Understanding the Shape of Java Software

Gareth Baxter, Marcus Frean, James Noble,
Mark Rickerby, Hayden Smith, Matt Visser
School of Mathematics, Statistics, and
Computer Science
Victoria University of Wellington
Wellington, New Zealand
firstname.lastname@mcs.vuw.ac.nz

Hayden Melton, Ewan Tempero
Department of Computer Science
University of Auckland
Auckland, New Zealand
firstname@cs.auckland.ac.nz

Abstract

Large amounts of Java software have been written since the language's escape into unsuspecting software ecology more than ten years ago. Surprisingly little is known about the structure of Java programs in the wild: about the way methods are grouped into classes and then into packages, the way packages relate to each other, or the way inheritance and composition are used to put these programs together. We present the results of the first in-depth study of the structure of Java programs. We have collected a number of Java programs and measured their key structural attributes. We have found evidence that some relationships follow *power-laws*, while others do not. We have also observed variations that seem related to some characteristic of the application itself. This study provides important information for researchers who can investigate how and why the structural relationships we find may have originated, what they portend, and how they can be managed.

Categories and Subject Descriptors D.2.8 [*SOFTWARE ENGINEERING*]: Metrics—Product metrics; D.1.5 [*PROGRAMMING TECHNIQUES*]: Object-oriented Programming

General Terms Design, Measurement

Keywords Power-law distributions, object-oriented design, Java

1. Introduction

Much of software engineering has focused on how software could or should be written, but there is little understanding of what actual software really looks like. We have development methodologies, design principles and heuristics, but even for a well-defined subset of software, such as that written in the Java programming language, we cannot answer simple questions such as "How many methods does the typical class have?" or even "Is there such a thing as a 'typical class'?"

What we would really like to know about software is "Is it good?" that is, does it have quality attributes such as high modifiability, high reusability, high testability, or low maintenance costs. We believe current methodologies lead to good software, but without knowing what good software looks like, we cannot know that the methodologies are actually working. We are left with circular arguments of the form "The methodologies are good because the software is good, and the software is good because the methodologies are good." Understanding the shape of existing software is a crucial first step to understanding what good software looks like.

Just as biologists classify species in terms of shape and structure and ecologists study the links and interactions between them, we have been collecting a body of software and analysing its abstract form. We remove semantics and focus on the network of connections where information flows between components. Just as biologists (and other scientists) seek to understand the characteristics of the population under study, so too would we like to know such basic features as the distributions of the software structures we find.

Of specific interest are recent claims that many important relationships between software artifacts follow a 'power-law' distribution (e.g. [34]). If this were true, it would have important implications on the kinds of empirical studies that are possible. One issue is the fact that a power-law distribution may not have a finite mean and variance. If this is the case, the central limit theorem does not apply, and so the sample mean and variance (which will always be finite, because the sample size is finite) cannot be used as estimators of the population mean and variance. This would mean that basing any conclusions on sample means and variances without fully understanding the distribution would be questionable at best.

In this paper, we extend past similar studies in two ways. First, we examine a much larger sample than previous studies. We have analysed a corpus of Java software consisting of 56 applications of varying sizes, and measured a number of different attributes of these applications. Second, we consider distributions other than those following a power-law. We find evidence that supports claims by others of the existence of power-law relationships, however we also find evidence that some distributions do *not* appear to obey a power-law. Furthermore, whether or not a relationship follows a power-law appears to depend on an identifiable characteristic of the relationship, namely, whether or not the programmer is inherently aware of the size of the relationship at the time the software is being written. We also see variations between applications. We speculate that this may be due to some characteristic in the application's *design*, that is, some property of the design is reflected in the distribution of some measurements.

The rest of the paper is organised as follows. In Section 2, we discuss the motivation for our study. Section 3 describes in detail the salient features of our study, namely the corpus we use and the metrics we gather. In Section 4 we give the analysis of our results, and in Section 5 we give our interpretation of this analysis. Section 6 discusses the most relevant related work, and we give our conclusions in Section 7.

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.

2. Motivation and Background

Software systems are now large, complex, and ubiquitous, however surprisingly little is known about the internal structures of practical software systems. A large amount of research has studied how software 'ought' to be written, how it 'should' be structured. Many rules, methodologies, notations, patterns and standards for designing and programming such large systems [9, 17, 24] have been produced. Psychological models have been constructed of the programming process [6, 33]. Quantitative models of software have been designed to predict the effort required to produce a system, measure the development rates of software over time (process metrics) or measure the volume of software in a system and its quality (product metrics) — see e.g. [7, 15, 27]. But we know very little about the large-scale structures of software that exists in the real world.

With the methodologies, notations, and other advice that has been developed, we should be able to say something about the software that results *if such advice is followed*. However the conditional is key — until recently there was very little work done in determining even if the advice that has been offered is actually been taken. There is some evidence that common advice is not being followed. For example, a number of people have advised against creating cycles of dependencies in software, but recent evidence suggests that not only do programmers regularly introduce cycles, but they are often very large [20].

One consequence of much of the advice offered with respect to object-oriented design is what we call the *Lego Hypothesis*, which says that software can be put together like Lego, out of lots of small interchangeable components [26, 29]. Software constructed according to this theory should show certain kinds of structure: components should be small and should only refer to a small number of closely related components.

In fact, we don't know whether or not this is true, because we lack models describing the kinds of large structures that exist in real programs. There are no quantitative, testable, predictive theories about the internal structures of large scale systems, or how those structures evolve as programs are constructed [8, 13]. While design patterns, rules, metrics and so on, can give guidance regarding developing program structure, they cannot predict the answers to questions about the large-scale structure that will result, such as: in a program of a given size, how many classes or methods will exist? How large will they be? How many instances of a each class will be created? How many other objects will refer to any given object? We need answers to these kinds of questions in order to be able understand how large scale software is actually organised, built, and maintained in practise.

Recently there has been an interest in looking for *power-law* relationships in software. A distribution of the number of occurrences N_k of an event of size k is a power-law if it is proportional to k raised to some power s. A common method used to detect possible power-laws is to rank the event sizes by how often they occur, and then plot N vs. the rank on logarithmic scales. A distribution following a power-law will appear as a line with slope s.

Studies of computer programs have considered both static [30, 31, 34] and dynamic [23, 25, 26] relationships, in different forms of software as diverse as LISP, visual languages, the Linux kernel, and Java applets [3, 22, 23, 25, 26, 28, 31], and the design of Java programs [5, 30, 31]. The conclusions from these studies is that power-laws appear to be quite common.

Our work follows from Wheeldon and Counsell, who examined a number of inter-class relationships in Java source code, namely Inheritance, Interface, Aggregation, Parameter Type, and Return Type in three Java systems: the core Java class libraries, Apache Ant, and Tomcat [34]. We attempted to reproduce the Wheeldon and Counsell study, and found examples of their metrics that, for some applications, did not appear to obey a power-law. One example is shown in Figure 1 (which appears again, with full explanation, as Figure 3). This figure shows a plot organised as described above — it is a log-log plot of frequency of occurrence of different values of a particular metric. The data in this figure seems to have a distinct curve to it. Had we plotted this on a normal scale, we would see something like a power-law curve, except 'truncated' at the high end. This figure casts some doubt as to whether the distribution shown is a power-law.

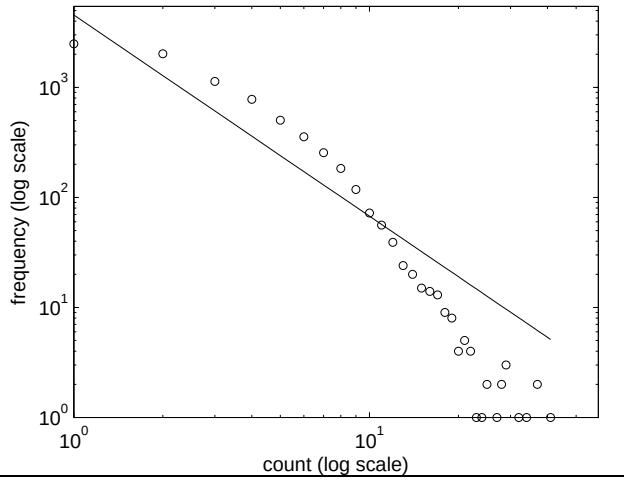

Figure 1. A distribution that does not appear to obey a power-law. Open circles are data, solid line is best fit power law distribution.

Our experience raised two questions. The first is, do the relationships others have studied really obey a power-law? While the evidence provided is compelling to the naked eye, there is little analytical support. In this paper, we will provide such an analysis to support our claims. The second question is, are the studies representative of software in general. This is not a question that can be answered easily due to the scale involved, however, our study involves a much larger corpus than other studies, and so provides better support for our claims.

