SELF-CALIBRATION OF METRICS OF JAVA METHODS
Towards the Discovery of the Common Programming Practice

RESEARCH THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE IN COMPUTER SCIENCE

TAL COHEN

SUBMITTED TO THE SENATE OF
THE TECHNION – ISRAEL INSTITUTE OF TECHNOLOGY

KISLEV 5762 HAIFA DECEMBER 2001

THE RESEARCH THESIS WAS DONE UNDER THE SUPERVISION OF DR. JOSEPH (YOSSI) GIL IN THE FACULTY OF COMPUTER SCIENCE. I AM INDEBTED TO DR. GIL FOR HIS SUPPORT AND HELPFUL SUPERVISION THROUGHOUT THIS RESEARCH.

THE GENEROUS FINANCIAL HELP OF THE MIRIAM AND AARON GUTWIRTH SCIENCE-BASED INDUSTRIES CENTER IS GRATEFULLY ACKNOWLEDGED.

Contents

1 Introduction **7**

1.1 Requirements for Self-Calibration 8

1.2 The Benefits of Self-Calibration 9

2 Experimental Setting **11**

2.1 The Input Sets 12

2.2 The Categorical Metrics 15

2.3 The Numerical Metrics 15

3 Analysis of Categorical Metrics **19**

4 Distribution of Numerical Metrics **25**

4.1 The Essential Statistics of Numerical Metrics 25

4.2 The Zipf-Like Distribution of Metrics 27

4.3 The Transformed Metric 32

5 Correlating Numerical Metrics **35**

6 Categorical Metrics vs. Numerical Metrics **37**

6.1 On Reading Method Profile Graphs 37

6.2 Profiling by Access Level 37

6.3 Profiling by Method Sort 39

6.4 Profiling by Abstraction Level 40

6.5 Profiling by Refinement 40

7 Conclusion and Future Research **43**

7.1 Benefits 43

7.2 Future Research 44

List of Figures

4.1 Distribution of the MCC (McCabe Cyclomatic Complexity) metric. 28
4.2 Distribution of the BC (size in bytecodes) metric. 29
4.3 Distribution of the PARAM (number of parameters) metric. 30
4.4 Distribution of the FA (fields access) metric. 31

6.1 Profiles of the different categories of ACL (Access Level). 38
6.2 Profiles of the different categories of SORT (method sort). 39
6.3 Profiles of the different categories of ABS (abstraction level). 41
6.4 Profiles of the different categories of REF (refinement indicator). 41

List of Tables

2.1 Software packages used in the experiments. 14

3.1 Cross table of ABS (abstraction level) by ACL (access level). 19
3.2 Cross table of SORT (method sort) by ACL (access level). 20
3.3 Cross table of ACL (access level) by ABS (abstraction level). 20
3.4 Cross table of ACL (access level) by SORT (method sort). 21
3.5 Cross table of ACL (access level) by STAT (static indicator). 21
3.6 Cross table of SORT (method sort) by ABS (abstraction level). 22
3.7 Cross table of REF (refinement indicator) by ABS (abstraction level). 22
3.8 Cross table of SORT (method sort) by REF (refinement indicator). 23

4.1 Essential statistics of numerical metrics. 25
4.2 Linear regression coefficients and statistics of numerical metrics. 29
4.3 Predicted frequency of methods by the number of parameters. 33
4.4 Essential statistics of the transformed metrics. 34

5.1 Correlations between metrics (top) and between transformed metrics (bottom). 36

List of CPP Findings

1 Dominance of public access 19
2 Finalizers are rarely used 20
3 Common use of package-wide global elements 21
4 Replacement is preferred over Refinement 22
5 Finalizers fail to use refinement 23
6 Java methods accept a relatively large number of parameters 26
7 Java methods send few messages 26
8 Metric distribution is skewed 26
9 Metric distributions have a linear regression model 27
10 The regression coefficient can be used for constructing composite metrics 28
11 Metric logarithm values are more meaningful than the original values 32
12 Large methods tend to operate on other objects 35
13 Local variables (and parameters) are used when accessing other objects 36
14 Complexity increases with decreased accessibility 38
15 Constructors are relatively simple 40
16 Using refinement decreases method complexity 42

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*. 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. Self-calibration strongly suggests that *transformed* metric values should be used for creating composite metrics. The transformed metrics are normally the $\log$ of the direct metric observations, and they are shown to be more meaningful than the original values.

Borrowing from the discipline of psychology, the research also suggests using *method profiles* as a visualizing and analysis technique which can be applied to the study of individual projects or categories of methods.

While both self-calibration and method profiles are very general and could in principle be applied to traditional programming languages, the focus of this research is on object-oriented languages using Java. The techniques are employed in a suite of ten numeric and five categorical metrics in a body of well over sixty thousand Java methods.

Chapter 1

Introduction

“Beware! The domain of metrics is deep and muddy waters,” is the first sentence Henderson-Sellers says on the topic in his book [14]. Indeed, the study of software metrics is one of the most illusive prospects in software engineering. Meyer [23] enumerates the following qualities as “key concerns”: correctness, robustness, extendibility, reusability, as well as compatibility, portability, ease of use, efficiency, timeliness, economy and functionality. These are all external factors, but Meyer clearly states: “What matters is the external factors, but they can only be achieved through the internal factors”. The major difficulty is in calibrating, or even correlating, an internal property of a software system, i.e., an internal metric (sometimes called *an observation*), against an external property [8], 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. The difficulties appear even more insurmountable when the focus is shifted from “product” metrics, i.e., direct measurement of the software, to “process” metrics, i.e., measurement of the process of producing the software [32].

This research offers a different sort of attack on this Gordian knot. Our approach, 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 shared culture of programming which is the collective creation of 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 [9], in the sense that it captures the folk-lore of software manufacturing. CPP does not however propound commonly used solutions to specific recurring programming problems. It is rather the global trend of using language features in large programming projects.

We believe that the CPP is 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, and in examining the resulting deviation from the CPP.

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 to this question 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 in this case 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.

1.1 Requirements for Self-Calibration

Two things are required in order to apply self-calibration. First, we must have a suite of *numeric* metrics such as that of Henderson-Sellers [13], Chidamber and Kemerer [5], or Mingins and Avotins [25]. As argued so convincingly by Meyer [24], each of these metrics must have an underlying theory to justify selecting it from the infinite possible values which can be computed on software. Moreover, it is also required that theory 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.

Not all metrics are monotonic. 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. When

values are plotted against their frequency for such non-monotonic metrics, the resulting distribution graph has a "hump", indicating that sometimes an increase in value correlates to an improvement in the external quality metrics, while sometimes a decrease in value is required for the same effect. It is future research to extend self-calibration to non-monotonic metrics.

Many class-level metrics (such as the number of heirs to a class, the number of methods defined in a class, etc.) are non-monotonic. Therefore, this study limits its scope to *methods*, instead of whole classes. There are also non-monotonic metrics in the scope of methods. For example, the number of lines of comment per line of code is a non-monotonic metric. We believe that the same is also true for the number of exception handlers per method; and there are naturally additional examples.

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++ [33], 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 [19] tackled the first aspect of this predicament in providing a tool-set for obtaining such metrics in Eiffel [21]. However, since Eiffel does not yet enjoy the wide industrial acceptance it deserves, it cannot be used for self-calibration. Self-calibration was made possible only with the advent of Java [2] and its bountiful class file format [15, 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 [36] execution, such as Smalltalk [11]. The difficulty is that most P-code representations are impoverished, and in the case of Smalltalk non-strongly-typed.

1.2 The Benefits of Self-Calibration

The achievements due to the application of the self-calibration technique which we report on here are multi-fold. First, we were able to identify, with excellent confidence levels, a distinct Mandelbrot-like distribution pattern *common to all numerical metrics* used in this research. Each such distribution is characterized by a single positive constant K, called the metric's *frequency coefficient*. Based on the identification of this distribution law, we argue that for many purposes the *logarithm* of each metric is more meaningful than its original value. In particular, we will see that correlation coefficients are accentuated using these logarithmic values. The constant K can be used for scaling when several metrics are to be combined into one. This scaling is required before weights can be applied to the

various metrics.

