overview of the system and point out the weak parts that could cause trouble and may be in need of redesign. Metrics can also be helpful when a company negotiates down a price when buying a system. Or it can be an advantage for the company that is selling the system if they can present metric-measures with good result; this assumes that the buying company is familiar with software metrics.

If a company outsources some part of the system development to a consulting-company they can measure the result and thereby grade it or get a quick insight on how well designed the code is. This will work best if the company first sets threshold values together with the consulting-company so that they have a frame to work within.

Another situation where software metrics can be useful is in a learning context, for instance when teaching object-oriented design in a programming course the student results can be measured and compared to each other. This could help the students to understand how different the design can be in a beforehand settled project, and it may also be a helping hand for the teacher to grade the systems.

7. CONCLUSION

So, can software metrics be used to describe a software system's object-oriented design quality? We believe that they can reflect some aspects of the design quality such as: complex methods/classes, package structure and the abstraction in a system. But object-oriented design is much more than that. None of the metrics covered by this report adequately measures the use of polymorphism or encapsulation (at class-level), which are two vital parts of the object-oriented paradigm. Another design area that's not measurable is the use of design patterns, which many programmers use to improve the design quality. Regarding the practical use of metrics to improve code design the same conclusion can be drawn; it can improve the design to some extent since the use of metrics can aid a developer to easily spot simple design flaws.

These, perhaps somewhat disappointing, conclusions aren't surprising to us since we, during this commission, discovered that not much has happened in the area of software metrics in the past years. Most of the global research was conducted in the 90s, which was when object-oriented languages had their big breakthrough. The subjectivity of software design quality makes it hard to develop metrics that fits with every developer's idea of what well designed code is. This dilemma, together with the fact that the use of metrics is time consuming and requires knowledge and experience, is probably why metrics aren't widespread in the programming community.

A very important issue when dealing with software metrics is experience; the more experience of measuring with metrics you have, the more the results will tell you. One idea when introducing metrics in a development process is first to start measuring at low level, perhaps when writing methods and then interpret the results but do nothing about it. When this is done over a time the developers can start measuring at class-, and package-level. After a while the developers hopefully get a feeling of what the results are telling them about the design and the developer team or company can set their own threshold values. When this state is reached it may be time to integrate the metrics fully and start design the systems around the metrics.

There has been many research efforts to combine metrics into a single, summary metric. This way one single value indicates the level of the design quality on some scale, this would be very useful for a developer since there is a risk of being overwhelmed with data when measuring with several different metrics, but this is maybe only a utopia. An attempt to describe the total maintainability in a system is done with the Maintainability Index metric; this seems to work to some extent. The problems occur when interpreting the measures, the metric doesn't tell you where in the design the problem is and hence doesn't tell the developer where redesign is needed.

The tools that were used in this report were either open-source or demo versions and can be considered a fragment of the tools on the market. Most of them worked fine, but we felt that they lacked some functionality. For instance when measuring a project it would be nice if the user could choose more freely what packages, classes and methods that are to be measured. Another problem was that there was very little information on how the metrics were implemented, which causes the user of the tool to figure it out manually. An example of this was the WMC and RFC metrics; none of the tools mentioned if constructors are treated as methods in a class.

An interesting observation in the experimental part was the lack of inheritance usage. This is remarkable since inheritance is one of the fundamental parts of object-oriented programming. It may be explained by that most developers inherit from classes and interfaces in the Java libraries and those inheritances weren't measured. Is it perhaps so that in practice it's hard to find concrete examples were deep inheritance hierarchies are needed? If that is the case then it is useless to measure inheritance in a design quality purpose.

Is there a place for software metrics in the future? Well, as long as no general design standard exists general metric threshold values will be difficult to determine. Locally however, rules for writing code can be constructed and metrics can be used to assure that the rules are followed. So metrics do have a future but they will always be limited by the subjectivity of software design.

A. Thesis Work Specification

A.1 Object Oriented Design Quality Metrics

A.1.1 Background

The basic idea behind OO design quality metrics is that it is possible to set a number of basic requirements that shall be fulfilled so that an object oriented API or software system is considered to be well designed. The more requirements that are to be fulfilled the higher the design quality. By formalizing the requirements and gather metrics for them it is possible to analyse code, and possible to grade developed systems in terms of the design quality. The foundation work is done in the area, and there exists some software that uses metrics to measure software code.

