Self-Calibration of Metrics of Java Methods

Tal Cohen Joseph (Yossi) Gil
Department of Computer Science
Technion—Israel Institute of Technology
Technion City, Haifa 32000, Israel

Abstract

Self-calibration *is a new technique for the study of internal product metrics, sometime called* "observations" *and calibrating these against their frequency, or probability of occurring in* common programming practice *(CPP). Data gathering and analysis of the distribution of observations is an important prerequisite for predicting external qualities, and in particular software complexity. The main virtue of our technique is that it eliminates the use of absolute values in decision-making, and allows gauging local values in comparison with a scale computed from a standard and global database.* Method profiles *are introduced as a visual means to compare individual projects or categories of methods against the CPP. Although the techniques are general and could in principle be applied to traditional programming languages, the focus of this paper is on object-oriented languages using Java. The techniques are employed in a suite of 17 metrics in a body of circa thirty thousand Java methods.*

1: Introduction

The study of software metrics is one of the most illusive prospects in software engineering. The major difficulty is in calibrating or even correlating an internal property of a software system, i.e., a metric, against an external property [3], such as maintainability, by means of a controlled experiment. The vast resources required for even a single software project precludes running it in a research laboratory setting; the cost to be incurred in comparing several such projects carried out in more or less equal settings is outright prohibitive.

This paper offers a different angle of attack on this Gordian knot. This angle, which we may call "self-calibration", is based on the hypothesis that professional programmers working in a more or less fixed settings, and in particular using the same programming language, will follow an implicit *common programming practice* (CPP). The CPP can be thought of as the culture of programming as created collectively by the community of users, educators, and leaders of a certain programming environment. The scope of the CPP is not defined within a single project or organization, but rather with respect to a set of widely available specimens of large programs. It refers to the standard practice and use of the language by recognized leading software manufacturers (such as Sun and IBM in the case of the Java programming language) and some of their flagship software artifacts, instead of the champion programmers within an organization.

In many ways, the CPP is similar to the concept of design patterns [4], in the sense that it captures the folk-lore of software manufacturing. CPP does not however reflect commonly used solutions to specific recurring programming problems. It is rather the global trend, of using language features in large programming projects.

0-7695-0918-5/00 $10.00 © 2000 IEEE

The CPP is therefore manifested in the statistical distribution of a wide variety of internal product metrics. Self-calibration amounts to using statistical methods to identify and analyze these distributions. The quality of a certain software project can then be evaluated by placing it on the graphs of distribution of these metrics.

In order to understand this better consider a metric such as the size of a routine. It is clear that with all other factors being equal, larger routines are more complicated. A calibration question then is to determine the extent by which an increase in size from 50 units of size (such as lines of code) to 100 units raises the cost of maintainability. The self-calibration method avoids this question by calibrating the size-metric against its relative frequency. The answer then that self-calibration provides is of the following sort: "Routines of size 50 are common in this kind of projects, and occur at frequency of 10%. Doubling the size decreases the frequency to 0.01%." Such a decrease in frequency would serve as a warning signal to the user of the method. Thus, in self-calibration, the interpretation assigned to a certain value of a metric is the frequency at which this value occurs in practice. That is to say, the probability of finding it in programs which follow the CPP.

The appliation of self-calibration is always against a suite of *numeric* metrics such as that of Henderson-Sellers [8], Chidamber and Kemerer [2], or Mingins and Avotins [19]. As argued so convincingly by Meyer [18], each of these metrics must have an underlying theory to justify selecting it from the infinite possible values which can be computed on software. Another requirement is a theory that ascribes a *monotonic* property to each metric, that an increase in the value of the metric would always lead to change in the same direction of the desirability of some external property. For example, there are strong theoretical reasons to believe that an increase in size would always lead to an increase in complexity.

The number of instance variables in a class is an example of a non-monotonic metric, since we believe that both too small and too high values are undesirable. Many other class-level metrics are non-monotonic. Therefore, this study limits its scope to *methods*, instead of whole classes. It is future research to extend self-calibration to non-monotonic metrics.

The second requirement for the application of self-calibration is the availability of a large input set, representative of the CPP, to provide a sound foundation for the statistical analyses. This requirement is hard to meet in languages such as C++ [21], in which the computation of anything but the most trivial metrics could not be achieved without accurately parsing the source code, which is largely unavailable in commercial programs. Maughan and Avotins [14] tackled the first aspect of this predicament in providing a tool-set for obtaining such metrics in Eiffel [16]. However, since Eiffel does not yet enjoy wide industrial acceptance it deserves, it cannot be used for self-calibration. Self-calibration was made possible only with the advent of Java [1] and its bountiful class file format [10, Chap. 4]. Not only the evaluation of metrics is made technically easier by a direct analysis of the class file, but it is also possible to collect metrics of commercial software systems. In principle, a similar approach could have been implemented in other languages relying on P-code [23] execution, such as Smalltalk [6]. The difficulty is that most P-code representations are impoverished, and in the case of Smalltalk non-strongly-typed.