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 Chapter 2 describes the settings of the experiment, including input data, the ten numerical metrics, and the five categorical metrics. Chapter 3 uses cross-tabulation analysis to discover some important characteristics of Java CPP. The distribution law of the numerical metrics is discovered using a linear regression model in the log-log scale in Chapter 4. Chapter 5 discusses interesting findings in a table of correlation coefficients for all numerical metrics. Method profiles are described and used in Chapter 6. Finally, Chapter 7 concludes this work and outlines directions for future research.

Chapter 2

Experimental Setting

Our input database consisted of nine software collections, spanning a total of over 66,000 methods in 8,500 classes. While we do not have the source files for all of these classes, estimates based on the sources that *are* publicly available indicate that this colossal collection of software comprises well over 2.5 million lines of Java source code. A total of over 40 different *metrics*, or *observations*, were computed for each of the input methods. Of these, the 15 most important and indicative metrics were chosen for detailed analysis, based on previous research in the field (e.g., [5], [13], [14], [16], [20], [21], and [24]). These indicators were of two kinds: *numerical* metrics, and *categorical* metrics, whose values cannot be expressed as numbers but rather as enumerated types [19]. The two kinds are sometimes called *raw metrics* and *selection criteria*, respectively (see [34]). 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) using

   `Class.forName(`*class_name*`)`.

   The Java reflection library `java.lang.reflect` was then used to obtain the signature information for each method defined in the class.

2. A class file parser was invoked to obtain further information on each method. This parser was implemented specifically for this research project, and is capable of producing many more code indicators than those investigated here. In particular, the parser includes an extensible data flow analyzer, which produces sophisticated metrics such as method and class chameleonicity [10], a measure of the code's reliance on polymorphic method execution.

Note that the values of metrics computed in this fashion may depend on the specific Java compiler and not only on the source of the program. Our measurements are therefore carried out with the proviso that all the class files were generated by the same compiler. Nevertheless, we do not expect to see much differences between compilers here, since the bulk of the optimization efforts currently revolves around the virtual machine, using techniques such as just-in-time compilation [1].

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

2.1 The Input Sets

The input sets comprised four software libraries or frameworks, two CORBA Object Request Broker (ORB) implementations, and three large applications or sample programs.

JDK *Runtime Library of Sun Java Development Kit version 1.2*

This library (`java.*` packages) comprises the basic runtime services of Java programs, including I/O, Java beans, applets, security, utility classes etc.

Swing *Java Foundation Classes (sometimes called JFC or "Swing") version 1.1*

This framework, which ships with JDK 1.2, comprises all `javax.swing.*` packages and provides high level GUI construction 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.

IBM-XML *IBM XML for Java 1.0*

This is a collection of service classes for parsing and creating XML documents in Java.

CORBA *OMG basic CORBA classes for Java*

A group-effort implementation of the Object Management Group's classes to map the CORBA API to Java.

Orbacus *ORBacus version 4.0.1 for Java*

This is an implementation of a CORBA 2.3 ORB for Java by Object Oriented Concepts, Inc. (OOC) [28]. Included are only the classes comprising the ORB itself.

Orbacus-Test *Test and Demo classes from ORBacus version 4.0.1 for Java*

These are the demonstration and test programs shipping with OOC's ORBacus version 4.0.1 for Java.

Orbix *Orbix 2000 for Java*

A different implementation of a CORBA 2.3 ORB, this one by IONA Technologies [29].

SF-samples *Sample Programs from the IBM San Francisco Framework Release 1.3.1*

The complete suite of demonstration and sample programs shipping with IBM's San Francisco Project ([38], [39]), release 1.3.3. The San Francisco Project itself includes thousands of classes, providing a wide framework for business applications. These methods were not included in this study for reasons explained below.

The class files in each of these nine 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 other classes. About 15% (1,310 classes) of the input classes were inner classes. The number of methods defined in these classes was 6,465, which is about 9.7% of all methods. In addition, the input included 320 anonymous classes (3.8% of all classes) with a total of 1,280 methods, which are less than 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.

Even though Java interfaces are not compiled to executable code, they are an important component of the Java CPP. In our experiments, we used the fact that interfaces receive a `.class` file representation to study these as well. The basic assumption is that methods whose signature is defined in an interface are abstract. Although abstract methods have no body, there are several meaningful metrics which apply to these, including access level and signature.

On the other hand, we have excluded from this study machine-generated classes, such as the classes created for Remote Method Invocation (RMI) support (`Stub` and `Skel` classes), and the methods found in them.

Table 2.1 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 66,391. Note that if a method is overridden in a subclass, both the original and the overriding implementations were analyzed. To the best of our knowledge, this is one of the largest software ensembles to be studied in the literature, if not the largest (as

Project	# Packages		# Classes		# Methods	
JDK	38	(12%)	1,081	(13%)	11,388	(17%)
Swing	17	(5%)	1,231	(14%)	11,830	(17%)
HotJava	25	(8%)	609	(7%)	5,705	(8%)
IBM-XML	6	(2%)	111	(1%)	1,291	(2%)
CORBA	50	(15%)	1,459	(17%)	9,253	(14%)
Orbacus	49	(15%)	1,046	(12%)	7,486	(11%)
Orbacus-Test	16	(5%)	584	(7%)	4,473	(7%)
Orbix	107	(33%)	1,329	(16%)	8,622	(13%)
SF samples	15	(5%)	1,050	(13%)	6,343	(12%)
Total	323		8,500		66,391	

Table 2.1: Software packages used in the experiments.

suggested by [7]). Of these, a total of 777 methods, or fewer than 1.2%, were found to be *native*, and hence excluded from our study. (*Native* methods are methods implemented using machine-specific instructions, which are not compiled into the JVM's bytecode format.)

Note that the relative weights of the packages in the input set are not equal. JDK, Swing and Orbix together comprise nearly half 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 over 87,000 Java methods from the IBM San Francisco project was not used. An initial analysis indicated that this data exhibits similar characteristics to those of the other, more easily available data. However, considering the huge amount of classes in San Francisco (more than the number of classes in all other projects combined), any special attribute or unique statistical behavior found there would have created an extremely strong bias. We did, however, include the classes from San Francisco's *sample programs*.

Input selection is a difficult process, which requires balancing factors such as code availability, personal evaluation of quality, etc. We believe that the nine projects included in this study represent a reflective sample of the common Java programming practice.

This CPP reflects *generic* programming in Java. We believe that programs created in some special domains (e.g., applets in Java, or real-time programs with respect to C programming) would exhibit a

different common programming practice than the generic CPP of their respective languages.

2.2 The Categorical Metrics

The five categorical metrics included in this study were:

1. *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*.

2. *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*, *protected*, or *package*, the latter being the default access level of methods in Java.

3. *Abstraction Level* (ABS) Methods were also classified 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*. We do not consider methods in a class denoted final as being final methods, even though such methods cannot be overridden. Note that the ABS metric is not entirely orthogonal to SORT, since constructors cannot be abstract or final.

4. *Static/Virtual* (STAT) A basic division of methods into two groups distinguishes static methods from those that are non-static ("virtual" methods in C++ terminology).

5. *Refinement* (REF) Yet another binary division is whether a method is a refinement of a directly or indirectly inherited and overridden method.

There are several other categorical metrics which we have collected, but have not included in this analysis. One of these metrics is the method's return type (**`void`**, a primitive type, or an object type). Another possible categorical metric uses 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).

2.3 The Numerical Metrics

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

Intrinsic Complexity (IC) 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 [20] 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 (Lines Of Code) 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, comments, and other personal preferences and text-formatting issues.

3. *Mathematical Opcodes* (MathOp) The number of mathematical opcodes (including integer and float arithmetic) appearing in the compiled method.

4. *Instantiation Opcodes* (NewOp) The number of instantiation sites in the compiled code, i.e., the number of times `new` and similar opcodes are found.

5. *Local Variables Access* (LV) The number of sites in the code that access local variables. This includes sites that access the method parameters, including the implicit **`this`** parameter in non-static methods.

   Note that any measure of sites in the code (here and in other metrics) is hardly an indicator of the actual number of times the operation (such as accessing local variables, in this case) takes place during the method's execution: a single site can be visited numerous times (e.g., inside a **`for`** loop), or not at all.

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.

6. *Internal Messages* (InM) The number of call sites in the code that send messages to **`this`**, or static messages to the class in which the method itself is defined.