A.1.2 Task

The measure method can be applied on a code base partly to supervise the system during the development and partly to give a quick survey of how well designed an existing system is.

Citerus works with several different commissions where efficient measurements for OO design quality can come in use:

1. In develop-, project management- and architecture assignments there are a need to continuously supervise design quality as a part of the current quality work. Adequate metrics could be helpful in this work.
2. In mentor commissions, good metrics and tools to measure design quality could fill a pedagogical purpose.
3. In perusal commissions and similar a conceivable area of use could be to get a quick overview of the developed systems code and thereof be able to focus the effort to critical parts in an early stage.

The thesis work is to:

1. Find, gather and discuss theories and metrics that can be applied within the areas mentioned above.
2. Collect a code base with both good and badly designed code. The supervisor supplies some code, the remaining code will be a part of the project to find, and appropriate code could be open source projects that has been independently evaluated.

3. Evaluate and select from existing metric tools that measure code developed in Java. Both commercial and free tools should be included, especially interesting is of course tools that can be used commercially for none or a small cost. There exist open source tools that can be of interest, some of them can perhaps be used as a base for development of own metrics.
4. With help of one or more tools evaluate a code base with given quality to study how theories and tools can be used in practise. Especially interesting is to investigate how the result, i.e. the metrics, can be put together so they in a good way act as intermediary of interesting information about the code and if they correspond with the code quality. The information must be useful to the developer so that she or he can transform the metrics to improvement of the source code.
5. Make conclusions about:
    1) Which metrics carry any information at all if the code is good?
    2) What measure values usually indicates good or bad code?
6. Propose strategies on how:
    1) Analysis of source code with metrics can be integrated in an ongoing software development project.
    2) Metrics can be used as a practical aid in code- and architecture perusal on already developed systems.

A.1.3 Performance and Supervision

The commission will be done at Citerus AB in Uppsala. The major part of the work will be done in an office at Citerus. Supervision will be given by an active consultant, which means that the major part of the work will be done without the supervisor in the direct nearness. Follow-ups and meetings will be at evening time. Communication with the supervisor will be possible thru telephone and email. The work is estimated to take 20 weeks fulltime for two persons.

A.1.4 Time Plan

Week	Work Description
4 - 7	Read and gather information about the subject.
8 - 9	Gather suitable code base.
10 - 12	Investigate and select existing measurement tools, and theoretical metrics.
13 - 17	Measure the code base with relevant tools and theoretical metrics.
16	Easter holiday.
18 - 20	Evaluate the measurements.
21 - 22	Discuss the metric usefulness in an ongoing/finished software development.
	Put together theories on how to evaluate a code base.
23 - 25	Report writing.

B. EXPERIMENTAL RESULT TABLES

In the following tables the minimum, maximum, median, mean and standard deviation is presented for every metric on every source code.

The formula for standard deviation is:

$\sigma = \sqrt{\frac{\sum \left(x - \bar{x}\right)^2}{n-1}}$ Where n is the number of data, x is the actual data and $\bar{x}$ is the mean of all the data.

An example of standard deviation:
Consider a normal distributed set of data were the following samples are collected: 2, 3, 5, 7, and 9. The mean of the samples are 5.20 and the standard deviation is 2.86. This is to be interpreted as 2/3 of all the data in the set is in the range between 2.34 and 8.06, i.e. 5.20 ± 2.86.

To get significant values from the standard deviation the data has to be normal distributed but this is not the case in all of our data. Still the standard deviation is calculated for every metric and presented in the result tables, and this has to be taken in consideration when interpreting the data. An example of when the standard deviation is misleading (Taken from table B.1): The mean of the Efferent coupling is 244,47 at package-level and the standard deviation is 632,60, which doesn't make much sense.