Using self-calibration we were able to identify, with excellent confidence levels, a distinct Mandelbrot-like distribution pattern common to *all* numerical metrics used in this paper. Each such distribution is characterized by a single constant K. Based on the identification of this distribution law, we make the case that for many purposes the *logarithm* of each metric is more meaningful than its original value. The constant K can be used for scaling when several metrics are to be combined into one.

Based on these finding we are able to borrow from the discipline of psychology a visualization and analysis technique which is primarily used in personality assessment. The borrowed technique, which we call *method profiles*, uses a transformed and normalized coordinate system of numeric metrics to show deviations from CPP. By drawing the method profile of e.g., private methods, we identified their unique CPP.

Outline. Section 2 describes experimental settings. Section 3 uses cross-tabulation analysis to discover some important characteristics of Java CPP. The distribution of the numerical metrics values is investigated in Section 4. Section 5 discusses findings in a table of correlation coefficients for all numerical metrics. Method profiles are described and used in Section 6. Finally, Section 7 outlines directions for future research.

2: Experimental Setting

Our input database consisted of five software collections, spanning a total of over thirty thousand methods in nearly three thousand classes. A total of 17 different *metrics* were computed for each of the input methods. These indicators were of two kinds: *numerical* metrics, and *categorical* metrics, whose values cannot be expressed as numbers but rather as enumerated types [14]. In our case, the values of all numerical metrics were natural numbers. However, in general, certain numerical metrics could be allowed negative as well as non-integral values.

The data was gathered by independently analyzing each class file, using two mechanisms:

1. The class file was loaded into the JVM (Java Virtual Machine) and the Java reflection library was then used to obtain the signature information for each method defined in the class.
2. Our class file parser was then invoked to obtain further information on each method.

We now turn to a more detailed description of the input, the categorical metrics, and the numerical metrics which are the subject of this paper.

The input sets comprised four software libraries and one large application:

JDK *Runtime Library of Sun Java Development Kit 1.2.* This library (`java.*` packages) comprises the basic runtime services of Java programs, including I/O, Java beans, applets, utility classes, etc.

Swing *Java Foundation Classes (sometimes called JFC or "Swing") 1.1.* This library, which ships with JDK 1.2, comprises all `javax.swing.*` packages and provides high level GUI functionality.

HotJava *Sun HotJava web browser version 2.0.* This fairly large application demonstrates the use of 'pure-Java' technology in the implementation of a full-fledged web browser.

XML *IBM XML for Java 1.0* A collection of service classes for parsing and creating XML documents in Java.

CORBA *OMG basic CORBA classes for Java 1.2* These are the Object Management Group's classes to map the CORBA API to Java.

The class files in each of these five collections were analyzed as described above. *Inner* classes were included in our analysis, since, with the exception of the pointer to the containing object, they behave and are used just like any other classes. About 27% (761 class) of the input classes were inner classes. The number of methods defined in these classes was 4,391, which is about 14.5% of all methods. In addition, the input included 244 anonymous classes (9% of all classes) with a total of 657 methods, which are about 2.2% of all methods. Since our primary concern here is methods and not classes, methods from anonymous classes were included in the analysis as well, even though the usage of these classes is, by definition, ad-hoc.

Table 1 summarizes the absolute number of packages, classes, and methods in each of these software collections, as well as their relative weight in the sample. The total number of methods used was $30,304$. To the best of our knowledge, this is one of the largest software ensembles to be studied in the literature. From these, a total of 747 methods, or fewer than 2.5%, were found to be *native*, and hence excluded from our study.

Note that the relative weights of the packages in the input are not equal. JDK, Swing and HotJava together comprise 94% of the input data, measured either by the number of classes or the number of methods. In general, selecting good inputs for a common practice study is not easy. For example, in our sampling of Java code, we refrained from using a huge collection of some 5,000 applet classes, which were located by a web spider. These applets appeared to be quite ad hoc, and not reflective of an orderly Java software development process. Likewise, data on nearly 140,000 Java methods from the famous IBM SanFrancisco project was not used. An initial analysis indicated that this data exhibits very different characteristics than those of the other, more easily available data. Specifically, it appears that much of the implementation of SanFrancisco used a small number of boilerplate classes and methods.

Project	# Packages		# Classes		# Methods	
JDK	38	(43%)	918	(33%)	11,090	(37%)
Swing	17	(19%)	1,162	(41%)	11,530	(38%)
HotJava	25	(28%)	575	(20%)	5,705	(19%)
XML	6	(7%)	99	(4%)	1,280	(4%)
CORBA	2	(2%)	58	(2%)	699	(2%)
Total	88		2,812		30,304	

Table 1. Software packages used in the experiments.

Input selection is a difficult process, which requires balancing factors such as code availability, personal evaluation of quality, etc. We are currently engaged in extending this study to a wider variety of large Java software collections, which would provide an even more reflective sample of the common Java programming practice.