3. Method

3.1 Gathering the Corpus

The corpus consists of 56 applications whose source code is available from the web. Many of the applications were chosen because they have been used in other studies (e.g., [10, 12, 26]), although comparison to these other studies isn't possible as version numbers were not always provided. Also, we weren't always able to acquire all applications used in those other studies. Further applications were then added to the corpus based on software that we were familiar with (e.g. Azureus, ArgoUML, Eclipse, NetBeans). Finally we identified popular (widely down-loaded) and actively developed open-source Java applications from various web-sites, including: developerWorks[1], SourceForge[2], Freshmeat[3], Java.net[4], Open Source Software In Java[5] and The Apache Software Foundation[6]. Figure 2 gives an indication of the distribution of the size

[1] `http://www-128.ibm.com/developerworks/views/java/downloads.jsp`

[2] `http://sourceforge.net/`

[3] `http://freshmeat.net/`

[4] `http://community.java.net/projects/`

[5] `http://java-source.net/`

[6] `http://apache.org/`

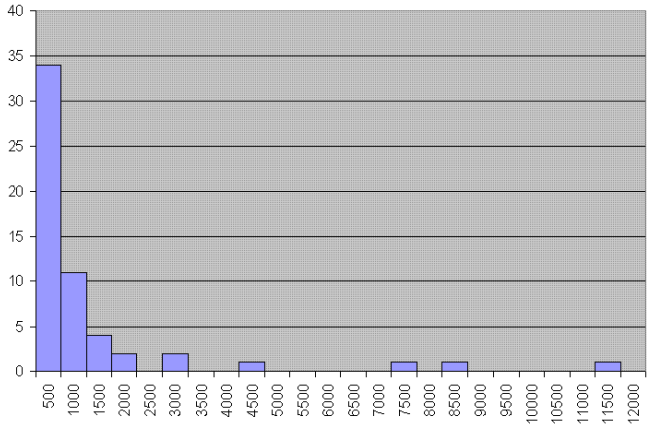

Figure 2. Distribution of application size in Corpus.

of the applications, measured in terms of the number of top-level classes. Appendix B gives more details of contents of the corpus we used.

3.2 Metrics

There are a number of variables that must be taken into account when carrying out this kind of research. In the interests of allowing others to reproduce and extend our results, we discuss our choices in detail.

Any Java program makes some use of the Standard API, and so there is a question of how much the Standard API is counted when doing the analysis. For example, when counting the number of methods per class, should the number of methods in the `java.lang.String` class be counted, or should the number of methods that use `String` as a parameter or return type be counted? This type is so heavily used that measuring its use seems likely to distort the results, and so it would seem reasonable to not consider it. However there are also less frequently used types, such as `java.util.jar.Pack200`, that seem less likely to distort the results and so maybe should be counted. It is not clear where to draw the line.

In this analysis we have chosen to consider only the human editable aspect of an application's construction, that is, the source code that is under the control of the application developers. For this reason, when metrics have been computed, we have considered only those classes declared in the source files of the application. Uses of the Standard API (and indeed any other API used but not constructed for the application) are not considered. In the descriptions below, the phrase "in the source" will reinforce this choice. Note that in the case where the application is the JDK/JRE, it is the Standard API being analysed.

All the metrics have been computed from the byte code representation of 'top-level' classes, that is, classes that are not contained within the body of another class or interface [11, chapter 8]. Relationships relating to inner classes are merged with their containing class. To restrict the analysis to only those classes in the application's source code, names discovered in the byte code were filtered according to package names of packages in the source code. Note that this means our analysis is limited to those applications that use a package structure.

We used two methods to carry out the analysis. One method applied to the byte code directly, using the Byte Code Engineering Library (BCEL)[7]. The other applied `javap`, a Java byte code disassembler that outputs representations of classes in a plain text format. From this, we were able to extract information about the structure of fields, methods, and opcode instructions, which we used to build a meta model of each application as a nested collection of the basic types 'package', 'class', 'method', and 'field'. These collections gave us a simple source for calculating metrics we were interested in. When byte code is generated, some information (particularly type information) is thrown away. This means some of our results will not match a similar analysis done directly on the source code. We discuss this point in more detail when we present the metrics.

Many of the metrics we use come from Wheeldon and Counsell, as indicated in the list below, and we use their naming scheme where possible [34, Figures 8-10]. Due to the difficulty in interpreting their descriptions [34, Figure 1] we give more detailed definitions here, with a more formal treatment in Appendix A. We will use the abbreviations given below. Where the abbreviation does not match the Wheeldon and Counsell names, we indicate the phrase on which they are based.

Our definitions assume that there is only one *top-level* [11] type declaration per source file (`.java` file). That is, we explicitly rule out the following situation, where two classes are declared in the same file (or *compilation unit*).

```
// A.java containing two class declarations
public class A { ... }
class B { ... }
```

The main reason for making this assumption is that it simplifies the definitions. However, compiling the file `A.java` above will yield two files, `A.class` and `B.class`. Since there is no requirement that a class be declared to be `public`, even when it is the only class in a compilation unit, there is no way to tell from looking at `B.class` that it was generated from the same source file as `A.class`.

In the following description, we occasionally need to distinguish between when a name refers to a **class** and when it refers to an **interface**. When no distinction is necessary, we will say the name refers to a **type**.

Number of Methods `nM` (WC)

For a given **type**, the number of all methods of all access types (that is, public, protected, private, package private) declared (that is, not inherited) in the type.

Number of Fields `nF` (WC)

For a given **type**, the number of fields of all access types declared in the type.

Number of Constructors `nC` (WC)

For a given **class**, the number of constructors of all access types declared in the class.

Note that since the measurements are taken from the byte code, this is guaranteed to be at least 1. If no constructor is specified, the Java compiler automatically generates a default public nullary constructor that is included in the byte code.

Subclasses `SP` — Subclass as Provider (WC)

For a given **class**, the number of top-level classes that specify that class in their `extends` clause.

Implemented Interfaces `IC` — Interface as Client (WC)

For a given **class**, the number of top-level interfaces in the source that are specified in its `implements` clause. For a given **interface**, the number of top-level interfaces in the source that are specified in its `extends` clause.

[7] `http://jakarta.apache.org/bcel`

Interface Implementations `IP` — Interface as Provider (WC)

For a given **interface**, the number of top-level classes in the source for which that interface appears in their `implements` clause. Note that when an inner class implements a given interface, it is the top-level class that contains it that is counted.

References to class as a member `AP` — Aggregate as Provider (WC)

For a given **type**, the number of top-level types (including itself) in the source that have a field of that type.

Members of class type `AC` — Aggregate as Client (WC)

For a given **type**, the size of the set of types of fields for that type.

References to class as a parameter `PP` — Parameter as Provider (WC)

For a given **type**, the number of top-level types in the source that declare a method with a parameter of that type.

Parameter-type class references `PC` — Parameter as Client (WC)

For a given **type**, the size of the set of types used as parameters in methods for that type.

References to class as return type `RP` — Return as Provider (WC)

For a given **type**, the number of top-level classes in the source that declare a method with that type as the return type.

Methods returning classes `RC` — Return as Client (WC)

For a given **type**, the size of the set of types used as return types for methods in that type.

Depends on `DO`

For a given **type**, the number of top-level types in the source that it needs in order to compile.

The intent is to count all top-level types from the source whose names appear in the source for the type. There are some rare situations (when only methods from parent classes are called on the object) where the types of local variables are not recorded in the byte code. Our experience is that this happens sufficiently rarely to have no effect on the results.

Depends On inverse `DOinv`

For a given **type**, the number of type implementations in which it appears in their source.

Public Method Count `PubMC`

The number of methods in a **type** with public access type.

Package Size `PkgSize`

The number of **types** contained direction in a package (and not contained in sub-packages).

Method size `MS`

The number of byte code instructions for a method. Note that this is not the number of bytes needed to represent the method.

4. Results

We have applied the 17 metrics described in the previous section to 56 applications from our corpus. This has yielded more data than can be conveniently shown here, so instead we have done some preliminary analysis based on various assumptions as to what the distribution of the data is, and present the results of analysis.

4.1 Analysis

The raw data consists of a number for each 'element' (method, top-level class, package) in each application. The first step was to group all values by application, count the number of occurrences of each value and record that in order of value. The primary goal of our analysis was then to determine whether the resulting distribution obeyed a power-law.