7. *Parameters* (PARAM) The number of parameters the function accepts, excluding the invisible parameter **`this`** for non-static methods.

8. *Self-Fields Access* (SFA) The number of sites in the method's code that access fields in **`this`** (for reading or writing purposes). For static methods, this value refers only to the number of sites that access static fields defined in the same class.

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

9. *External Messages* (ExM) The number of call sites in the code used for sending messages to objects other than **`this`**, or static messages to other classes.

   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`**).

10. *Fields Access* (FA) The number of sites in the code that access fields in objects other than **`this`** (for reading or writing purposes).

There is a large number of additional metrics that were collected, but are not analyzed in this work. In particular, we can distinguish, for some of the metrics, between the *vocabulary*, which is the number of unique instances used, and the *text*, which is the total number of instances appearing (i.e., including repeated appearances). For example, the vocabulary of the local variables access (LVA) metric is the number of unique local variables that are potentially accessed by the code. The text of the LVA metric is the number of sites in the code that access a local variable. In the latter measurement, two different sites that access the same variable are both counted. As another example, consider the internal messages (InM) metric. The vocabulary of InM is the number of different, unique messages that the method can potentially send (depending on its flow of execution) to its own object (**`this`**). The text of the InM metric is the number of sites in the code that send a message to **`this`**. If the same message is sent in two different places, it is counted twice in this case.

Of the ten metrics listed above, this dichotomy applies to seven: MathOp, NewOp, LVA, InM, SFA, ExM and FA. In all cases, both variants exhibited a very similar behavior, though naturally, the text metric is always equal to or larger than its respective vocabulary metric. Here we have used the *text* variant of each metric. The exact relationship between the two is a subject for further research.

Additional metrics that were measured but are not used in this work include the number of exception handlers in the method, the number of static messages sent by a method, the number of local variables accessed only for reading, the number of local variables that are both read and written, etc. Some of these metrics are components of metrics we did use. Others were tested and we believe them to be non-monotonic. Finally, several metrics were not used simply because we consider them to be of little or no interest.

Chapter 3

Analysis of Categorical Metrics

In this chapter we study the distribution of the categorical metrics in our sample. Table 3.1 is a cross table of the abstraction level (ABS) metric by the access level metric (ACL). Here and in the following chapters, the most important findings are summarized under appropriate headlines.

	Private	Protected	Package	Public
Final	13.0%	6.9%	11.4%	68.7%
Concrete	6.7%	5.4%	6.6%	81.3%
Abstract	0.0%	1.7%	1.5%	96.9%
Total	6.1%	5.0%	6.2%	82.7%

Table 3.1: Cross table of ABS (abstraction level) by ACL (access level).

CPP Finding 1: *Dominance of public access.*

From the last row of the table we see that the vast majority, almost 83%, of the 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 an application program (e.g., 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.

The finding that only about 6% 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 4% 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.

Further insight into the usage of access level can be gained from Table 3.2, which cross tabulates method sort (SORT) against ACL. We see that a rather large fraction (over 17%) of constructors have non-public 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 [9]. Further testing shows that over 6% of the concrete, public classes have no public constructors, a finding that strongly reinforces this speculation.

	Private	Protected	Package	Public
Finalizer	0.0%	84.2%	0.0%	15.8%
Plain	6.8%	5.3%	5.0%	82.9%
Constructor	2.4%	3.2%	11.9%	82.5%
Total	6.1%	5.0%	6.2%	82.7%

Table 3.2: Cross table of SORT (method sort) by ACL (access level).

The dual of Table 3.1 is given in Table 3.3 which cross tabulates ACL by ABS.

	Final	Concrete	Abstract
Private	4.1%	95.9%	0.0%
Protected	2.6%	93.8%	3.6%
Package	3.5%	93.9%	2.5%
Public	1.6%	85.9%	12.5%
Total	1.9%	87.4%	10.7%

Table 3.3: Cross table of ACL (access level) by ABS (abstraction level).

Examining the last column in this table we see that an overwhelming majority (over 87%) of all methods are “concrete”, and that Java programs make minimal use (less than 2%) 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%).

CPP Finding 2: *Finalizers are rarely used.*

Table 3.4 is the dual of Table 3.2, being a cross table of ACL by SORT. Here we can 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.

	Finalizer	Plain	Constructor
Private	0.0%	93.4%	6.6%
Protected	2.0%	87.7%	10.3%
Package	0.0%	68.2%	31.8%
Public	0.0%	83.6%	16.4%
Total	0.1%	83.4%	16.4%

Table 3.4: Cross table of ACL (access level) by SORT (method sort).

More interesting information about the different access levels can be found in Table 3.5, cross tabulating ACL by STAT.

	Non-static	Static
Private	89.8%	10.2%
Protected	98.1%	1.9%
Package	75.9%	24.1%
Public	81.5%	18.5%
Total	82.5%	17.5%

Table 3.5: Cross table of ACL (access level) by STAT (static indicator).

CPP Finding 3: *Common use of package-wide global elements.*

Table 3.5 shows us that approximately a quarter (24.1%) of all package-level methods are static. This high figure indicates that many packages define "global" elements (within the package scope), shared by several (or all) classes in the package. This can be viewed as further evidence for the need, in Java, for a stronger linguistic unit than the package (see [3]).

Static methods are much less common in other access levels. They are least common among protected methods (less than 2%), and somewhat more common among private methods (about one in ten). About 17.5% of all methods are static.

Table 3.6 cross tabulates SORT by ABS.

	Final	Concrete	Abstract
Finalizer	0.0%	100.0%	0.0%
Plain	2.3%	84.9%	12.8%
Constructor	0.0%	100.0%	0.0%
Total	1.9%	87.4%	10.7%

Table 3.6: Cross table of SORT (method sort) by ABS (abstraction level).

There is not much surprising information in this table. Note that 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.

Let us now turn to the characteristics of refining methods. Table 3.7 cross tabulates REF by ABS.

	Final	Concrete	Abstract
Non-refinement	2.3%	84.7%	13.0%
Refinement	0.1%	99.9%	0.0%
Total	1.9%	87.4%	10.7%

Table 3.7: Cross table of REF (refinement indicator) by ABS (abstraction level).

Naturally, no abstract method is a refiner. An abstract method can override an inherited one, but not every overriding is a refinement. In particular, any overriding by an abstract method is not defined as a refinement, since the new method does not invoke the overridden one.

The interesting finding in this table is the low portion of final refining methods. While final methods comprise 1.9% of the whole input set, only 0.1% of the refining methods are final. Where there is one refinement, there is probably room for additional future refinements, which could explain this datum.

CPP Finding 4: *Replacement is preferred over Refinement.*

From Table 3.8, which cross tabulates SORT by REF, we find that refinement is not very common among plain methods (only 2.3% of plain methods are refiners). This implies that the Java Common Programming Practice prefers *replacement* (commonly called *American semantics*) over refinement (*Scandinavian semantics*) [4, page 196].

	Non-refinement	Refinement
Finalizer	76.3%	23.7%
Plain	97.7%	2.3%
Constructor	5.4%	94.6%
Total	82.5%	17.5%

Table 3.8: Cross table of SORT (method sort) by REF (refinement indicator).

Unsurprisingly, most constructors are refiners: they use **super**(...) (explicitly or implicitly) to invoke the inherited constructor. Yet about one in twenty constructors (5.4%) does not call **super**(...). With the single exception of the constructor of `java.lang.Object`, which has no inherited constructor to call, all other constructors which are not refiners simply call an overloaded constructor from the same class, using **this**(...). In most cases, constructors calling **this**(...) are used for providing default parameters. This means that the lack of support for default parameter values in Java causes an estimated increase of circa 5% in the number of constructors alone.

CPP Finding 5: *Finalizers fail to use refinement.*

From Table 3.8 we find that about one in four finalizers is a refiner, i.e., relies on an inherited finalizer to do some of the cleanup job. This value is surprisingly low. We have expected all finalizers to call the finalizer they override (all finalizers override an inherited finalizer, directly or indirectly, since the method `finalize()` is defined in `java.lang.Object`). In C++, for example, the destructors for base objects are automatically invoked when the derived class's destructor is used. This is done in order to ensure, among other things, proper cleanup for private members in base classes, etc. To reach the same result in Java, finalizers must *explicitly* call the inherited `finalize()`, yet we see that most finalizers do not bother to do so. Apparently, many programmers believe that since `java.lang.Object`'s `finalize()` method does nothing, there is no point in calling the inherited finalizer in most cases. However, this behavior of the default finalizer could theoretically change in future versions of the Java platform, in which case most finalizers would become broken code.