The three categorical metrics included in this study were:

- *Method Sort* (SORT) Java distinguishes a special kind of methods, called *constructors*, which are only called when an object is constructed. In contrast, *finalizers* are somewhat similar to C++ destructors, but are far less common. Finalizers are invoked just prior to an object being disposed by the garbage collector. All remaining methods are considered *plain*.
- *Access Level* (ACL) An orthogonal classification of methods is by their visibility or access level. The access level of a method is either *public*, *private*, *package*, or *protected*.
- *Abstraction Level* (ABS) Yet another classification of methods is by their abstraction level. Methods designated as *abstract* cannot have body, and must be overridden in descendant classes. In contrast, *final* methods must have body and cannot be overridden in descendant classes. All other methods are *concrete*. Note that the ABS metric is not entirely orthogonal to SORT, since constructors cannot be abstract or final.

There are at least two other categorical metrics which can be thought of as part of this framework, but are not reported here. A distinction could be made between `static` and non-static methods. Also, one could use the return type, as well as information gathered by data flow analysis to further classify *plain* methods as *inspectors* (also known as *selectors*), *mutators* (also known as *modifiers*), and even *revealers* (which are methods that allow direct modification of state from outside).

Fourteen numerical metrics, in three major groups were studied in this research.

Intrinsic Complexity (IS) These are metrics which pertain to the complexity of the computation carried out in the method itself and the difficulty in understanding the message source.

1. *McCabe [15] Cyclomatic Complexity* (MCC) A value of 1 indicates the method has no branches; it is executed in a "fall-through" linear manner. Higher values indicate a higher number of branches.
2. *Size in bytecodes* (BC) Note that in the absence of source code, this is the closest approximation we have to the widely accepted LOC metric. In fact, we argue that BC is in many ways preferable to

LOC, since the measure it gives of program size is independent of indentation and other formatting personal preferences.

3. *Number of Local Variables* (LV) The total number of local variables as allocated on the JVM stack. This includes the method parameters, as well as the implicit `this` parameter in non-static methods.
4. *Number of Modified Local Variable* (MLV) The number of local variables which the method can modify (write to) during execution.
5. *Mathematical Opcodes* (MathOp) The number of mathematical opcodes (including integer and float arithmetic) appearing in the compiled method.
6. *Instantiation Opcodes* (NewOp) The number of instantiation sites in the compiled code, i.e., the number of times `new` and similar opcodes are found.

Self Interaction (SI). This is the group of metrics concerned with the interoperability of the method with other members, such as methods, static methods and fields, of the class it is defined in.

7. *Messages Sent to This* (MT) The number of call sites in the code that send messages to `this`. Note that the number of call sites (here and in other metrics) is hardly an indicator of the actual calls made during the method's execution: a single call site may be executed numerous times (e.g., inside a `for` loop) or not at all.
8. *Static Messages* (SM) The number of call sites for static messages (i.e., the number of times the opcode `invokestatic` appears in the method's bytecodes).
9. *Parameters* (PARAM) The number of parameters the function accepts, excluding the invisible parameter `this` for non-static methods.
10. *Number of Accessed Self-Fields* (ASF) The number of fields, in `this`, that are potentially accessed (for read only) by the method.
11. *Number of Modified Self-Fields* (MSF) The number of fields, in `this`, that are potentially modified by the method.

Interaction with Others (IO). This group of metrics deal with the complexity of dependence of a method in other classes to perform its duties.

12. *Messages Sent to Others* (MO) The number of call sites in the code used for sending messages to objects other than `this`. Naturally, in some runs the resulting message can be a message to `this`, depending on the call target's nature (e.g., a local variable that can be assigned `this`).
13. *Number of Accessed Fields* (AF) The number of fields, in objects other than `this`, that are potentially accessed (for read only) by the method.
14. *Number of Modified Fields* (MF) The number of fields, in objects other than `this`, that are potentially modified by the method.

3: Analysis of Categorical Metrics

In this section we study the distribution of the categorical metrics in our sample. Table 2(a) is a cross table of the abstraction level (ABS) metric by the access level metric (ACL). From the last row of the table we see that the vast majority, over 75%, of methods are public. All other methods are distributed roughly equally between the three remaining categories: private, protected, and package. This phenomenon cannot be explained solely by the large weight of library code in our input, since in restricting the measurements to HotJava we find that 69% of all methods are public. Thus, it appears that the abundance of public access level is a typical characteristic of Java programming.

	Private	Protected	Package	Public
Final	11.1%	11.4%	13.1%	64.4%
Concrete	7.6%	9.5%	9.8%	73.1%
Abstract	0.0%	3.4%	2.2%	94.4%
Total	6.8%	8.9%	9.0%	75.3%

(a) Cross table of ABS by ACL

	Final	Concrete	Abstract
Private	4.1%	95.9%	0.0%
Protected	3.2%	92.5%	4.3%
Package	3.6%	93.7%	2.7%
Public	2.1%	83.8%	14.0%
Total	2.5%	86.3%	11.2%

(b) Cross table of ACL by ABS

Table 2. Cross tables of abstraction (ABS) vs. access level (ACL)

	Private	Protected	Package	Public
Finalizer	0.0%	75.9%	0.0%	24.1%
Plain	7.5%	9.1%	7.0%	76.4%
Constructor	2.5%	7.1%	21.9%	68.4%
Total	6.8%	8.9%	9.0%	75.3%

(a) Cross table of SORT by ACL

	Final	Concrete	Abstract
Finalizer	0.0%	100.0%	0.0%
Plain	2.9%	84.2%	12.9%
Constructor	0.0%	100.0%	0.0%
Total	2.5%	86.3%	11.2%

(b) Cross table of SORT by ABS

Table 3. Cross tables of method sort (SORT) by abstraction level (ABS) and by access level (ACL).