Some of the distributions derived from our analysis of software structure look like straight lines when plotted with logarithmic scales on both axes. This is the hallmark of a power-law distribution, which is interesting because of its 'scale-free' properties, which we will describe below. Any other distribution will not be exactly a straight line in such a plot.

Not all the plots look exactly straight. Some have a sort of curve to them. We can respond by either saying that we do not care, as they are *nearly* straight, at least for part of the range, or we can say that they really are not power-laws at all, and are characterised by some other distribution. Secondly, even if it 'really' is a power-law, because the data is noisy and because there is a finite sample size and a finite range of 'sizes', a power-law curve won't exactly fit the data, especially at large values of the metric. This also means that some alternative distributions might be made to fit the data just as well — we might not be able to discriminate, even for the plots that look pretty straight.

Our approach is to take the data, and do rigorous best-fits to several different distributions, and see first whether it is reasonable to fit a power-law, second whether a power-law is more reasonable than the others, third whether the data can be divided into two or more groups according to which distribution fits 'best'.

4.1.1 Power-Law

In general a power-law distribution has the form [21]:

$$p_{powlaw}(x) \propto x^{-\alpha}\,, \tag{1}$$

where α is a positive constant and we assume x to be non-negative. In our case, x is the value of the metric as defined in the previous section. If $\alpha < 1$ there must be a finite maximum value of x, in order for the distribution to be normalisable. If $\alpha > 1$, normalisability requires that the minimum value of x not be equal to zero. For $\alpha \leq 2$ the mean of the distribution is infinite (assuming there is no upper cutoff in x). When $\alpha > 2$ the mean is proportional to the small-x cutoff. For $\alpha \leq 3$ the variance is also infinite. One consequence of this fact is that the central limit theorem doesn't hold for such distributions, so the mean and variance of a sample (which will always be finite) cannot be used as estimators for the population mean and variance.

A distribution is said to be scale free if [21]:

$$p(bx) = g(b)p(x)\,, \tag{2}$$

where g does not depend on x. This means the relative probability of occurrence of 'events' of two different sizes (bx and x) depends only on the ratio b, and not on the 'scale' x. One of the reasons for the interest in power-laws is that they possess this scale-free property. If we can show that the distributions we see in our analysis of software obey a power-law, we can say that there is no characteristic size (where 'size' might mean in-degree, for example) to the components. A scale-free distribution such as a power-law would contradict the Lego Hypothesis.

While an idealised power-law distribution might be strictly scale-free, for the distributions we encounter in real systems this can only be approximately true. The data in our studies only occurs at discrete, integer values of x. This imposes a small-size cutoff on x — the smallest value of x we measure is 1. There is also a large-size cutoff of x, as the programs in the corpus are of finite size. Nevertheless, we are still interested in power-laws. The scale-free property (2) may still hold over a limited range. We can never say for certain that a distribution *is* a power-law – because we are always dealing with measured data that involve some noise, and

also finite size effects — but we might be able to say that it is approximately a power-law, well characterised by a power-law over a large range, or more likely to be a power-law than something else.

4.1.2 Other Candidates

Given our experience with plots such as that shown in Figure 1, we are interested in distributions that are close to power-laws, but resemble the curves we have seen. Two other distributions which have some credibility as 'natural' distributions are:

Log-normal distribution. Power-laws and log-normals look the same at low values of 'x' (i.e., at the high frequency end), but the tail is 'fatter' for a power-law. For continuous x a log-normal probability density function is defined as:

$$p_{lognorm}(x) = \frac{1}{x\sigma\sqrt{2\pi}} \exp\left\{\frac{-(\ln x - \mu)^2}{2\sigma^2}\right\} , \qquad (3)$$

while for discrete values of x, the normalisation will be more complicated, and the distribution is of absolute probability, not probability density.

Note that our data is not ranked, so it is usually, but not necessarily monotonically decreasing with x: sometimes the smallest value of x does not have the highest frequency. Log-normal distributions can reproduce this pattern, but to fit a power-law we must treat this 'turnover' as a statistical anomaly.

Stretched exponential. This is known to occur in natural distributions [18] (it is the same as the two-parameter Weibull distribution [32] which is used to model electrical component failure probabilities):

$$p_{strexp}(x) = \frac{c}{x_0}\left(\frac{x}{x_0}\right)^{c-1} \exp\left\{-\left(\frac{x}{x_0}\right)^{c}\right\} . \qquad (4)$$

Again, this is the continuous x version of the distribution. The form is the same in the discrete case, but the normalisation is different. A stretched exponential looks just like a power-law for small values of x, but has a sort of exponential behaviour for large x.

Both of these (depending on the choice of parameters) are slightly curved on a log-log plot, so they are likely to be good fits to the data we have that is not exactly straight. Neither has the long tail characteristic of a power-law, so the curves drop off sharply at the right hand side of a log-log plot.

The distinguishing features of power-laws are therefore 'straightness' in the log-log domain, and not dropping off as fast as the others for large values of x. This is sometimes called a 'fat tail' or 'long tail', in contrast with the 'truncated tail' evident in Figure 1. One potential problem is that the data is poorest in this tail region — our best statistics will be at the non-tail end.

4.1.3 Weighted Least Squares Fits

Fitting a distribution to data means choosing the parameters of the distribution so that it is 'closest' to the data. One way to do this is to minimise the sums of the squares of the differences between the data values and the distribution values.

Suppose the data takes value h_i at x_i, where i runs from 1 to k, the number of data points. If the value of the distribution at x_i is given by $f(\alpha, \beta, x_i)$, where α and β are the parameters of the distribution, we want to choose α and β so that the *residual*:

$$Q = \sum_{i=1}^{k} [h_i - f(\alpha, \beta, x_i)]^2 \qquad (5)$$

is as small as possible.

Weighted least squares fitting is where we use this method but allow for different uncertainties in different data points by

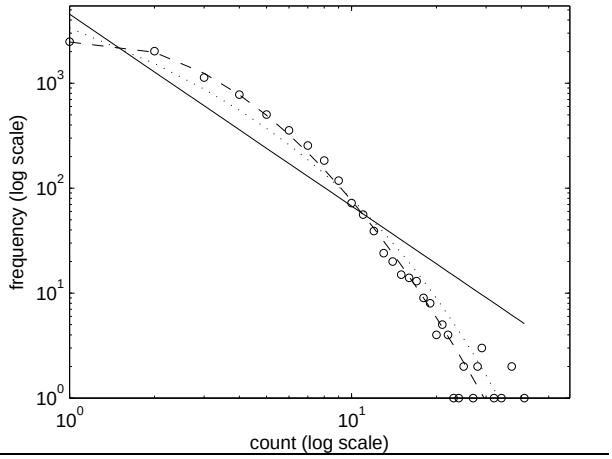

Figure 3. AC distribution and fitted curves for Eclipse. Open circles are data, solid line is best-fit power-law, dashed line is best-fit log-normal and dotted line is best-fit stretched exponential.

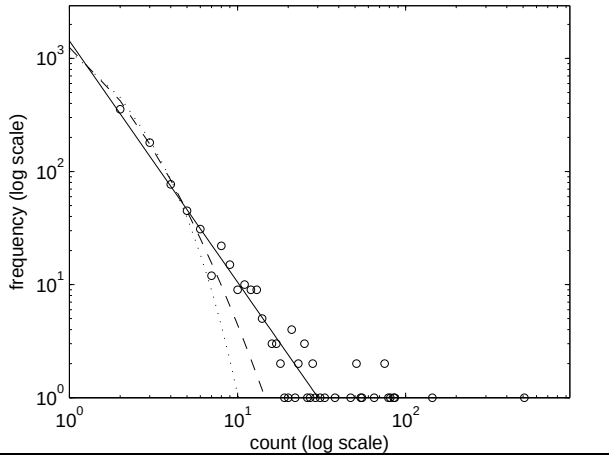

Figure 4. AP distribution and fitted curves for NetBeans.

introducing a weight to each square in the sum:

$$Q = \sum_{i=1}^{k} w_i [h_i - f(\alpha, \beta, x_i)]^2 . \qquad (6)$$

w_i should reflect how much uncertainty there is in the value of a data point. We set $w_i = 1/h_i$. Thus

$$Q = \sum_{i=1}^{k} \frac{1}{h_i} [h_i - f(\alpha, \beta, x_i)]^2 . \qquad (7)$$

4.1.4 Uncertainty and Confidence Intervals

If f is the 'true' distribution, we would have $E[h_i] = f(\alpha, \beta, x_i)$ where $E[z]$ denotes the expected value of z. Expanding each term in (7) and neglecting higher terms we find

$$E[Q] \sim k - 1 \qquad (8)$$

and

$$\mathrm{Var}[Q] = E[(Q - E[Q])^2] \sim k - 2 . \qquad (9)$$

We have assumed h_i is binomially sampled from a distribution with mean f/N, where N is the sample size, $N = \sum_i h_i$.