We believe that the proper solution is for the Java language definition to include an implicit call to the inherited `finalize()` in any finalizer that does not make such a call explicitly. This would be similar to the implicit call to **super**() in constructors.

Chapter 4

Distribution of Numerical Metrics

In this chapter we study the distribution of our 10 numerical metrics. Table 4.1 gives some of the essential statistics of each of these metrics. These statistics were computed only for the 57,286 non-abstract, non-native methods.

Metric	Min	Max	# values	Mean	Median	Common	SD	SD/Mean
1. McCabe (MCC)	1	198	85	2.09	1	1	3.70	177%
2. Bytecodes (BC)	1	13911	889	48.86	15	5	149.27	306%
3. Math copcodes (MathOp)	0	204	69	0.46	0	0	2.87	621%
4. New opcodes (NewOp)	0	378	92	0.65	0	0	4.00	620%
5. Local variables access (LVA)	0	2707	270	8.58	3	1	23.48	274%
6. Internal Messages (InM)	0	160	63	1.03	1	0	2.54	248%
7. Parameters (PARAM)	0	24	24	0.98	1	0	1.29	132%
8. Self fields access (SFA)	0	520	96	1.52	0	0	4.92	324%
9. External Messages (ExM)	0	1606	168	2.91	1	0	11.92	410%
10. Fields access (FA)	0	361	87	0.87	0	0	4.14	475%

Table 4.1: Essential statistics of numerical metrics.

4.1 The Essential Statistics of Numerical Metrics

A metric which was widely studied in the literature is the number of parameters to methods. Meyer [22] argues that in a good object oriented design, the average value of this metric tends to be small.

CPP Finding 6: *Java methods accept a relatively large number of parameters.*

In the Eiffel Base library [21], 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 [16] report on only slightly higher numbers, ranging between 0.3 and 0.9, in a variety of Smalltalk projects. Table 4.1 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 24 (!) arguments. One can also surmise from the "# values" column that this is not a singular phenomena. It should be noted that all eight methods with 20 or more parameters are from the San Francisco Project's sample programs. There are 11 such methods, all of which are used for initialization. For example, the method `com.ibm.sf.samples.TaskFactory.createTask` has two variants, one of which accepts 24 parameters, the other "only" 23. Outside of San Francisco's samples, only one method accepts as may as 19 parameters (`layoutMenuItem` in Sun's Swing class `BasicMenuItemUI`).

A total of 197 methods in our sample (0.34%) had 8 or more arguments (compared to the maximum of 7 parameters in the Eiffel library methods). A possible explanation is that our sample size of circa seventy thousand methods was larger by an order of magnitude than the 5,489 methods found in both the Eiffel Base and Vision libraries combined. We will revisit this point below after deriving a formula approximating the distribution of metric values.

CPP Finding 7: *Java methods send few messages.*

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 InM and ExM metrics (internal and external messages, respectively). In total, we have that non-abstract Java methods make an average of 3.94 method calls, which is much less than the corresponding Smalltalk values. However, one must keep in mind that in Smalltalk (in contrast to Java), every comparison, assignment, and mathematical operation involves message sending.

CPP Finding 8: *Metric distribution is skewed.*

Table 4.1 also reveals a huge standard deviation as a phenomenon which sweeps all metrics. In fact, the standard deviation 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 half of the 10 distributions, and equal to the *common value* in 8 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 very close to the minimal value.

4.2 The Zipf-Like Distribution of Metrics

These findings lead us to the non-surprising belief that methods obey a Zipf-like distribution [37], 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 15, 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`, etc.)

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 Figures 4.1 and 4.2. 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 ladder [35], [27] for analysis of distribution (with the exception of the exponential decay hypothesis, $y = K10^{kx}$, fifth on the ladder).

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 $1/57286$. The next such line corresponds to a frequency of $2/57286$, etc.

CPP Finding 9: *Metric distributions have a linear regression model.*

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

$$\log f(x) = C - K \log(x), \tag{4.1}$$

where C and K are some constants, or alternatively

$$f(x) = cx^{-K}, \tag{4.2}$$

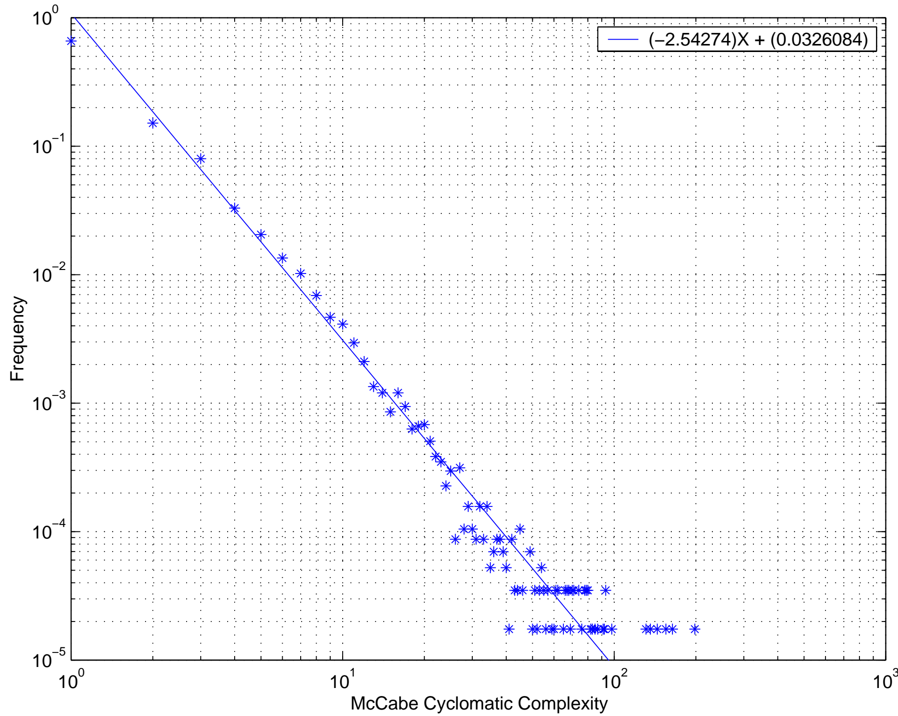

Figure 4.1: Distribution of the MCC (McCabe Cyclomatic Complexity) metric.

where c is some other constant ($c = 10^C$). Equation (4.2) is reminiscent of Mandelbrot law [17] for distribution of words in natural language text.

We call K the metric *frequency coefficient*.

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, Figures 4.3 and 4.4 give the distribution in the number of parameters and the number of fields access operations, respectively.

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* 10 numerical metrics. This pattern also appears in the vocabulary variance of the methods selected, and in many other metrics not included here. Table 4.2 summarizes the results of linear regression analysis in the log-log space of all numerical metrics.

We see that the frequency coefficients are usually in the range of approximately 1.9 to 3.0. The two exceptions are the BC metric, for which $K = 1.40$, and the number of parameters, for which $K = 3.42$.