The finding that only about 9% of all methods have package access level is rather surprising in its indication of low encapsulation at the package abstraction level. A related phenomena is that less than 6% of abstract methods are not public. (Note that Java does not allow private methods to be abstract.)

The largest fraction of non-public access level occurs at final methods. This can be explained by final methods tending to be of low implementation level and hence requiring stronger encapsulation.

Examining the last column in this table we see that an overwhelming majority (86%) of all methods are "concrete", and that Java programs make minimal use (2.5%) of method finalization. Even in JDK, which must finalize many methods for security reasons, the fraction of final methods is small (4.8%). In contrast, in HotJava finalized methods are much less frequent (0.7%).

The dual of Table 2(a) is given in Table 2(b) which cross tabulates ACL by ABS.

Further insight into the usage of access level can be gained from Table 3(a), which cross tabulates method sort (SORT) against ACL. We see that a rather large fraction (21%) of constructors have package access level. In other words, many classes can only be instantiated from within the package. Combining this bit of information with the fact that most methods are public we may find here an indication of usage of the ABSTRACT FACTORY and FACTORY METHOD design patterns [4].

We also see that the use of finalizers is rare. In fact, only about one in a thousand classes includes a finalizer. Curiously, finalizers tend to be protected, but due to the small number of finalizers, the statistical relevance of this feature is debatable.

Table 3(b) cross tabulates SORT by ABS. There is not much surprising information in this table. In Java all constructors *must* be concrete. The small number of finalizers in our sample are all concrete, even though this is not mandated by the language.

4: Distribution of Numerical Metrics

In this section we study the distribution of our 14 numerical metrics. Table 4 gives some of the essential statistics of each of these metrics. These statistics were computed only for the $26,254$ non-abstract, non-native methods.

A metric which was widely studied in the literature is the number of parameters to methods. Meyer [17] argues that in a good object oriented design, the average value of this metric tends

Metric	Min	Max	# values	Mean	Median	Common	SD	$\frac{SD}{Mean}$
1. McCabe (MCC)	1	136	73	2.53	1	1	4.20	166%
2. Bytecodes (BC)	1	7566	654	51.48	18	5	133.47	259%
3. Local variables (LV)	0	65	35	2.96	2	1	2.69	91%
4. Modified locals (MLV)	0	63	23	0.58	0	0	1.40	242%
5. Math opcodes (MathOp)	0	204	69	0.86	0	0	4.06	474%
6. New opcodes (NewOp)	0	378	38	0.49	0	0	2.96	608%
7. Msg. to `this` (MT)	0	74	41	0.88	0	0	2.05	234%
8. Static msg. (SM)	0	211	35	0.38	0	0	2.10	556%
9. Parameters (PARAM)	0	19	17	1.07	1	1	1.33	124%
10. Acc. self fields (ASF)	0	25	20	0.72	0	0	1.21	168%
11. Mod. self fields (MSF)	0	42	29	0.46	0	0	1.41	310%
12. Msg. to others (MO)	0	268	92	2.37	1	0	6.36	268%
13. Accessed fields (AF)	0	72	25	0.31	0	0	1.18	380%
14. Modified fields (MF)	0	28	17	0.14	0	0	0.69	480%

Table 4. Essential statistics of numerical metrics.

to be small. In Eiffel [16] Base library, the average number of arguments is 0.4, while in Eiffel Vision (which can be thought of as the Eiffel equivalent of Swing), this number is 0.7. Lorenz and Kidd [11] report on only slightly higher numbers, ranging between 0.3 and 0.9, in a variety of Smalltalk projects. Table 4 shows that in Java this number is even higher. The maximal number of arguments to methods is 3 in Eiffel Base, and 7 in Eiffel Vision. In contrast, in our study of Java, we find that there is at least one method with as many as 19 (!) arguments. One can also surmise from the "# values" column that this is not a singular phenomena. In fact, 98 methods in our sample had 8 or more arguments. A possible explanation is that our sample size of circa thirty thousand methods was much larger than the $5,489$ methods found in Eiffel Base+Vision libraries together. We will revisit this point below after deriving a formula approximating metric value distribution.

Lorenz and Kidd also report that the average number of message sends in a method ranges between 5 and 20 in the various Smalltalk projects they studied. In our suite the number of message sends is the sum of the MT, SM and MO metrics. In total, we have that Java methods make an average of 3.63 method calls, which is much less than the corresponding Smalltalk values.