This gives us a way to estimate how good our fit is. We have effectively a distribution for Q, based on our assumption that the

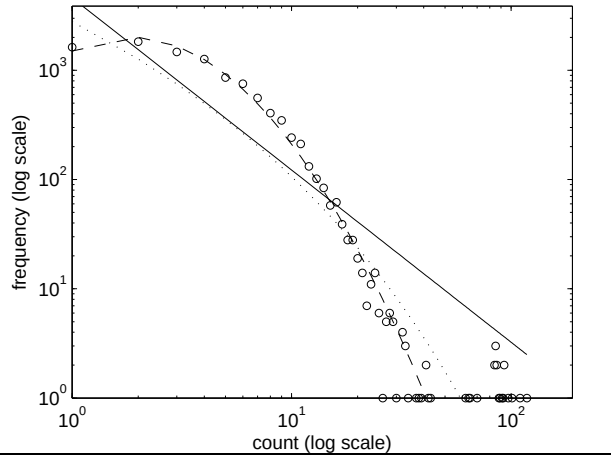

Figure 5. PC distribution and fitted curves for Eclipse.

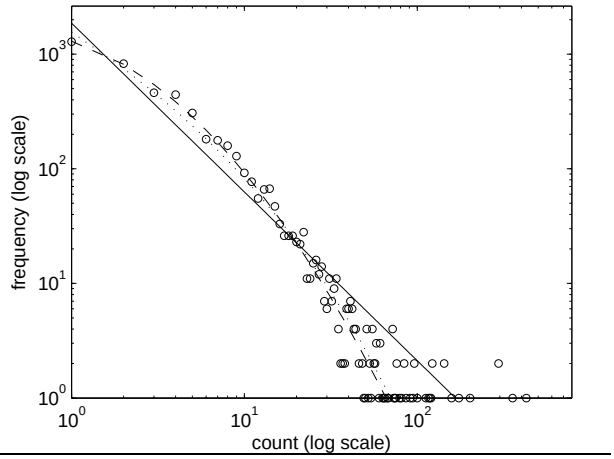

Figure 6. nF distribution and fitted curves for JRE.

data follows the candidate distribution f. We can then choose a *Confidence Interval* (CI) for Q, and if the value for Q that we actually find from our fitting procedure actually falls within this range, we can take this as evidence for our assumption about f.

For example, if the distribution is 'really' the one we have fitted, we would expect Q to be within 1.64σ of $E[Q]$, where $\sigma = \sqrt{\text{Var}[Q]}$, 90% of the time. $E[Q] \pm 1.64\sigma$ is called a 90% confidence interval (CI), and if the minimum value of the residual Q that we *do* get falls within this range, we say that the distribution fits the data at the 90% CI. (This is *not* the same as saying that "we are 90% sure the distribution is right.")

4.1.5 Fitting the data

in the current study, the minimisation of equation (7) was done numerically, with $f(\alpha, \beta, x_i)$ replaced by each of the three distributions (1), (3) and (4) in turn. The raw data is in the form of frequencies occurring at integral values of x. Note that the normalisation of these distributions at discrete values differs from the normalisation of a continuous distribution, and it is important to take this into account. This normalisation depends of course on the parameter values. The log normal and stretched exponential distributions each have two parameters, while the power-law distribution is defined by a single parameter. A second parameter could be introduced by allowing the constant of normalisation to vary (in a log-log plot, a power-law appears as a straight line, with slope given by the single parameter, α, also known as the 'exponent'. The 'offset' of the line is given by the normalisation constant, so fitting an offset parameter is equivalent to fitting the normalisation constant). We found that the fit was very similar when the fit was done with only a single parameter (calculating the normalisation explicitly), returning very similar exponent values and residuals.

The aim of this exercise is mainly to establish the plausibility of the different distributions fitting the data, therefore we do not give uncertainties in the fitted parameters, or speculate on the interpretation of, for example, different fitted power-law exponents.

Table 1 shows a small excerpt from the results of the fit process. This shows the estimated parameters for each of the three distributions using the full datasets: `a_pow` is for power-law, `m_log` and `s_log` are for log-normal, and `a_str` and `b_str` are for the stretched exponential. The next three columns show the residuals for each of the fitted curves, `tot_cnt` is the sum of the frequencies, and the last column is the number of data points.

Recall that the expected value for the residuals is $k - 1$ and the variance is $k - 2$. This means, for the first row of Table 1 (the AC metric) the 90% confidence interval would be 25 ± 8.03 ($1.64 \times \sqrt{24}$), and so we can conclude that the log-normal distribution fits the data at the 90% CI, but the other two distributions do not.

Figure 3 shows an example of a plotted dataset with fitted curves (and is the same as Figure 1). This figure is a log-log plot of the number of types (y-axis) having a given number of fields (x-axis), that is, the AC metric, for Eclipse. The best-fit for a power-law is shown as a solid line, the best fit for the log-normal is shown as a dashed curve, and the best fit for the stretched exponential is a dotted curve. In this case, there is a pronounced curve in the data, and in fact the log-normal has a much better fit than the power-law.

Figures 4-15 show a representative sample of fitted curves for different metrics and different applications. The parameters and residuals for these curves are shown in Table 2.

4.1.6 Summarising the results

For each metric of each program in the corpus, the fits were done first to the whole set of available data, then the number of points was reduced by removing 5, 10, 15, or 20 percent of the data points (or 'cuts') from both ends — that is, using only the 'middle' 90, 80, 70, or 60 percent of the non-zero data points.

The residuals for each fit were then compared for the three distributions. We checked whether each fit was consistent with the data at 95%, 90%, 80% and 60% confidence intervals, and then the power-law fit was compared to the best (residual closest to the expected value) of the other two fits. Each metric for each program could then be classified at each CI with 'flags' as follows:

- a Power-law residual is within the CI and both other residuals outside CI.
- b Power-law residual within CI and one or both of the other residuals within CI.
- c Log normal and/or stretched exponential residual within CI, but power-law residual outside CI.
- d None of the residuals within CI.
- x No data.

Roughly speaking, this order (ignoring x) represents *decreasing* support for the distribution of the data being a power-law. While b does not rule out a power-law, the fact that it fits one of the other candidate distributions indicates more doubt than a indicates. Since we chose our other candidate distributions to be close to power-law, a d suggests that not only do we not have a power-law, but we do not even have something close.

Metric	a_pow	m_log	s_log	a_str	b_str	Q_{pow}	$Q_{\log}$	Q_{str}	tot_cnt	k
AC	1.72	0.62	0.83	0.64	1.03	163.54	32.63	52.79	668	26
AP	3.03	0.52	8.40	1.94	0.95	12.06	410.05	19.61	326	23
DO	1.11	1.56	0.78	0.28	0.63	1004.79	187.74	575.22	1251	97
DOinv	1.12	-3.20	3.80	0.33	0.54	1045.67	684.07	650.27	1251	634
IC	2.12	-0.23	0.85	1.09	0.90	3.91	8.92	12.72	89	14
IP	3.29	0.72	7.79	2.04	0.97	8.18	240.15	2.88	157	9
MS	0.91	2.38	1.20	0.08	0.57	7304.28	1354.52	5545.82	9859	1854
PC	1.65	0.68	0.85	0.61	1.05	254.13	22.11	61.78	1105	18
PP	1.83	-0.30	1.20	0.69	0.89	8.27	14.55	19.22	127	113
PubMC	1.36	0.95	1.14	0.42	0.98	199.13	70.02	110.66	1005	306
RC	1.55	-1.07	1.90	0.52	0.76	994.56	510.30	426.12	1240	38
RP	2.44	-0.30	0.75	1.51	0.86	25.34	41.39	50.76	263	31
SP	1.41	-2.95	8.80	0.57	0.80	37.45	47.79	36.81	133	94
nC	3.07	-0.06	0.50	1.72	0.99	37.32	13.01	32.34	1153	10
nF	1.40	0.72	1.24	0.45	1.03	80.27	36.37	43.36	668	146
nM	1.21	1.22	1.15	0.33	0.93	292.00	113.29	205.25	1170	320
pkgSize	0.92	2.80	2.85	0.00	1.19	13.73	12.53	13.51	72	128

Table 1. The estimated parameters for the three distributions for `arguuml-0.18.1` for the full dataset.