CPP Finding 10: *The regression coefficient can be used for constructing composite metrics.*

The value of K can be used as a weighting factor in combining several metrics into a single

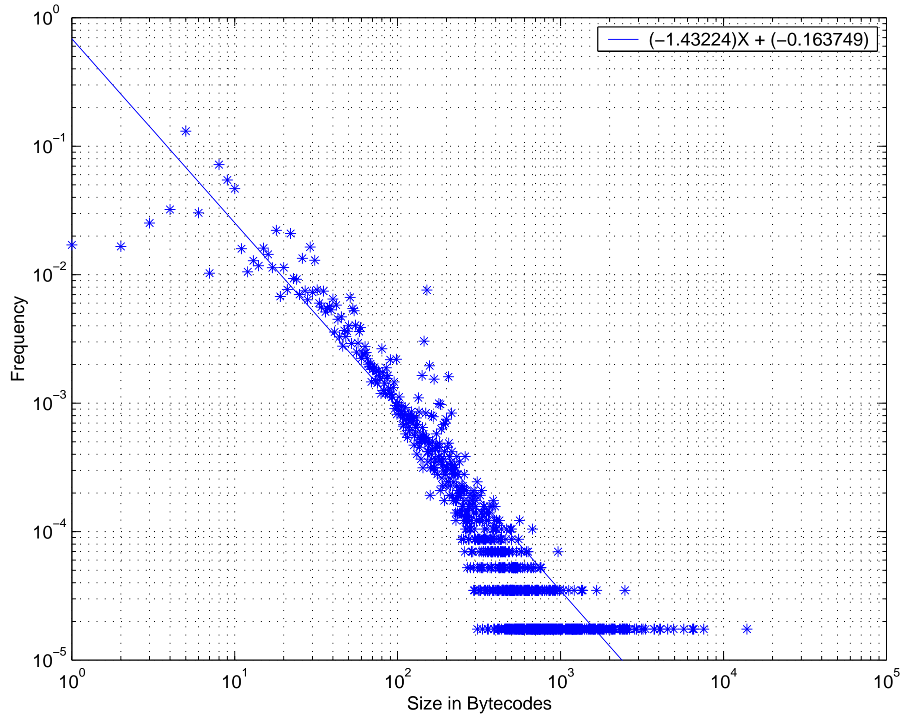

Figure 4.2: Distribution of the BC (size in bytecodes) metric.

Metric	K	C	R^2	p
1. McCabe (MCC)	2.54	0.03	0.98	0
2. Bytecodes (BC)	1.43	-0.16	0.88	0
3. Math copcodes (MathOp)	2.39	-0.41	0.97	0
4. `New` opcodes (NewOp)	2.55	-0.28	0.96	0
5. Local variables access (LVA)	1.92	0.17	0.95	0
6. Internal Messages (InM)	2.93	0.14	0.96	0
7. Parameters (PARAM)	3.42	0.39	0.93	6.9e-09
8. Self fields access (SFA)	2.39	0.04	0.99	0
9. External Messages (ExM)	2.14	-0.00	0.98	0
10. Fields access (FA)	2.34	-0.22	0.98	0

Table 4.2: Linear regression coefficients and statistics of numerical metrics.

composite metric [14, Chap 5.4]. Statistically, the frequency coefficient is crucial in computing the essential statistics of a distribution of the sort of (4.2). It is not difficult to see that if the distribution of x obeys (4.2), then the expected value of x depends only on the frequency coefficient (and not on

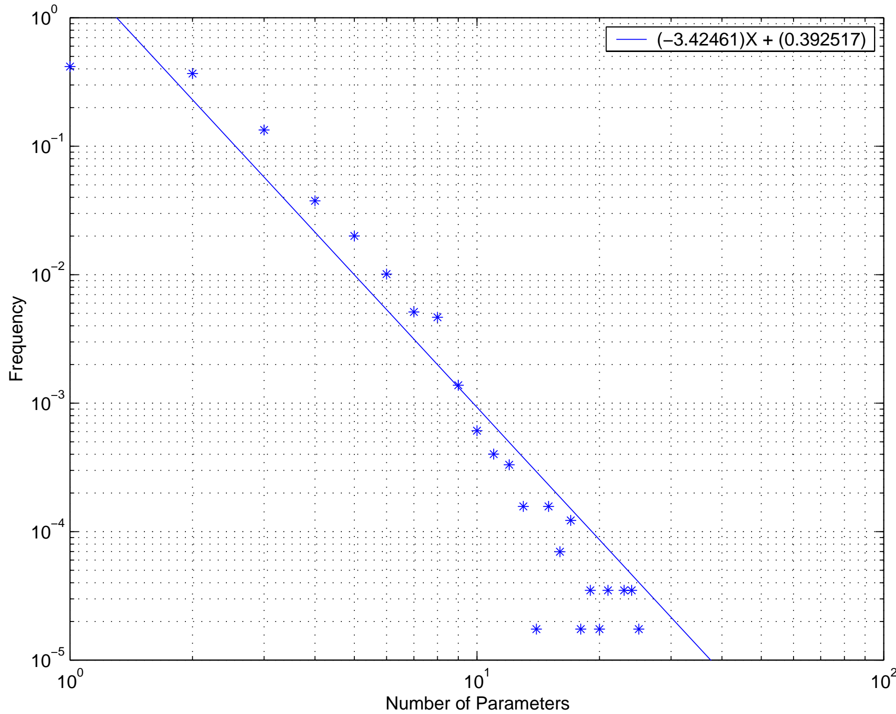

Figure 4.3: Distribution of the PARAM (number of parameters) metric.

the size of the sample),

$$E(x) = \frac{\zeta(K-1)}{\zeta(K)} - 1 \tag{4.3}$$

for all $K > 2$, where

$$\zeta(K) = \sum_{x=1}^{\infty} x^{-K}. \tag{4.4}$$

(The function $\zeta(\cdot)$ is nothing other than Riemann's Zeta function, but this is inessential to the derivation leading to (4.3).) 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, and (to a lesser extend, since this a border case and the value might change with a different input set) in the LVA metric.

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

$$\sigma(x) = \sqrt{\frac{\zeta(K-2)}{\zeta(K)} - \left(\frac{\zeta(K-1)}{\zeta(K)}\right)^2}. \tag{4.5}$$

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

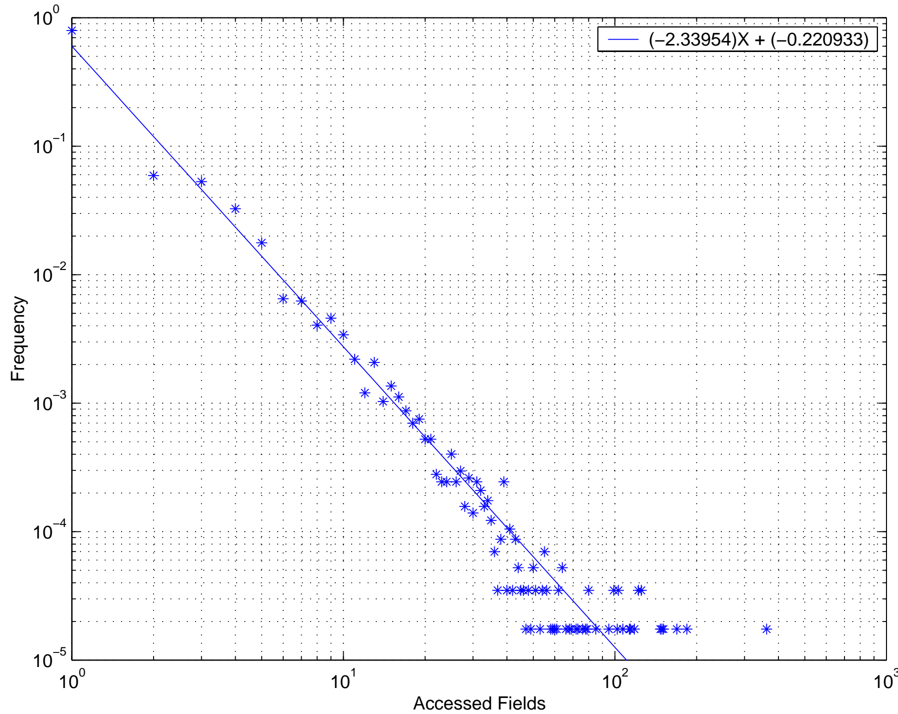

Figure 4.4: Distribution of the FA (fields access) metric.

Unfortunately, (4.3) and (4.5) are not good predictors of the respective values of a distribution *approximated* by (4.2), and hence they are of mere theoretical interest. We learn however from (4.3) and (4.5) that the value of the intercept C is less interesting since it does not take part in the important statistics of the distribution.

Table 4.2 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 over 85% of this variability can be explained by the regression model. Typically, R^2 is at the level of 95% and sometimes even as high as 99%. 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 in all cases but one they were underflowed to zero by the underlying mathematical software [18] . 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

$$\sum_{m=0}^{\infty} f(m) = 1 \tag{4.6}$$

where $f(m)$ denotes the predicted frequency of a metric assuming a value m. Another complicating matter is the skewing of the regression line caused by metrics values which occur a very small number of times, as apparent by the horizontal lines at the bottom of Figures 4.1, 4.2, 4.3, and 4.4. To prevent this skewing, we have ignored the five least-common values of each metric when calculating the regression constants. In our experiments, this choice had led to the best-fitting regression lines, as measured by the R^2 and p values, and this was further confirmed by simple visual comparisons. A possibly better solution, left for future research, is to group the metric values in intervals, especially for metric values which occur a very small number of times. Such a procedure is likely to improve the regression fitting by eliminating the clustering along the horizontal lines at the bottom of the distribution diagrams.