System: Fitnesse					
Metric	**Min**	**Max**	**Median**	**Mean**	**S.D.**
Cyclomatic Complexity	1	41	1,00	1,32	1,16
Extended Cyclomatic Complexity	1	41	1,00	1,32	1,15
Weighted Method per Class	0	62	5,00	7,82	9,28
Lack of Cohesion	0	1	0,33	0,33	0,33
Depth of Inheritance Tree	0	3	0,00	0,58	0,75
Number of Descendant Children	0	70	0,00	0,58	4,41
Response For a Class	0,00	96,00	14,00	17,50	15,70
Efferent Coupling (class)	1,00	43,00	7,00	8,36	5,87
Efferent Coupling (package)	0,00	2665,00	97,00	244,47	632,60
Afferent Coupling (package)	8,00	3206,00	34,00	459,06	887,14
Instability	0,00	0,94	0,48	0,46	0,39
Abstractness	0,00	0,20	0,07	0,08	0,07

Normalized Distance	0,01	0,33	0,08	0,14	0,17
Dependency Inversion Principle	0,02	1,00	0,20	0,38	0,40
Acyclic Dependency Principle	0,58	0,96	0,93	0,82	0,21
Acyclic Dependency Principle Rec.	0,94	0,94	0,94	0,94	0,00
Encapsulation Principle	0,00	1,00	0,51	0,52	0,26

Table B.1: *Package-, class- and method-level metrics on Fitnesse*

System: JDom					
Metric	**Min**	**Max**	**Median**	**Mean**	**S.D.**
Cyclomatic Complexity	1	358	1,00	3,04	14,58
Extended Cyclomatic Complexity	1	358	1	3,16	14,53
Weighted Method per Class	0	665	18,00	43,47	95,68
Lack of Cohesion	0	0,94	0,37	0,41	0,36
Depth of Inheritance Tree	0	1	0,00	0,21	0,41
Number of Descendant Children	0	6	0,00	0,21	0,86
Response For a Class	0	153	20,00	30,28	34,34
Efferent Coupling (class)	1,00	31,00	8,00	9,47	6,18
Efferent Coupling (package)	18,00	359,00	57,50	133,00	138,08
Afferent Coupling (package)	0,00	64,00	35,00	30,33	26,00
Instability	0,22	1,00	0,88	0,76	0,31
Abstractness	0,00	0,33	0,11	0,15	0,13
Normalized Distance	0,09	0,09	0,09	0,09	0,00
Dependency Inversion Principle	0,02	1,00	0,54	0,60	0,42
Acyclic Dependency Principle	0,96	0,96	0,96	0,96	0,00
Acyclic Dependency Principle Rec.	0,96	0,96	0,96	0,96	0,00
Encapsulation Principle	0,33	1,00	0,88	0,77	0,27

Table B.2: *Package-, class- and method-level metrics on JDom*

System: JDom					
Metric	**Min**	**Max**	**Median**	**Mean**	**S.D.**
Cyclomatic Complexity	1	28	1,00	2,10	2,25
Extended Cyclomatic Complexity	1	27	1,00	2,41	6,65
Weighted Method per Class	0	196	16,00	30,71	42,15
Lack of Cohesion	0	0,94	0,37	0,41	0,36
Depth of Inheritance Tree	0	1	0,00	0,21	0,41

Number of Descendant Children	0	6	0,00	0,21	0,86
Response For a Class	0	153	20,00	30,28	34,34
Efferent Coupling (class)	1	31	8,00	9,47	6,18
Efferent Coupling (package)	18	359	57,50	133,00	138,08
Afferent Coupling (package)	0	64	35,00	30,73	26,00
Instability	0,22	1,00	0,88	0,76	0,31
Abstractness	0,00	0,33	0,11	0,15	0,13
Normalized Distance	0,09	0,09	0,09	0,09	0,00
Dependency Inversion Principle	0,02	1,00	0,54	0,60	0,42
Acyclic Dependency Principle	0,96	0,96	0,96	0,96	0,00
Acyclic Dependency Principle Rec.	0,96	0,96	0,96	0,96	0,00
Encapsulation Principle	0,33	1,00	0,88	0,77	0,27

Table B.3: *Package-, class- and method-level metrics on JDom without the 2 methods with the highest CC*