Application	Metric	a_pow	m_log	s_log	a_str	b_str	Q_{pow}	$Q_{\log}$	Q_{str}	k
eclipse	AC	1.82	0.80	0.82	0.58	1.03	3685.96	44.52	701.82	41
eclipse	PC	1.57	1.24	0.79	0.44	0.76	10613.55	214.25	3470.69	118
eclipse	IC	1.91	-0.06	1.12	0.70	1.01	65.96	33.83	64.41	117
eclipse	MS	1.11	2.53	1.22	0.18	0.38	286047.39	13143.66	91823.29	4172
5jre	nF	1.47	0.90	1.29	0.42	1.03	933.12	113.72	229.11	427
jre	nM	1.26	1.54	1.25	0.31	0.92	2374.36	218.22	1084.17	257
jre	nC	2.74	-0.06	0.70	1.16	1.00	340.55	68.30	243.58	14
jre	SP	1.84	-0.03	1.10	0.72	1.01	91.50	46.63	61.28	353
jre	IC	1.86	0.10	0.99	0.73	1.02	74.64	37.64	40.27	451
netbeans	AP	2.13	-0.15	0.95	0.87	0.96	44.61	105.89	184.63	508
netbeans	IP	3.14	-0.02	0.50	1.63	1.00	109.44	11.08	45.33	7
netbeans	PP	1.85	-0.25	1.35	0.58	0.93	93.60	204.40	308.58	618
tomcat	MS	0.89	2.45	1.25	0.00	0.51	10318.07	4334.30	7020.94	1634
tomcat (5% cut)	MS	1.68	1.65	1.75	0.29	0.96	320.63	569.26	285.78	562
openoffice	IP	3.74	0.43	9.15	2.26	0.95	133.66	19713.79	21.95	8
compiere	RC	1.20	1.20	0.62	0.30	0.37	3113.16	457.41	923.42	18

Table 2. Fitted parameters for applications and metrics shown in plots.

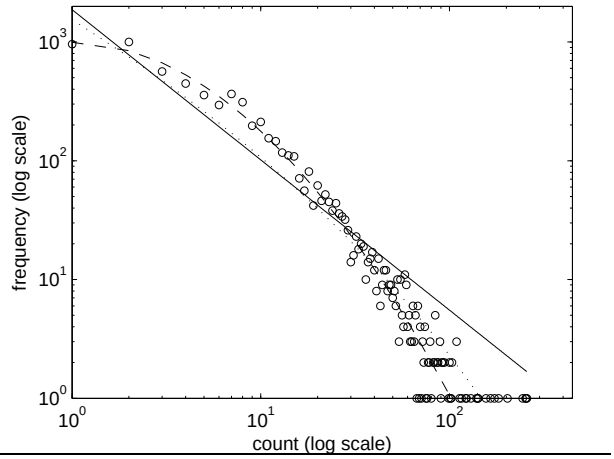

Figure 7. nM distribution and fitted curves for JRE.

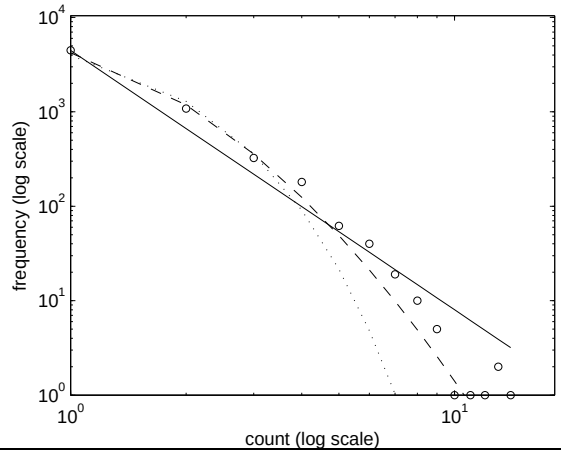

Figure 8. nC distribution and fitted curves for JRE.

	AC	AP	DO	DOinv	IC	IP	MS	PC	PP	PubMC	RC	RP	SP	nC	nF	nM	pkgSize
jeppers	*b*	x	*b*	d	x	x	*b*	*b*	d	*b*	*b*	d	x	*b*	*b*	*b*	*b*
fitjava	*b*	*b*	c	*b*	x	x	c	*b*	*b*	*b*	*b*	*b*	*b*	d	*b*	*b*	d
junit	*b*	c	*b*	*b*	c	x	d	*b*	*b*	*b*	*b*	*b*	c	*b*	*b*	*b*	*b*
jgraph	*b*	*b*	*b*	*b*	*b*	d	d	*b*	*b*	*b*	*b*	*b*	d	*b*	*b*	*b*	*b*
jparse	*b*	*b*	c	d	*b*	d	d	d	*b*	*b*	d	*b*	*b*	*b*	*b*	c	*b*
jaga	*b*	*b*	d	*b*	*b*	x	d	*b*	*b*	c	c	*b*	*b*	c	*b*	c	*b*
joggplayer	*b*	*b*	*b*	*b*	x	x	**a**	c	*b*	*b*	c	d	d	*b*	*b*	*b*	*b*
fitlibraryforfitnesse	c	c	c	*b*	d	d	d	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
javacc	*b*	*b*	*b*	d	*b*	d	d	d	*b*	*b*	*b*	*b*	*b*	c	*b*	*b*	*b*
lucene	d	*b*	c	d	d	x	d	d	*b*	*b*	d	*b*	*b*	*b*	*b*	*b*	*b*
rssowl	*b*	*b*	c	*b*	d	d	d	c	*b*	*b*	c	*b*	*b*	c	*b*	*b*	*b*
sablecc	*b*	*b*	d	d	*b*	x	d	d	d	d	d	*b*	*b*	d	*b*	c	*b*
jag	c	*b*	*b*	c	*b*	x	d	c	*b*	*b*	c	*b*	*b*	c	*b*	*b*	*b*
antlr	c	*b*	c	*b*	*b*	*b*	d	*b*	*b*	*b*	d	*b*	*b*	*b*	*b*	*b*	*b*
jrefactory	*b*	*b*	d	d	c	d	d	d	d	*b*	c	*b*	*b*	d	*b*	*b*	*b*
hsqldb	*b*	*b*	*b*	d	*b*	x	d	d	*b*	*b*	c	*b*	*b*	**a**	*b*	*b*	*b*
jedit	c	*b*	c	c	*b*	d	d	c	*b*	*b*	c	*b*	*b*	**a**	*b*	*b*	*b*
axion	c	*b*	c	c	*b*	d	d	d	*b*	*b*	d	*b*	*b*	**a**	*b*	*b*	*b*
galleon	d	*b*	*b*	*b*	*b*	c	d	c	*b*	*b*	c	*b*	*b*	d	c	*b*	*b*
james	*b*	*b*	*b*	*b*	*b*	d	d	c	*b*	*b*	d	*b*	*b*	d	*b*	*b*	*b*
colt	c	*b*	d	*b*	*b*	d	d	c	*b*	*b*	d	*b*	*b*	d	*b*	*b*	*b*
aglets	d	*b*	c	d	d	*b*	d	c	*b*	*b*	c	*b*	*b*	c	*b*	*b*	*b*
jhotdraw	c	*b*	c	c	*b*	*b*	d	c	*b*	*b*	c	*b*	*b*	c	c	*b*	*b*
ganttproject	c	*b*	c	d	*b*	*b*	d	c	*b*	*b*	c	*b*	*b*	c	*b*	c	*b*
jetty	c	*b*	d	c	*b*	*b*	d	c	*b*	*b*	d	*b*	*b*	c	*b*	*b*	*b*
ireport	*b*	*b*	c	d	*b*	d	d	c	*b*	*b*	d	*b*	*b*	c	*b*	*b*	*b*
jext	*b*	*b*	d	c	*b*	x	d	*b*	*b*	*b*	*b*	*b*	*b*	d	*b*	*b*	*b*
pmd	*b*	*b*	d	c	*b*	d	d	d	d	*b*	c	*b*	*b*	c	*b*	*b*	*b*
aoi	c	*b*	c	d	*b*	d	d	c	*b*	d	d	*b*	*b*	c	c	c	*b*
jung	c	*b*	c	c	*b*	c	d	d	*b*	c	c	*b*	*b*	c	*b*	*b*	*b*
megamek	c	*b*	c	c	*b*	d	d	d	*b*	*b*	d	*b*	*b*	*b*	*b*	*b*	*b*
jfreechart	c	*b*	c	c	*b*	*b*	d	c	*b*	*b*	c	*b*	*b*	c	c	*b*	*b*
poi	c	*b*	d	d	*b*	*b*	d	d	*b*	c	d	c	*b*	d	c	d	*b*
jmeter	d	*b*	c	c	*b*	*b*	d	d	*b*	c	c	*b*	*b*	c	c	c	*b*
glassfish	**a**	*b*	c	d	*b*	*b*	d	c	*b*	*b*	c	*b*	*b*	*b*	*b*	c	*b*
jchempaint	c	*b*	c	c	*b*	d	d	c	*b*	d	d	*b*	*b*	d	d	d	*b*
jasperreports	*b*	*b*	d	d	c	c	d	c	*b*	*b*	d	*b*	*b*	c	*b*	*b*	*b*
scala	*b*	*b*	c	c	*b*	d	d	c	*b*	d	d	*b*	*b*	c	*b*	d	*b*
drjava	*b*	*b*	d	d	c	c	d	d	d	d	d	*b*	d	d	*b*	d	*b*
ant	c	*b*	c	c	*b*	c	d	c	*b*	c	d	*b*	*b*	d	c	c	*b*
sandmark	c	*b*	c	d	*b*	*b*	d	d	*b*	c	d	*b*	*b*	d	c	c	*b*
tomcat	c	*b*	c	c	*b*	*b*	d	d	*b*	*b*	d	*b*	*b*	d	c	*b*	*b*
hibernate	c	*b*	c	d	*b*	c	d	d	*b*	c	d	*b*	c	d	c	c	*b*
sequoiaerp	c	*b*	d	c	*b*	d	d	c	*b*	c	d	*b*	*b*	d	c	c	*b*
columba	c	*b*	c	c	*b*	c	d	d	*b*	c	c	*b*	*b*	d	c	c	*b*
argouml	c	**a**	d	d	*b*	c	d	c	*b*	c	d	d	*b*	c	c	c	*b*
compiere	c	*b*	d	d	*b*	d	d	d	*b*	d	d	*b*	c	c	*b*	d	*b*
derby	c	*b*	c	d	*b*	c	d	c	*b*	c	d	*b*	*b*	d	c	c	*b*
azureus	c	*b*	c	d	*b*	d	d	d	*b*	c	c	c	*b*	c	c	c	c
geronimo	d	*b*	d	c	**a**	c	d	d	d	d	d	d	*b*	d	c	d	*b*
jtopen	d	*b*	d	d	*b*	d	d	d	*b*	c	d	*b*	*b*	d	c	c	*b*
openoffice	c	*b*	d	c	*b*	d	d	c	*b*	d	c	*b*	*b*	d	d	d	*b*
jboss	c	*b*	d	d	*b*	c	d	d	c	c	d	c	*b*	d	c	c	d
jre	c	*b*	d	d	c	d	d	d	*b*	d	d	*b*	c	d	c	c	*b*
netbeans	d	**a**	d	d	*b*	c	d	d	**a**	c	d	c	*b*	d	c	c	c
eclipse	c	**a**	d	d	*b*	d	d	d	**a**	c	d	*b*	c	d	c	c	c