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 4.2 into (4.2) we find that this frequency is 0.57% (to reach this value, one has to repeatedly apply (4.2) to all integer values of x greater than 8. The sum converges to 0.57%). In other words, in the abovementioned collection of Eiffel methods we would then expect over 30 such methods (a Poisson distribution model can be easily applied here, where n is the number of methods and p is the probability of a method having 8 or more parameters) The fact that there are none is significant, and indicates a meaningful difference in style and in the Common Programming Practice between the two languages.

4.3 The Transformed Metric

CPP Finding 11: *Metric logarithm values are more meaningful than the original values.*

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,

$$m' = \begin{cases} \log(m) & \text{if} \min(m) > 0 \\ \log(m+1) & \text{if} \min(m) = 0 \end{cases}, \tag{4.7}$$

where m is the original metric value, is more meaningful then the non-transformed value. Consider for example the method size in bytecodes. It is not so important to know that the number of bytecodes is

exactly 1,973. 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. In checking Table 4.3, 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 four. 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.

Metric Value	Predicted Frequency
2	22.89%
3	5.72%
4	2.13%
5	1.00%
6	0.53%
7	0.32%
8	0.20%
9	0.13%
10	0.09%

Table 4.3: Predicted frequency of methods by the number of parameters.

Table 4.4 is a revision of Table 4.1 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. The inverse of the standard deviation was defined as half the difference between of inverse transforms of the mean plus and minus the standard deviation.

Comparing Table 4.1 and Table 4.4 we see similar phenomena for all metrics: as expected, the mean value decreases by using the logarithmic transformation. Also, although the standard deviation 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	SD/Mean
1. McCabe (MCC)	1.49	1.05	70%
2. Bytecodes (BC)	17.90	30.69	171%
3. Math copcodes (MathOp)	0.16	0.56	354%
4. `New` opcodes (NewOp)	0.29	0.67	237%
5. Local variables access (LVA)	3.51	6.05	172%
6. Internal Messages (InM)	0.62	0.98	157%
7. Parameters (PARAM)	0.70	0.93	132%
8. Self fields access (SFA)	0.69	1.34	194%
9. External Messages (ExM)	1.21	2.14	176%
10. Fields access (FA)	0.32	0.87	275%

Table 4.4: Essential statistics of the transformed metrics.

Chapter 5

Correlating Numerical Metrics

Applying the logarithmic transformation is especially important in models which assume linearity of numerical metrics. An important case in point is co-variance and correlation coefficients. Table 5.1 presents the coefficient of correlation between each of the pairs of numerical metrics. In each table cell, the top value shows the pairwise coefficient of correlation between the original respective metrics, while the bottom value shows this coefficient after applying the logarithmic transformation. If the difference between these two values is greater than 0.04, then the larger value is printed in boldface.

There are several things to notice in the table. First, all values are positive (though a few values are too close to zero to be of any significance). This observation strengthens our belief that all numerical metrics reflect different perspectives of method complexity.

We witness strong positive correlation (0.67) between BC and MCC, which grows to a more significant value of 0.75 for the transformed metrics. Longer methods tend to have more branches and in general, follow a more complicated flow of control.

CPP Finding 12: *Large methods tend to operate on other objects.*

There are significant correlations between the number of messages sent to other objects and the size in bytecodes (0.85 or 0.80). This indicates that larger methods tend to use methods from other classes. On the other hand, the correlations between the size and the number of internal messages (InM) is not as strong (0.53 or 0.49). Thus we can conclude that more than large methods tend to manipulate the object on which they were invoked, they manipulate other objects. These "other objects" can include objects passed as parameters, or objects that are referenced by the current object's

Metric	2	3	4	5	6	7	8	9	10
1. McCabe	0.67 **0.75**	0.49 **0.56**	0.17 **0.34**	0.74 0.71	**0.42** 0.31	0.12 **0.19**	**0.51** 0.41	0.48 **0.59**	0.51 0.47
2. Bytecodes		0.36 **0.47**	**0.69** 0.53	0.90 0.92	**0.53** 0.49	0.11 **0.32**	**0.61** 0.49	**0.85** 0.80	0.55 0.55
3. Math copcodes			0.06 **0.19**	0.48 0.52	**0.19** 0.12	0.16 **0.21**	0.29 **0.33**	0.15 **0.31**	**0.42** 0.35
4. `New` opcodes				**0.52** 0.41	**0.42** 0.22	0.00 **0.08**	**0.34** 0.26	0.63 0.65	0.26 **0.32**
5. Local variables access					0.52 0.50	0.19 **0.41**	**0.66** 0.52	**0.82** 0.69	**0.56** 0.47
6. Internal Messages						0.06 0.09	**0.28** 0.09	**0.34** 0.27	0.24 0.26
7. Parameters							0.06 0.08	0.06 **0.23**	0.06 0.06
8. Self fields access								**0.55** 0.35	**0.40** 0.22
9. External Messages									0.41 0.44
10. Fields access									

Table 5.1: Correlations between metrics (top) and between transformed metrics (bottom).

fields, as well as objects created by the method itself during its execution.

CPP Finding 13: *Local variables (and parameters) are used when accessing other objects.*

Another high correlation (0.82 or 0.69) is found between the number of external messages (ExM), and the number of local variable access operations (LVA). This could indicate that local variables are often used as parameters to methods when sending external messages. The relatively low correlation between LVA and InM (0.52 or 0.50) might indicate that local variables (which naturally include the calling method's own parameters) are less often used as parameters to internal messages.

The LVA metric is also strongly correlated to the BC and MCC metrics (0.90 and 0.74, respectively for the original metrics, and 0.92 and 0.71 for the transformed metrics).

Other than those mentioned above there are no significant high correlation coefficients in Table 5.1.

Chapter 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.

6.1 On Reading Method Profile Graphs

Figures 6.1, 6.2 and 6.3 give three different profiles of method groups. The ticks on the X-axis correspond to the metrics, with the standard numbering as used above (see e.g., Table 4.1). Recall that metrics 1–5 belong in the intrinsic-complexity group, metrics 6–8 form the self-interaction group, while the interaction-with-others group includes metrics 9 and 10.

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 standard deviation 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. Since the various metrics are presented in an arbitrary order within each group, the line used for connecting the points is primarily a visual aid, and no additional significance should be attributed to it.

6.2 Profiling by Access Level

Figure 6.1 gives the profile of public, protected, private and package level access methods.

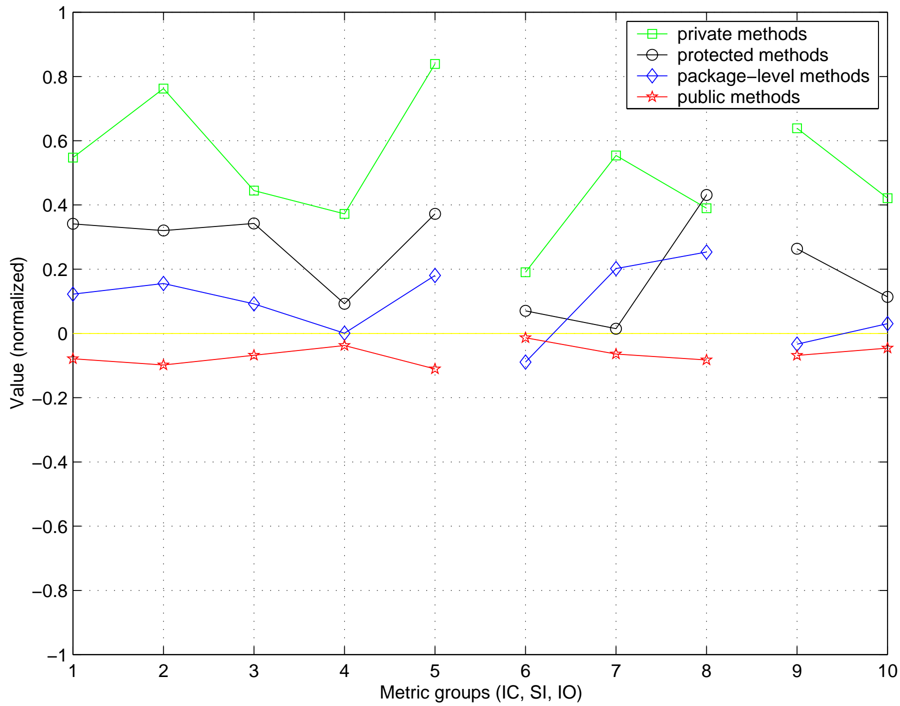

Figure 6.1: Profiles of the different categories of ACL (Access Level).