Table 4 also reveals a huge standard deviation (SD) as a phenomenon which sweeps all metrics. In fact, the SD is typically several times larger than the mean value. Another indication of skewness in metrics distribution is that the *minimum value* is equal to the *median value* in 10 out of the 14 distributions, and equal to the *common value* in 11 of them. Even in the remaining distributions, the median and the common are much closer to the minimum than to the maximum of the range. Similar behavior is exhibited by the mean statistics, which also tends to be close to the minimum.

These finding lead us to the non-surprising belief that methods obey a Zipf-like distribution [24], i.e., that there is a rather large number of "small" methods, and that the number of "big" methods decreases along a hyperbolic curve. This belief is also strengthened by examining the BC metrics in which we see that the average number of byte codes is around 50, where the median is 18, which roughly correspond to two or three source code lines. (It is also interesting to note that the most *common* bytecode value is 5, which is exactly the size of a standard "get" method such as `java.awt.Component.getName`).

In order to verify this, we examined more closely the distribution of methods' cyclomatic complexity (MCC), and the number of bytecodes. The results are as depicted in Figure 1. Both charts in the figure are drawn in a log-log scale. By doing so, we are able to simultaneously test all steps in the Tukey [22] ladder for analysis of distribution. (The exponential decay hypothesis, $y = K10^{kx}$, fifth on the Tukey ladder is being excepted here.)

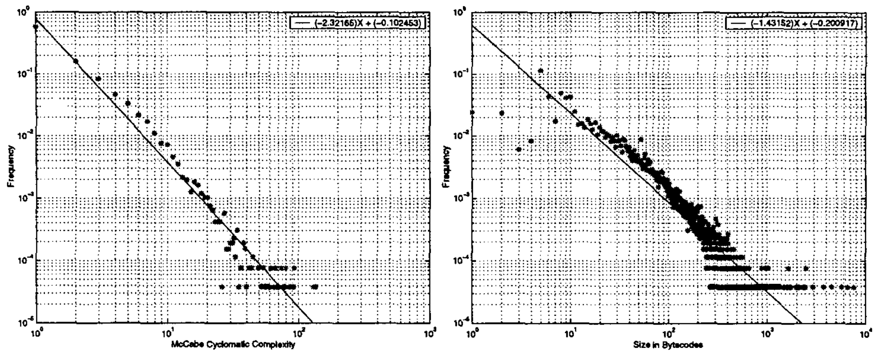


Figure 1. Distribution of MCC (a) and BC (b) metrics

The fact that most points at the lower right corner in the chart appear to fall on a horizontal line is no coincidence. These points correspond to those values of the metric that show only a small number of times in our inputs. The points on the lowermost horizontal line correspond to those metric values which show up exactly once, i.e, with frequency $1/26,254$. The next such line corresponds to a frequency of $2/26,254$, etc.

We see that in the log-log coordinate system, both distributions can be approximated fairly accurately by a straight line. The coefficients of these two lines are given in the respective charts. Thus, if $f(x)$ denotes the frequency in which a numerical metric shows the value x, we have that

$$\log f(x) = C - K\log(x), \tag{1}$$

where C and K are some constants, or alternatively

$$f(x) = cx^{-K}, \tag{2}$$

where c is some other constant ($c = 10^C$). Equation (2) is reminiscent of Mandelbrot law [12] for distribution of words in natural language text.

We applied similar analysis to all other metrics. Since the minimal value of these metrics is zero it was necessary to shift their values up by 1. For example, Figure 2 gives the distribution in the number of parameters and the number of modified self fields.

In the figure we see again that the distribution follows a straight line in the log-log scale. In fact, a similar distribution pattern shows up in *all* 14 numerical metrics. Table 5 summarizes the results of linear regression analysis in the log-log space of all numerical metrics.

We see that the values of K are usually in the range of 2 to 4. The only exception is the BC metric, for which $K = 1.46$. The value of K can be used as a weighting factor in combining several metrics into a single composite metric [9, Chap 5.4]. Statistically, the value of K is crucial in computing the essential statistics of a distribution of the sort of (2). It is not difficult to see that if the distribution of x obeys (2), then the expected value of x depends only on K (and not on the size of the sample)

$$E(x) = \frac{\zeta(K-1)}{\zeta(K)} - 1 \tag{3}$$

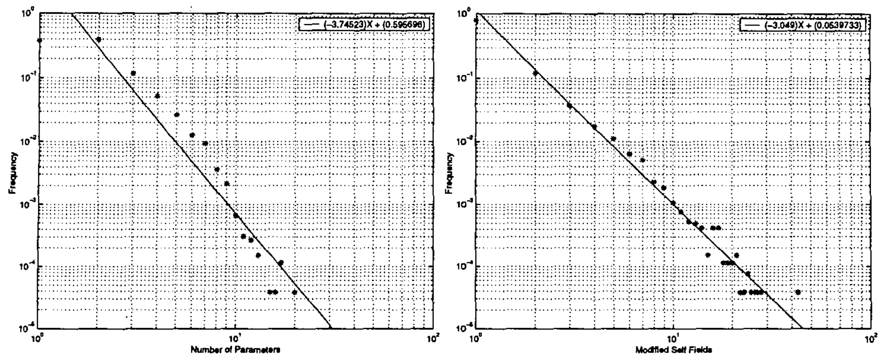


Figure 2. Distribution of PARAM (a) and MSF (b).

for all $K > 2$, where

$$\zeta(k) = \sum_{x=1}^{\infty} x^{-K}. \tag{4}$$

(The function $\zeta(\cdot)$ is nothing other than Riemann's Zeta function, but this is inessential to the derivation leading to (3).) When $K \leq 2$, $E(x)$ is unbounded, i.e., it increases with the size of the data. In our case, this phenomena is to be expected in the BC metric.