Table 3. Quality of fit at Confidence Interval 80% for full dataset: a–good fit only to power-law, b–good fits to more than one curve, c–good fit only to other curves, d–no good fits. Applications are ordered by increasing size (number of classes).

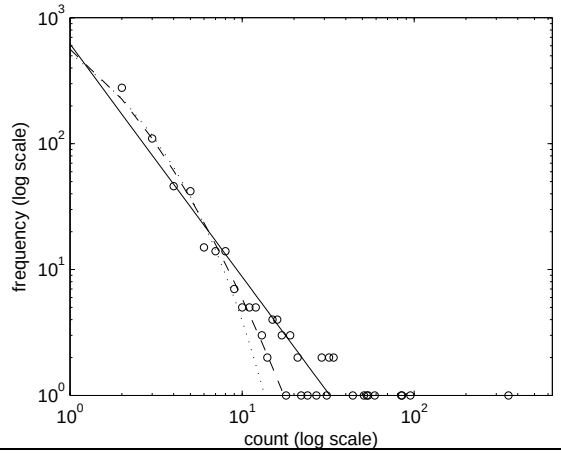

Figure 9. SP distribution and fitted curves for JRE.

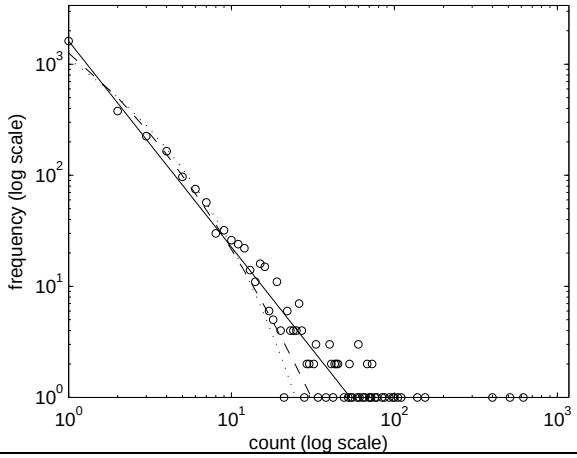

Figure 12. PP distribution and fitted curves for NetBeans.

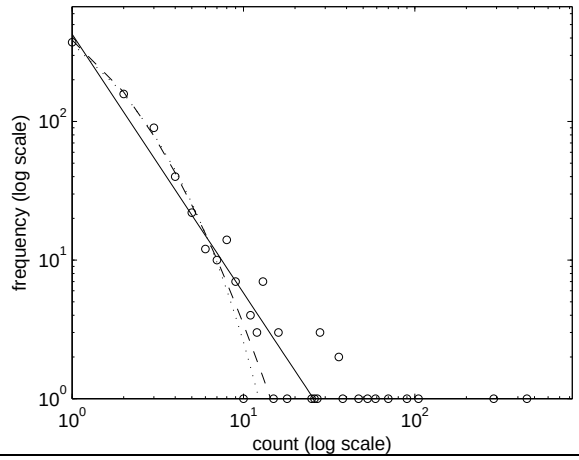

Figure 10. IC distribution and fitted curves for JRE.

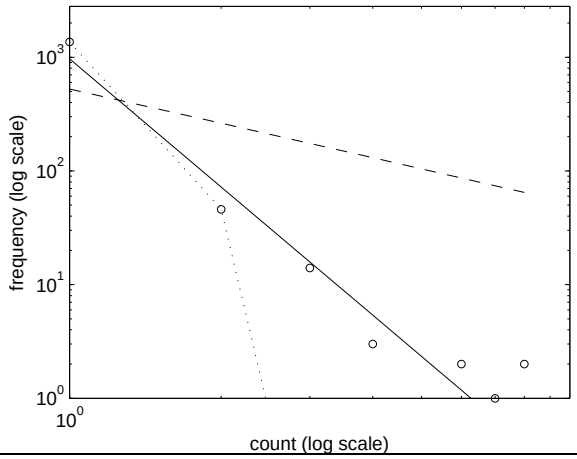

Figure 13. IP distribution and fitted curves for Openoffice.

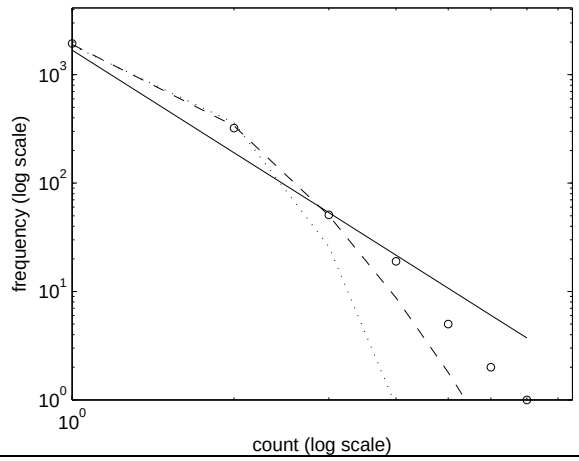

Figure 11. IP distribution and fitted curves for NetBeans.

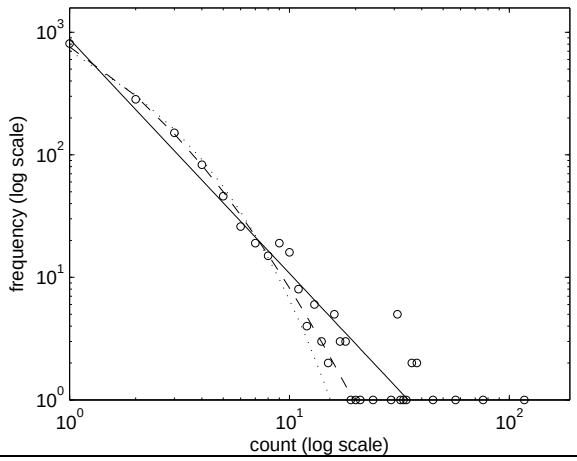

Figure 14. IC distribution and fitted curves for Eclipse.