System: Collection API					
Metric	**Min**	**Max**	**Median**	**Mean**	**S.D.**
Cyclomatic Complexity	1	13	1,00	1,87	1,59
Extended Cyclomatic Complexity	1	8	1,00	2,00	1,54
Weighted Method per Class	1	84	18,00	24,87	23,20
Lack of Cohesion	0,12	0,56	0,41	0,38	0,19
Depth of Inheritance Tree	0	3	1,00	1,23	0,83
Number of Descendant Children	0	9	1,00	1,69	2,66
Response For a Class	1	55	17,00	24,53	17,20
Efferent Coupling (class)	0	14	5,00	6,53	4,56
Efferent Coupling (package)	0	0	0,00	0,00	0,00
Afferent Coupling (package)	0	0	0,00	0,00	0,00
Instability	-	-	-	-	-
Abstractness	0,44	0,44	0,44	0,44	0,00
Normalized Distance	-	-	-	-	-
Dependency Inversion Principle	0,49	0,49	0,49	0,49	0,00
Acyclic Dependency Principle	-	-	-	-	-
Acyclic Dependency Principle Rec.	-	-	-	-	-
Encapsulation Principle	-	-	-	-	-

Table B.4: *Package-, class- and method-level metrics on Collection API*

System: Taming Java Threads					
Metric	**Min**	**Max**	**Median**	**Mean**	**S,D,**
Cyclomatic Complexity	1	8	1,00	1,77	1,42
Extended Cyclomatic Complexity	1	10	1,00	1,95	1,72
Weighted Method per Class	0	46	3,00	7,27	9,26
Lack of Cohesion	0	0,82	0,00	0,26	0,33
Depth of Inheritance Tree	0	2	0,00	0,06	0,31
Number of Descendant Children	0	2	0,00	0,06	0,31
Response For a Class	0	33	8,00	9,10	8,38
Efferent Coupling (class)	1	14	3,00	4,61	3,12
Efferent Coupling (package)	0	17	7,00	8,40	6,43
Afferent Coupling (package)	0	20	4,00	8,40	9,84
Instability	0,40	1,00	0,55	0,62	0,27
Abstractness	0,00	0,29	0,03	0,09	0,12
Normalized Distance	0,00	0,60	0,29	0,32	0,24
Dependency Inversion Principle	0,05	1,00	0,85	0,65	0,44
Acyclic Dependency Principle	0,67	1,00	0,84	0,84	0,24
Acyclic Dependency Principle Rec,	0,67	0,67	0,67	0,67	0,00
Encapsulation Principle	0,50	1,00	0,88	0,83	0,21

Table B.5: *Package-, class- and method-level metrics on Taming Java Threads*

System: BonForum					
Metric	**Min**	**Max**	**Median**	**Mean**	**S,D,**
Cyclomatic Complexity	1	92	2,00	4,28	9,29
Extended Cyclomatic Complexity	1	104	2,00	5,12	10,78
Weighted Method per Class	0	245	59	58,35	56,55
Lack of Cohesion	0	0,95	0,77	0,69	0,29
Depth of Inheritance Tree	0	0	0,00	0,00	0,00
Number of Descendant Children	0	0	0,00	0,00	0,00
Response For a Class	0,00	180,00	21,00	32,88	44,90
Efferent Coupling (class)	1,00	21,00	10,00	8,21	5,59
Efferent Coupling (package)	-	-	-	-	-
Afferent Coupling (package)	-	-	-	-	-
Instability	-	-	-	-	-
Abstractness	0,00	0,00	0,00	0,00	0,00

Normalized Distance	-	-	-	-	-
Dependency Inversion Principle	0,00	0,00	0,00	0,00	0,00
Acyclic Dependency Principle	-	-	-	-	-
Acyclic Dependency Principle Rec,	-	-	-	-	-
Encapsulation Principle	-	-	-	-	-