Moreover, if $K > 3$, we can write the SD $\sigma(x)$ of the random variable x as a function of K,

$$\sigma(x) = \sqrt{\frac{\zeta(K-2)}{\zeta(K)} - \left(\frac{\zeta(K-1)}{\zeta(K)}\right)^2}. \tag{5}$$

If $K \leq 3$, then the SD is not bounded and will depend on the sample size.

Unfortunately, (3) and (5) are not good predictors of the respective values of a distribution *approximated* by (2), and hence they are of mere theoretical interest. We learn however from (3) and (5) that the value of the intercept C is less interesting since it does not take part in the important statistics of the distribution.

Metric	K	C	R^2	p
1. McCabe (MCC)	2.32	-0.10	0.93	0
2. Bytecodes (BC)	1.43	-0.20	0.86	0
3. Local variables (LV)	2.79	0.36	0.79	7.7e-13
4. Modified locals (MLV)	3.02	0.00	0.90	5.5e-12
5. Math opcodes (MathOp)	2.01	-0.61	0.91	0
6. New opcodes (NewOp)	2.00	-0.99	0.80	4.9e-14
7. Msg. to `this` (MT)	2.55	-0.22	0.93	0
8. Static msg. (SM)	2.11	-0.95	0.80	3.2e-13
9. Parameters (PARAM)	3.75	0.60	0.91	3.9e-09
10. Acc. self fields (ASF)	3.54	0.36	0.94	3.1e-12
11. Mod. self fields (MSF)	3.05	0.05	0.97	0
12. Msg. to others (MO)	2.17	-0.09	0.92	0
13. Accessed fields (AF)	2.78	-0.39	0.90	6.8e-13
14. Modified fields (MF)	3.25	-0.26	0.97	7.6e-13

Table 5. Linear regression coefficients and statistics of numerical metrics.

Table 5 also gives the values of R^2 and p regression statistics. The R^2 statistic is a measure of the extent at which the variability of the dependent variable (the frequency of occurrence of a certain metric value) can be accounted for by the linear regression model. We see that at least 80% of this variability can be explained by the regression model. Typically, R^2 is at the level of 90% and sometimes even as high as 97%. Such values are extremely high for phenomena which cannot be attributed to some underlying physical law.

The p statistic is the probability of accepting the null hypothesis, namely that the variability in the dependent variable is not a result of the linear regression value. The values of p are *extremely* small, and at times they underflowed to zero by the underlying mathematical software [13] . These values look even smaller when compared to the 5% and 1% confidence levels commonly used in tests of this sort. With high confidence we conclude then that the distribution of metric values can be explained by a linear regression model.

In finding the regression constants, we did not try to ensure that

$$\sum_{m=0}^{\infty} f(m) = 1 \tag{6}$$

where $f(m)$ denotes the predicted frequency of a metric assuming a value m.

Let us now apply this linear regression model to find the frequency of methods with eight or more parameters. By using the appropriate constants from Table 5 into (2) we find that this frequency is 0.4% (to reach this value, we have to repeatedly apply (2) for all integer values of x greater than 8. The sum converges to 0.4%). In other words, in the abovementioned collection of Eiffel methods we would then expect about 21 such methods. The fact that there are none (a Poisson distribution model can be easily applied here) is significant, and indicates a meaningful difference in style between the two languages.

The high values of R^2 and the small values of p do not only reassure us in the linear regression model. We argue that the logarithm of a metric, or more precisely,

$$m' = \log(m+1), \tag{7}$$

where m is the original metric value, is more meaningful then the non-transformed value.

Consider for example the method size. It is not so important to know that the number of byte-codes is exactly (say) 1968. More important is the order of magnitude, which is reflected by the transformed value. This also holds with metrics with a smaller range of variability such as the number of parameters.

Metric Value	2	3	4	5	6	7	8	9	10
Predicted Frequency	6.44%	2.19%	0.95%	0.48%	0.27%	0.16%	0.11%	0.07%	0.05%

Table 6. Predicted frequency of methods by the number of parameters.

In checking Table 6, we see that an increase of 1 in the value of the metric from 2 to 3 reduces the predicted frequency by a factor of three. On the other hand, an increase from 9 to 10 reduces the predicted frequency only by a third, and is therefore much less significant. These aberrations are eliminated by using the transformed logarithmic metric value.