CPP Finding 14: *Complexity increases with decreased accessibility.*

Concentrating first on the intrinsic complexity group of metrics, we see that private methods achieve higher values, by 0.4 to 0.8 standard deviation 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 0.4 standard deviation units. Protected methods are only slightly more complex than the average. In contrast, public methods are marginally *simpler* than the average. The overall result is a clearly visible hierarchy of intrinsic complexity. This exposes an order between access levels—from the most complex private methods down to the least-complex public ones.

The order still exist, though in a less clear manner, in other metric groups. For example, moving on to the self-interaction group of metrics, we see that the differences between the four ACL categories here are generally smaller. Private methods still tend to have higher metric values in this group. One can also notice that public methods again rank slightly below the average in this group. No such clear statement can be made about protected and package level access methods. It is interesting to note that

"package" methods have a small number of parameters, and at the same time, a high number of self fields access sites. This could indicate that these methods are used for granting classes in the package privileged access to internal fields (e.g., "`get`" methods).

In the third, interaction-with-others, group of metrics we again see that private methods give higher metric values, and that protected methods rank second. Here, both "package" and public methods rank somewhat below the average. This could indicate that these more-accessible methods generally deal with the class itself, having a lower interaction with other objects.

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.

6.3 Profiling by Method Sort

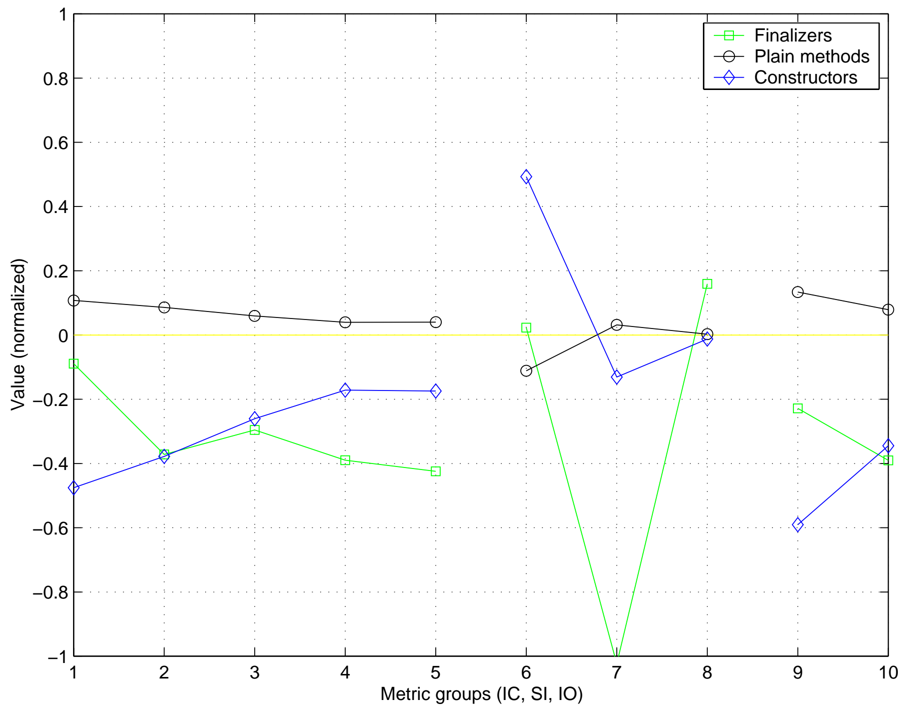

Figure 6.2: Profiles of the different categories of SORT (method sort).

In Figure 6.2, which shows the profiles of the different categories of SORT, we see that finalizers tend to be significantly simpler than other methods in all metric groups, with few exceptions. It is obvious that finalizers score far below the average on the number of parameters metrics, since finalizers by definition accept no parameters at all (other than the implied **`this`**). It is somewhat surprising that

finalizers send *more* internal messages than plain methods. They also access self fields more than either of plain methods and constructors do. This point requires further investigation. However, since there is a relatively small number of finalizers, one should not ascribe too much meaning to these findings.

CPP Finding 15: *Constructors are relatively simple.*

Constructors on the other hand tend to be simpler than plain methods, with few notable exceptions. It is not surprising at all that constructors would tend to have very little interaction with other objects, and access self fields. The fact that by language definition, constructors are obliged to call (using `super(...)` or `this(...)`) an inherited constructor or an alternative constructor of the same class, explains why constructors tend to have a larger than the average number of internal messages. It is somewhat surprising that on average, constructors require *less* parameters than plain methods. This could probably be explained by the fact that in Java, overloading is used to provide default values to constructors. Thus, if for example a constructor with three parameters is defined, it is often accompanied by variants with two, one, or even no parameters at all, which would tend to lower the average. Another reason for this would be that the default constructor, generated by the Java compiler for all classes with no constructor definition, takes no parameters.

Again, since the bulk of methods are plain, their behavior is close to the average.

6.4 Profiling by Abstraction Level

Figure 6.3 shows the profile of concrete vs. final methods (most numerical metrics are inapplicable to abstract methods).

Not much can be observed in this figure. Since final methods are a small minority (less than two per cent; recall Table 3.3), the average metric values for concrete methods are extremely close to the overall average values.

It is apparent that final methods have more self interaction, and less interaction with others, than concrete ones. More research is required to sort out this point.

6.5 Profiling by Refinement

Finally, Figure 6.4 shows the profile of refining vs. non-refining methods. Recall that refining methods are those that invoke a version of the same method that they directly or indirectly override.

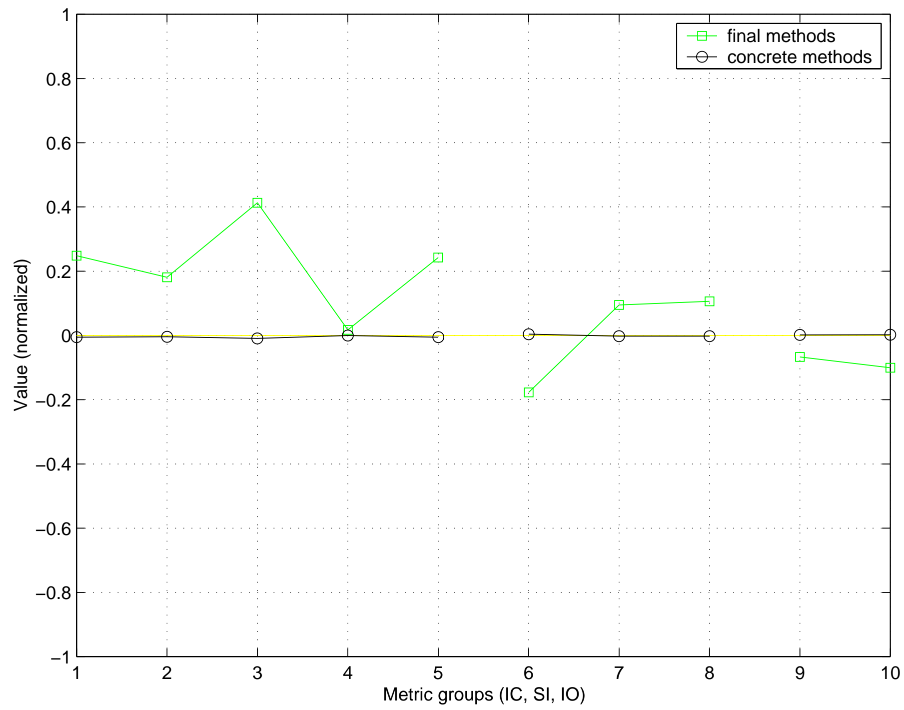

Figure 6.3: Profiles of the different categories of ABS (abstraction level).