Table B.6: *Package-, class- and method-level metrics on BonForum*

System: Student Project					
Metric	**Min**	**Max**	**Median**	**Mean**	**S,D,**
Cyclomatic Complexity	1	16	1,50	2,08	2,59
Extended Cyclomatic Complexity	1	16	2,00	2,24	2,71
Weighted Method per Class	1	73	3,00	6,94	16,53
Lack of Cohesion	0	0,85	0,00	0,11	0,30
Depth of Inheritance Tree	0	0	0,00	0,00	0,00
Number of Descendant Children	0	0	0,00	0,00	0,00
Response For a Class	6	92	7,50	18,94	26,71
Efferent Coupling (class)	-	-	-	-	-
Efferent Coupling (package)	0	0	0,00	0,00	0,00
Afferent Coupling (package)	0	0	0,00	0,00	0,00
Instability	-	-	-	-	-
Abstractness	0,00	0,00	0,00	0,00	0,00
Normalized Distance	-	-	-	-	-
Dependency Inversion Principle	0,00	0,00	0,00	0,00	0,00
Acyclic Dependency Principle	-	-	-	-	-
Acyclic Dependency Principle Rec,	-	-	-	-	-
Encapsulation Principle	-	-	-	-	-

Table B.7: *Package-, class- and method-level metrics on Student Project*

C. CORRELATION TABLE

	Level of Significance for a Directional Test				
	0.05	**0.025**	**0.01**	**0.005**	**0.0005**
	Level of Significance for a Non-directional Test				
df	0.10	0.05	0.02	0.01	0.001
21	1.72	2.08	2.52	2.83	3.82
22	1.72	2.07	2.51	2.82	3.79
23	1.71	2.07	2.50	2.81	3.77
24	1.71	2.06	2.49	2.80	3.75
25	1.71	2.06	2.49	2.79	3.73
26	1.71	2.06	2.48	2.78	3.71
27	1.70	2.05	2.47	2.77	3.69
28	1.70	2.05	2.47	2.76	3.67
29	1.70	2.05	2.46	2.76	3.66
30	**1.70**	**2.04**	**2.46**	**2.75**	**3.65**

Table C.1: Critical values of t.

D. REFERENCES

[1] Shyam R. Chidamber, Chris F. Kemerer, *A Metrics Suite For Object Oriented Design*, M.I.T. Sloan School of Management, 1993.

[2] Elaine J. Weyuker, *Evaluating Software Complexity Measures*, IEEE Transactions on Software Engineering, Vol. 14, Issue 9, pp. 1357-1365, 1988.

[3] Shyam R. Chidamber, Chris F. Kemerer, *Towards A Metrics Suite For Object Oriented Design*, OOPSLA'91, pp. 197-211, 1991.

[4] Norman E. Fenton, Shari L. Pfleeger, *Software Metrics A Rigorous & Practical Approach*, Second Edition, PWS Publishing Company, 1997.

[5] Ghassan Alkadi, Ihssan Alkadi, *Application of a Revised DIT Metric to Redesign an OO Design*, Journal of Object Technology, Vol. 2, Issue 3, pp. 127-134, 2003.

[6] Software Engineering Standards Committee of the IEEE Computer Society, *IEEE Standard for a Software Quality Metrics Methodology*, IEEE Std 1061-1998, 1998.

[7] Shyam R. Chidamber, Chris F. Kemerer, *A Metrics Suite For Object Oriented Design*, IEEE Transactions on Software Engineering, Vol. 20, Issue 6, pp. 476-493, 1994.

[8] T.J. McCabe, *A Software Complexity Measure*, IEEE Transactions on Software Engineering, Vol. 2, pp. 308-320, 1976.

[9] Wei Li, *Another Metric Suite For Object Oriented Programming*, The Journal of Systems and Software, Vol. 44, Issue 2, pp. 155-162, 1998.

[10] Geoffrey K. Gill, Chris F. Kemerer, *Cyclomatic Complexity Density and Software Maintenance Productivity*, IEEE Transactions on Software Engineering, Vol. 17, Issue 12, pp. 1284-1288, 1991.

[11] Ole-Johan Dahl, Kristen Nygaard, *How Object Oriented Programming Started*, http://heim.ifi.uio.no/~kristen/FORSKNINGSDOK_MAPPE/F_OO_start.html, 2004-06-04.

[12] Roger S. Pressman, *Software Engineering A Practitioner's Approach*, Fourth Edition, McGraw-Hill, 1997.

[13] Frederick T. Sheldon, Kshamta Jerath, Hong Chung, *Metrics for Maintainability of Class Inheritance Hierarchies*, Journal of Software Maintenance and Evolution: Research and Practice, Vol. 14, Issue 3, pp. 147-160, 2002.