Table 7 is a revision of Table 4 where the statistics were computed using the transformed metrics values. The values presented in the table are after applying the transformation inverse. Doing so is tantamount (almost) to computing the geometrical instead of the arithmetical mean. Comparing Table 4 and Table 7 we see similar phenomena for all metrics: as expected, the mean value decreases by using the logarithmic transformation. Also, although the SD remains large, we see that it decreases not only in absolute terms, but also in relation to the new values of the mean.

Metric	Mean	SD	$\frac{SD}{Mean}$	Metric	Mean	SD	$\frac{SD}{Mean}$
1. McCabe (MCC)	1.69	1.39	82%	8. Static msg. (SM)	0.19	0.51	271%
2. Bytecodes (BC)	20.12	35.08	174%	9. Parameters (PARAM)	0.78	0.99	127%
3. Local variables (LV)	2.39	1.85	77%	10. Acc. self fields (ASF)	0.48	0.77	160%
4. Modified locals (MLV)	0.32	0.70	216%	11. Mod. self fields (MSF)	0.24	0.60	253%
5. Math opcodes (MathOp)	0.29	0.84	287%	12. Msg. to others (MO)	1.03	1.94	188%
6. New opcodes (NewOp)	0.25	0.59	236%	13. Accessed fields (AF)	0.16	0.47	294%
7. Msg. to `this` (MT)	0.53	0.88	166%	14. Modified fields (MF)	0.08	0.30	395%

Table 7. Essential statistics of the transformed metrics

5: Correlating Numerical Metrics (Omitted)

6: Categorical Metrics vs. Numerical Metrics

We now turn to the study of the values of the numerical metrics in the different categories as defined by the categorical metrics. To this end, "profile diagrams" are introduced here as an economical and effective means for visualizing and understanding the numerical metric characteristics of a single method or a group of methods.

Figure 3 gives the profile of public, protected, private and package level access methods. The ticks on the X-axis correspond to the metrics, with the standard numbering as used above (see e.g., Table 4). Recall that metrics 1–6 belong in the intrinsic-complexity group, metrics 7–11 form the self-interaction group, while the interaction-with-others group includes metrics 12–14.

The Y-axis uses the *transformed logarithmic* metrics value. In order to be able to describe multiple metrics using the same scale, we applied the standard linear scaling and shifting which brings the distribution to a mean 0 and a SD 1. A profile of a group of methods (or a single method for that matter) is drawn in the diagram by marking the values of all metrics and then connecting the points in each group of methods.

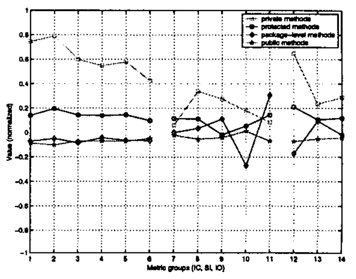


Figure 3. Profiles of the different categories of ACL.

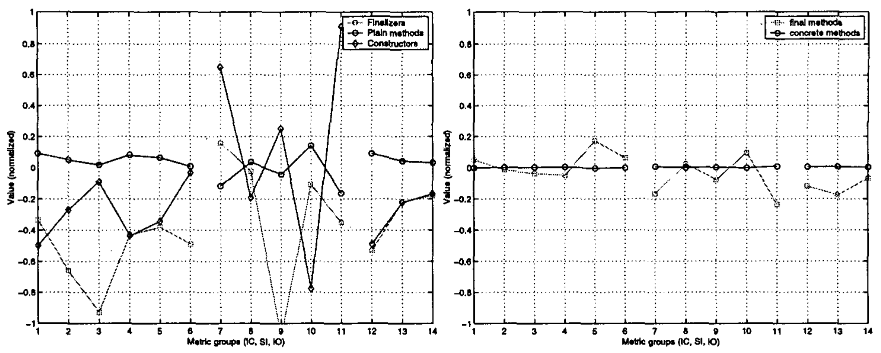
Let us first concentrate in the intrinsic complexity group of metrics. We see that private methods achieve higher values, by 0.4 to 0.6 SD units, compared to the entire collection of methods. This is in agreement with our intuition, which is that private methods will tend to hide the nitty-gritty of class implementation. A similar phenomenon, but to a lesser extent, is observed with protected methods. The intrinsic complexity of these methods is higher than that of the average by 0.15 to 0.2 SD units. In contrast, protected and public methods are marginally simpler than the average. In fact, the profiles of these two categories are very similar in the IC metrics group.

Moving on to the self-interaction group of metrics, we see that the differences between the four ACL categories here are much smaller. Private methods still tend to have higher metric values in this group. One can also notice that protected methods rank slightly higher than the average in this group. No such clear statement can be made about protected and package level access methods. However, in contrast with the previous group of metrics, this group distinctly separates this two categories. It is interesting to note that "package" methods have a small number of accessed self fields, and at the same time, a high number of modified self fields. This could indicate that these methods are used for granting classes in the package privileged access to internal fields.

In the third, interaction-with-others, group of metrics we again see that protected methods give higher metric values, and that private methods even exceed these. Since the bulk of the methods are public, it is hardly surprising that in this group, just as in all other groups, public methods scores are very close to the average.