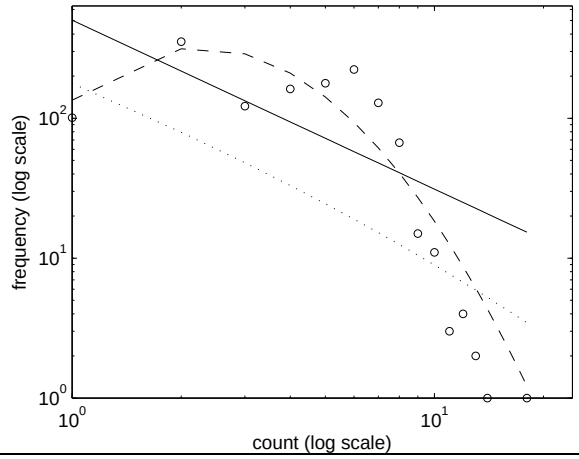

Figure 15. RC distribution and fitted curves for Compiere.

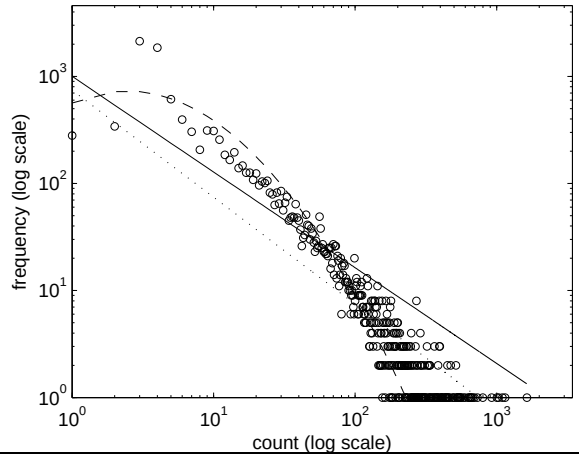

Figure 16. MS distribution and fitted curves for Tomcat.

Table 3 shows these results for the 80% CI and using complete datasets (0% cuts). In this table, the applications are ordered in increasing size, as measured by number of classes. The four groups are: applications with fewer than 200 classes, applications with fewer than 500 classes, applications with fewer than 1000 classes, and those with more than 1000 classes. To aid comprehension, we use different typefaces for the entries.

For the moment, we will just note patterns and trends, and leave interpretation and discussion to the next section. The first thing to note (other than the sheer size), is that, while all values are represented, b (multiple distributions have good fits) is quite prominent. The next point is that a (good fit only to power-law) is relatively rare.

Looking at individual metrics for the larger applications (last category), we note that AC, PC, and RC tend to have c and d, indicating lack of support for them having a power-law distribution, whereas their opposites, AP, PP, and RP, as well as SP, tend to have a and b. In almost all cases, however, there are exceptions for individual applications. IC and IP show the opposite trend, with IC having mainly a and b and IP having mainly c and d.

It must be kept in mind that Table 3 represents only 5% of the results of the curve fitting (which itself represents a summarisation of the original data) — there are the other CIs and cuts. What the results show for the other cuts and CIs is what one would expect. As the cut size increases, meaning the highest and lowest frequency

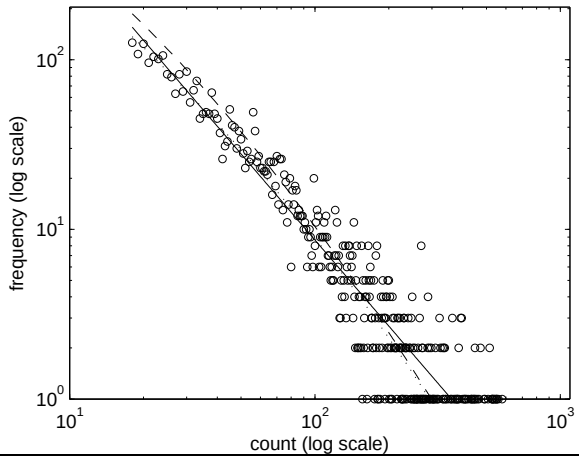

Figure 17. MS distribution and fitted curves for Tomcat after a 5% cut.

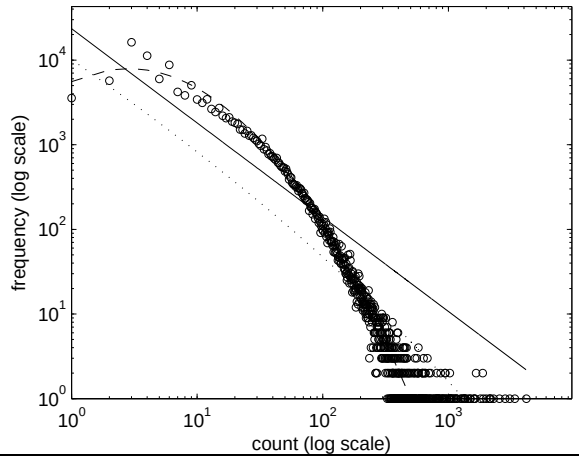

Figure 18. MS distribution and fitted curves for Eclipse.

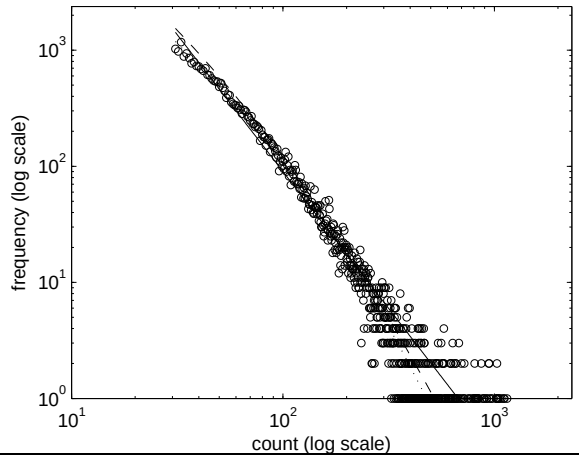

Figure 19. MS distribution and fitted curves for Eclipse after a 5% cut.

data (where most of the variation occurs) is removed, we get better fits for all three distributions (that is, tending toward b). Similarly, as the CI is increased, it also becomes easier to get a good fit.

We chose to show the 80% CI as it seemed the most representative. The 60% CI is not that different from what is shown in Table 3, and all of the differences are what one would expect — more d's (no good fits) at 60% than at 80% or tending toward b when going from 60% to 80%.

To finish this section, we show a few more fitted curves. In this case, Figures 16-19, we show various MS distributions. These are interesting as they have many more data points than the others, being based on methods not types. We also show the effect of applying a 5% cut.

5. Discussion

5.1 Interpretation

Recall that several of our metrics measure 5 inter-type relationships — Inheritance (SP), Aggregation (AC and AP), Parameter (PC and PP), Return (RC and RP), and Interface (IC and IP). The 'C' variant of the metric for a relationship measures the 'client' end and 'P' the 'provider' end. Or, if the code were represented as a directed graph with types as vertices and the different relationships as edges, then 'C' would be the out-degree and 'P' the in-degree for each relationship of each vertex. We note that out-degree is impacted by decisions made with respect to the type represented by the vertex, whereas in-degree is the result of decisions made with respect to other types.

In the previous section, we noted that AC, PC, and RC distributions tended not to have good fits to a power-law, but AP, PP, RP, and SP did. From the comments above, this suggests out-degree distributions are not power-laws but in-degree are. The distributions we are seeing for the 'C' metrics tend to be truncated at the high-value (low-frequency) end. A person changing the code for a class is inherently aware of its outward dependencies (e.g. the number of types it uses or the number of interfaces it implements), but they are not inherently aware of the number of classes that subtype it or call methods on it. They therefore have less control over the latter than they do over the former. Furthermore, we believe there is a tendency is to avoid (consciously or subconsciously) 'big things', whether due to difficulty of management (e.g., methods with many parameters) or simply through training ("Don't write big classes!"). This suggests that 'C' relationships are more likely than 'P' relationships to have 'truncated' curves. We can generalise this to hypothesise that any metric that measures something that the programmer is inherently aware of will tend to have a 'truncated' curve, that is, not be a power-law.

The nF, nM, and PubMC, distributions are explained by our hypothesis. They are all aspects of a type description that the developer is inherently aware of, and all tend not to have support for power-laws.