[14] Aldo Liso, *Software Maintainability Metrics Model: An Improvement in the Coleman-Oman Model*, http://www.stsc.hill.af.mil/crosstalk/2001/08/liso.html, 2004-05-17.

[15] Daniela Glasberg, Khaled El Emam, Walcelio Melo, Nazim Madhavji, *Validating Object Oriented Design Metrics on a Commercial Java Application*, National Research Council of Canada, NRC 44146, 2000.

[16] David N. Card, Khaled El Emam, Betsy Scalzo, *Measurement of Object Oriented Software Development Projects*, Software Productivity Consortium, Herndon, Virgina, 2001.

[17] B. Henderson-Sellers, L. L. Constantine, I. M. Graham, *Coupling and Cohesion (Towards a Valid Metrics Suite for Object Oriented Analysis and Design)*, Object Oriented Systems, Vol. 3, pp. 143-158, 1996.

[18] Letha Etzkorn, Carl Davis, Wei Li, *A Statistical Comparison of Various Definitions of the LCOM Metric*, The University of Alabama, Huntsville, TR-UAH-CS-1997-02, 1997.

[19] Radu Marinescu, *Measurement and Quality in Object Oriented Design*, University of Timisoara, 2002.

[20] Wei Li, Sallie Henry, *Maintenance Metrics for the Object Oriented Paradigm*, Software Metrics Symposium, 21-22 May, pp. 52-60, 1993.

[21] Robert C. Martin, *Object Oriented Design Quality Metrics an Analysis of Dependencies*, http://www.objectmentor.com/resources/articles/oodmetrc.pdf, 2004-06-04.

[22] David P. Darcy, Chris F. Kemerer, *Managerial use of Metrics For Object Oriented Software: An Exploratory Analysis*, IEEE Transactions on Software Engineering, Vol. 24, Issue 8, pp. 629-639, 1998.

[23] Robert C. Martin, *Design Principles and Design Patterns*, http://www.objectmentor.com/resources/articles/Principles_and_Patterns.PDF, 2004-06-04.

[24] Jarret Rosenberg, *Some Misconceptions About Lines of Code*, Fourth International Software Metrics Symposium, 5-7 Nov, pp. 137-142, 1997.

[25] Joshua Bloch, *Effective Java*, Addison-Wesley Pub. Co, First Edition, 2001.

[26] Don Coleman, Dan Ash, Bruce Lowther, Paul Oman, *Using Metrics to Evaluate Software System Maintainability*, Computer, Vol. 27, Issue 8, pp. 44-49, 1994.

[27] Mei-Huei Tang, Ming-Hung Kao, Mei-Hwa Chen, *An Empirical Study on Object Oriented Metrics*, State University of New York, Albany, 1999.

[28] Kurt D. Welker, Paul W. Oman, *Software Maintainability Metrics Models in Practise*, http://www.stsc.hill.af.mil/crosstalk/1995/11/Maintain.asp, 2004-06-04.

[29] Lionel C. Briand, John Daly, Victor Porter, Jurgen Wust, *A Comprehensive Empirical Validation of Design Measures for Object Oriented Systems*, Fifth international Software Metrics Symposium, 20-21 Nov, 1998.

[30] Lionel C. Briand, Jurgen Wust, John W. Daly, Victor Porter, *Exploring the Relationships between Design Measures and Software Quality in Object Oriented Systems*, http://www.sce.carleton.ca/faculty/briand/pubs/jss.pdf, 2004-06-04.

[31] Michelle Cartwright, Martin Shepperd, *An Empirical Investigation of an Object Oriented Software System*, IEEE Transactions on Software Engineering, Vol. 26, Issue 8, pp. 786-796, 2000.

[32] Robert C. Martin, *Agile Software Development*, Pearson Education Inc, 2003.

[33] Rachel Harrison, Steve J. Counsell, *An Evaluation of the MOOD Set of Object Oriented Software Metrics*, Vol. 24, Issue 6, pp. 491-496, 1998.

[34] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[35] *Bios for Contributing Authors*, http://java.sun.com/docs/books/tutorial/information/bios.html#jbloch, 2004-06-04.