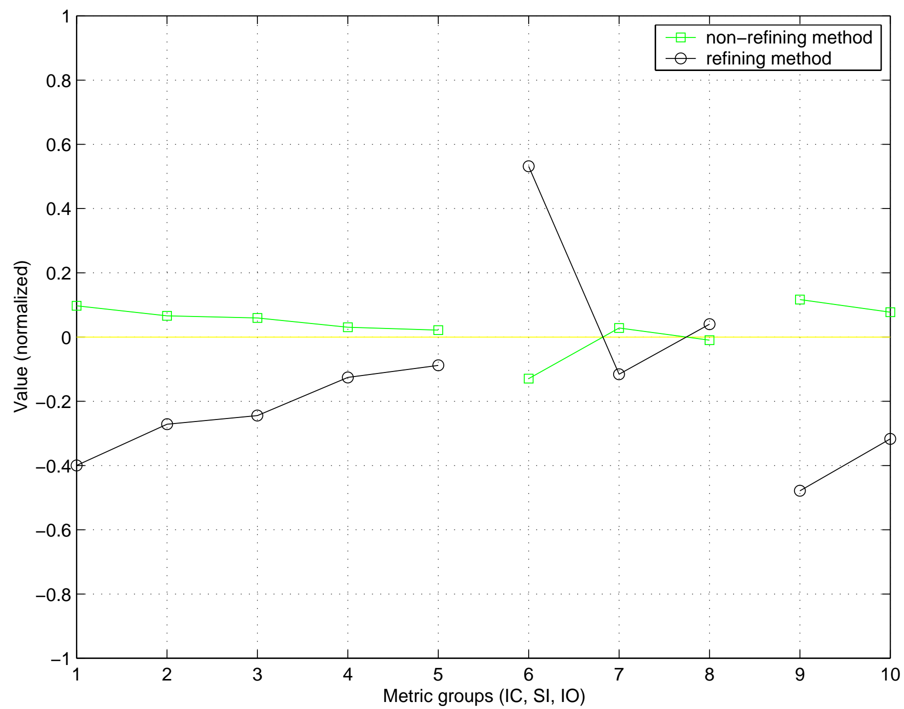

Figure 6.4: Profiles of the different categories of REF (refinement indicator).

CPP Finding 16: *Using refinement decreases method complexity.*

Refining methods are clearly less complex than non-refining ones. This is especially apparent in the interaction-with-others group of metrics.

The two exceptions, both in the group of self-interaction metrics, show that refining methods send far more internal messages than average (by almost 0.6 standard deviation units), and have slightly more access sites for self fields. The larger number of internal messages can be explained by the fact that by their nature, *all* refining methods send at least one such message, whereas such a lower bound does not exist for other methods. The difference in the SFA metric is probably too small to be of any significance.

Chapter 7

Conclusion and Future Research

This first work on self-calibration only served to demonstrate the technique. We have applied self-calibration to monotonic metrics of methods in the Java programming language, while finding what we call the *common programming practice* of this language. Even this modest application of the technique was sufficient for highlighting its benefits.

We have shown that each metric can be characterized by a single positive constant (K, the frequency coefficient). The coefficient can be used for scaling when several metrics are to be combined into one. This scaling is required before weights can be applied to the various metrics.

Based on the Mandelbrot-like distribution common to all metrics measured, we argue that for many purposes the *transformed metric* (defined in Section 4.3) is more meaningful than the original metric value.

We were further able to borrow a visualization and analysis technique from the discipline of psychology. The borrowed technique, which we call *method profiles*, allows researchers to easily identify unique aspects in the common programming practice of a given development environment.

7.1 Benefits

This work's most important contribution is the introduction of the modified metric, and the scaling of metrics using self-calibration. We believe that using the scaled, modified metrics to create composed ones can lead to more meaningful software measurements. This, in turn, could perhaps solve software metrics' lack of acceptance in the software industry ([26], [31]).

Finally, we note that the huge database generated for this research project can probably be useful for additional purposes, not necessarily related to the issue of software metrics.

7.2 Future Research

Much more work must be done in order to apply both self-calibration and method profiles 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 [12], were not included here. It would also be interesting to include metrics generated by software-quality analysis tools, such as lint [6] or Jtest [30].

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 (4.6). Also, we believe that the linear regression would be even better if intervals are used to classify some of the rarer metric values.

Non-monotonic Metrics Self-calibration, as presented here, applies to monotonic metrics only. Additional research is required for extending it to non-monotonic metrics, such as the code-to-comment ratio, the number of exception handlers, etc.

Class Metrics We strongly believe that self-calibration is also applicable to class metrics in addition to method metrics (and perhaps also to package- and project-wide metrics). Many key class metrics are clearly non-monotonic (e.g., the number of fields defined in a class, the interface size, etc.). Thus, this research direction depends on the extension of the technique to enable calibrating non-monotonic metrics.

Specialized Domains and Other Programming Languages The Common Programming Practice, as defined in this research, applies to generic Java programming. As noted in Section 2.1, some domains (such as Applets in the case of Java programs) are likely to have their own typical CPP. The CPP can also be applied to other programming languages, such as C++ and Eiffel. It would be interesting to find and compare the CPP values for the same set of metrics across different programming languages.

In addition, we believe that applying data-mining techniques to the database mentioned in Section 7.1 will yield many interesting results. Early research in this direction indicates the existence of *nano-patterns* within methods, a discovery that could lead to a new way of classifying methods.

Bibliography

[1] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a Just-in-Time Java compiler. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 280–290, 1998.

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

[3] M. Biberstein, J. Gil, and S. Porat. Sealing, encapsulation and mutability. 2001. To appear in ECOOP 2001.

[4] T. Budd. *An Introduction to Object-Oriented Programming*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2nd edition, 1997.

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

[6] I. F. Darwin. *Checking C Programs with lint*. O'Reilly and Associates, 1st edition, 1988.

[7] N. C. Debnath, R. Y. Lee, and H. R. Abachi. An analysis of software engineering metrics in oo environment.

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

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

[10] J. Gil and A. Itai. The complexity of type analysis of Object Oriented programs. In E. Jul, editor, *Proceedings of the 12th European Conference on Object-Oriented Programming*, number 1445 in Lecture Notes in Computer Science, pages 601–634, Brussels, Belgium, July 20–24 1998. ECOOP'98, Springer Verlag.

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

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

[13] 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.

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

[15] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1999.

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

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

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

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

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

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

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

[23] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.

[24] B. Meyer. Free EiffelBase: Eiffel libraries go open source. *Journal of Object-Oriented Programming*, pages 8–17, Nov./Dec. 1998. Eiffel.

[25] 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.

[26] A. Minkiewicz. Software measurement: What's in it for me? *Software Development Magazine*, Mar. 2000.

[27] A. Myrvold. Data analysis for software metrics. *Journal of Systems Software*, pages 271–275, Dec. 1990.

[28] ORBacus for C++ and Java. http://www.ooc.com/ob/, 2001.

[29] Orbix 2000. http://www.iona.com/products/orbix2000_home.htm, 2001.

[30] ParaSoft. Automatic Java software and component testing: Using Jtest to automate unit testing and coding standard enforcement. http://www.parasoft.com/products/jtest/papers/jtestwp_4.pdf, 2001.

[31] S. Rifkin. What makes measuring software so hard? *IEEE Software*, pages 41–45, May/June 2001.

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

[33] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[34] T. Talbi, B. Meyer, and E. Stapf. A metric framework for object-oriented development. In *Proceedings of TOOLS USA Conference*, pages 164–172, 2000.

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

[36] N. Wirth. From programming language design to computer construction. *Commun. ACM*, 28(2), Feb. 1985.

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

[38] F. Zuliani, A. Krause, A. Buch, E. Cattoir, M. Chilanti, S. El-Rafei, S. Abinavam, and W. Mueller. *San Francisco Concepts and Facilities*. International Business Machines Corp., Armonk, New York, 1998.

[39] F. Zuliani, P. Tamminga, D. Feidherbe, and L. Misciagna. *SanFrancisco GUI Framework: A Primer*. International Business Machines Corp., Armonk, New York, 1999.