Figure 4(a) shows the profiles of the different categories of SORT.

In the figure we see that finalizers tend to be significantly simpler than other methods in all metric groups, with the exception of only two metrics. It is somewhat surprising that finalizers send *more* messages to `this` than plain methods. This point requires further investigation. Constructors on



(a) SORT (b) ABS

Figure 4. Profiles of the different categories of SORT (a) and ABS (b).

the other hand tend to be simpler than most other methods, with few notable exceptions. It is not surprising that constructors tend to modify self fields, and require a large number of parameters.

The fact that by language definition, constructors are obliged to call (directly or indirectly) an inherited constructor, explains why constructors tend to have a slightly larger than the average `this` calls. Again, since the bulk of methods are plain, their behavior is close to the average.

Finally, Figure 4(b) shows the profile of concrete vs. final methods (most numerical metrics are inapplicable to abstract methods). Not much can be observed in this figure. However, it is apparent that final methods have lesser metric values in the IO group. More research is required to sort out this point.

7: Future research

This first work on self-calibration only served to demonstrate the technique. However, much more work must be done in order to apply the technique even in the limited domain of Java methods. We see several directions in which this work could and should be extended in the near future.

Metrics Suite Several well known metrics, such as LOC or Halstead Software Science metrics [7], were not included here. Similarly, the return type of a method (`void`, primitive, or polymorphic), or its classification as static or non-static were missing.

Data Set As large as our input was, we believe that there is still need to expand it to include other prominent examples of Java programming.

Analysis Techniques It is not entirely clear that metrics should be always shifted by the magic value 1, and further statistical investigation is required to sort this point out. Similarly, we need mathematical techniques for constraining the linear regression to satisfy (6). Also, we believe that the linear regression would be even better if intervals are used to classify some of the rarer metric values.

Application We intend to apply the technique to analyze profiles of not only categories of methods, but also of projects. Initial results in this direction show that the SanFrancisco methods exhibit a very distinct method profile.

More importantly, we note that self-calibration should not be limited to the study of Java methods. It would be interesting to apply it to classes. As mentioned above, this requires more research into the topic of calibrating non-monotonic metrics.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, 1996.

[2] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6):263–265, 1994.

[3] N. Fenton. *Software Metrics: A rigorous Approach*. Chapman and Hall, London, 1991.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing. Addison-Wesley, 1995.

[5] J. Y. Gil and A. Itai. The complexity of type analysis of object oriented programs. In E. Jul, editor, *Proceedings of the 12^{th} European Conference on Object-Oriented Programming*, number 1445 in Lecture Notes in Computer Science, pages 601–634, Brussels, Belgium, July20–24 1998. ECOOP'98, Springer Verlag.

[6] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison-Wesley, 1989.

[7] M. H. Halstead. *Elements of Software Science*. Elsevier Scientific Publishing Company, Amsterdam, 1977.

[8] B. Henderson-Sellers. Some metrics for object-oriented software engineering. In J. P. B. Meyer and M. Tokoro, editors, *Proceedings of Technology of Object-Oriented Languages and Systems: TOOLS6*, pages 131–139, Sydney, Australia, 1991. Prentice Hall.

[9] B. Henderson-Sellers. *Object Oriented Metrics*. Object-Oriented Series. Prentice-Hall, 1996.

[10] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publishing Company, Inc., 2nd edition, 1999.

[11] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice-Hall, 1994.

[12] B. B. Mandelbrot. An informational theory of the statistical structure of languages. In W. Jackson, editor, *Communication Theory*, pages 486–502. Betterworth, 1953.

[13] Using MATLAB. http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/using_ml.pdf, 1998.

[14] G. Maughan and J. Avotins. A meta-model for object-oriented reengineering and metrics collection. In *Proceedings of TOOLS Europe 1996*, 1996.

[15] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.

[16] B. Meyer. *EIFFEL: The Language*. Object-Oriented Series. Prentice-Hall, 1992.

[17] B. Meyer. *Reusable Software The Base Object-Oriented Component Libraries*. Prentice-Hall Object-Oriented. Prentice-Hall, 1994.

[18] B. Meyer. The role of object-oriented metrics. *IEEE Computer*, 31(11):123–125, November 1998.

[19] C. Mingins and J. Avotins. Quality suite for reusable eiffel software (qsres). Technical Report 6, Monash University, Caulfield Campus, 900 Dandenong Road, East Caulfield, Victoria 3145, Australia, 1995.

[20] H. Rombach. Design measurement: Some lessons learned. *IEEE Transactions on Software Engineering*, 7(3):17–25, 1990.

[21] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[22] J. Tukey. *Exploratory Data Analysis*. Addison-Wesley, Reading, Massachusetts, 1977.

[23] N. Wirth. From programming language design to computer construction. *Communications of the ACM*, 28(2), Feb. 1985.

[24] G. K. Zipf. Relative frequency as a determinant of phonetic change. *Harvard Studies in Classical Philology*, 40:1–95, 1929.