Unfortunately our hypothesis does not explain the IC and IP distributions. We believe that the main cause of the poor fits for the IP distributions is the small datasets (no more than 11 data points, and see for example Figure 13). This, however, does not explain IC (e.g., Figures 10 and 14). nC also suffers from having small datasets, which might explain the results we see.

DO and DOinv are related — DO is the 'client' end, and DOinv the 'provider'. However in this case there is not a strong distinction between the two, both being c and d. The DO relationship is effectively including all of AC, PC, RC, and IC, as well as types used for local variables. This would mean that the behaviour of IC noted above would oppose the behaviour of the others, which may explain the results. We do know that types used for local variables (or rather, not used in the published interface) do account for significant dependency structures [19].

MS, with few exceptions (all small applications), does not fit any distribution at the 80 CI. However, at 90 CI and above, there are good fits to all of them. Our hypothesis would suggest this should be a truncated curve (the size of the method being a decision made as it is written) but it would seem that there is too much noise to be sure.

There is another important point to make. There is quite noticeable variation on the degree of fit between different applications. This raises an interesting question: if a given relationship (metric) does follow a particular distribution, why do we not see this distribution for all applications, how is it that this variation exists?

Two answers spring to mind. The first is that different applications come from different domains, and it is possible that different domains have different distributions. For example, NetBeans and Openoffice often have different values (usually c vs d or a vs d). NetBeans is an IDE, whereas Openoffice is an office suite, and in fact is really several applications wrapped as one. We picked these two because they were both originally Sun products. That said, Compiere is ERP and seems somewhat different in nature than, for example, Openoffice, and yet the distributions seem mainly similar.

Another answer is that there is another thing that is potentially quite different (and much harder to see) between the applications — their design. If we are seeing different distributions due to different designs, if we could understand how aspects of the design related to the kind of distribution exhibited, there is the potential for developing a *quantitative measure* for design quality. Having such a measure could have tremendous impact on how software is developed in the future.

Of course before this can happen, we must understand (presuming such a relationship exists) *which* distribution corresponds to a good design and which does not. It is not obvious that, for example, the power-law distribution is found in 'good' designs — it could just as easily be the opposite! Our results do not provide much advice either way. This does, however, suggest an extremely interesting avenue for future research.

5.2 Threats to Validity

The most likely threat to the validity of our conclusions is the corpus we used. It consists entirely of open-source applications of small to medium size. Some applications originated from commercial organisations, but it is not obvious that the IBM and Sun-donated code is typical of closed-source code. Other studies have suggested there is little difference between open-source and closed-source software [19], but we cannot say whether or not this is true here. While we cannot claim that our corpus represents a random sample of Java software, our situation is no different than corpora used in applied linguistics. Hunston describes a number of ways corpora may be reasonably used [14]. Our corpus is what she describes as a reference corpus, which are often used as base-line for further studies. Thus, a random sample is not necessary in order to produce an valid result. Our results hold for what is in our corpus: whether or not they hold for other collections will in itself be of interest.

So we cannot say for sure how representative our corpus is of Java software in general, or even open-source software in particular. Nevertheless, the commonality we have seen across all of the applications we analyse gives us confidence that our conclusions will hold generally.

A similar issue is that our corpus consists only of Java applications. It is possible we may see different distributions when looking at other languages such as C# or C++. While there appears nothing obviously different between Java and languages such as C# or C++ with respect to our study, they do share the property of having

static type checking, so while we may see no differences for such languages, we may see differences in languages, such as Smalltalk, that do not have static type checking.

A property of the software we have studied that we have not addressed in our study is the manner in which the software was created. Our hypothesis is based on the lack of global view a developer has of the application being developed. Recently, there has been a significant increase in the use of sophisticated Integrated Development Environments (IDE) such as Eclipse, and one characteristic of these IDEs is that they provide a better view of the source code than has been available in the past. The use of such IDEs may affect the shape of the distributions we have been investigating. We believe most of the code in our corpus was written before the advent of such IDEs, but some of the variation we see may be due to how the code was written. Again Smalltalk may show differences as it has always had an IDE.

As noted earlier, because we measure from byte code, there is some information from the source code not available to us. The circumstances for which this is the case seem to be such that this will be rare.

6. Related Work

As with many other things, Knuth was one of the first to carry out empirical studies to understand what code that is actually written looks like [16]. He presented a static analysis of over 400 FORTRAN programmes and dynamic analysis of about 25 programs. His main motivation was compiler design, with the concern that compilers may not optimise for the typical case as no-one knew what the typical case was. His analysis was at the statement level, counting such things as the number of occurrences of an `IF` statement, or the number of executions of a given statement.

Collberg et al. have carried out a study of 1132 Java programs [4]. These were gathered by searching for `jar` files with Google and removing any that were invalid. Their main goal was the development of tools for protection of software from piracy, tampering, and reverse engineering. Like Knuth, they argued that their tools could benefit by knowing the typical and extreme values of various aspects of software. Consequently, their interest is in the low-level details of the code with a view toward future tool support or language design.

Although their interest is in low-level details, Collberg et al. do gather a number of similar statistics to ours, such as number of classes per package, number of fields per class, number of methods per class, size of the constant pool, and so on. However comparison with their results is problematic, as they appear to include all classes referred to in an application, whereas we only consider classes that appear in the application source.

Gil and Maman analysed a corpus of 14 Java applications for the presence of *micro patterns*, patterns at the code level that represent low-level design choices [10]. They found that 3 out of 4 classes matched one of the 27 micro patterns in their catalogue, and just over half of the classes are catalogued by just 5 patterns. This is a form of structural analysis, however it focuses on individual classes, rather than at the application level as we have done.

As already mentioned, Wheeldon and Counsell have performed a similar analysis to ours. They looked at JDK 1.4.2, Ant 1.5.3, and Tomcat 4.0. They computed the 12 metrics as noted in section 3 and concluded that what they were seeing were power-laws. There are some differences between their work and ours. Most notably is how the metrics were computed. Wheeldon and Counsell used a custom doclet to extract the relevant information, which limited them to just the information available from the Javadoc comments. Also, they were not specific as to what choices they made for the variables discussed in section 3.

We believe the inconsistency between Wheeldon and Counsell's conclusions and ours is due to our more extensive corpus. Our original intention was to reproduce their study and, we thought, results. The 'truncated-curve' distribution only really became apparent in the repetition across multiple applications. In fact, their figure 2(b) appears to have something of a curve to it. Our work does, however, add significant evidence to support their hypothesis that there are regularities that are common across all non-trivial Java programs.

7. Conclusion

We have studied the hypothesis that the distribution of a number of metrics on object-oriented software obey a power-law. We did so over a larger sample size than has been considered by past similar studies, and applied analysis techniques to characterise how closely each distribution obeyed a power-law. We have presented our method and analysis in what we hope is sufficient detail to allow our studies to be reproduced with confidence.

What we found was that while there were distributions for which there was good evidence for a power-law, there are a number for which there was little evidence that a power-law exists. This is in contrast with what earlier studies have suggested. We hypothesise that any metric that measures a relationship that the programmer is inherently aware of will tend to have a 'truncated' curve, that is, not be a power-law.

Of particular interest is the fact that some applications frequently differed for some metrics from the other applications, indicating that *some attribute of the application's code can affect the resulting distribution*. This finding has potentially tremendous implications. If the distribution does depend on either design quality or domain, then knowing the distribution of a 'good' design would provide a much sounder foundation for developing software than currently exists. As open-source applications make extensive use of version control and bug-tracking systems, we believe the data necessary for such studies as correlations between distribution and prevalence of defects will be possible.

There remains much work to be done. Further studies are needed to determine how representative our findings are. This means expanding the studies to other (especially larger) applications, to applications developed in other environments, such as closed-source, to other domains (for example, real-time software is not represented in our corpus at the moment), and to other languages.

We need to be able to explain why we see some distributions in some applications for some metrics and not others. For example, we need models that explain how these distributions arise. In the case of power-law distributions, there is no theory to explain why we should see such scale-free structures in software. Two main hypothetical mechanisms have been put forward [1] to account for the origin of scale-free network structure in other domains: growth with preferential attachment [2], in which existing nodes link to new nodes with probability proportional to the number of links they already have, and hierarchical growth [33] in which networks grow in an explicitly self-similar fashion. Additionally arguments from optimal design have been proposed [30, 28]. It is still far from clear, however, what (if any) fundamental theory might account for the ubiquity of the phenomenon in software.

Ultimately, we need to understand the relationship between large-scale structures found in software, and quality attributes such as understandability, modifiability, testability, and reusability. We believe this study is an important step toward that goal.

Acknowledgements

We would like to thank the anonymous referees for their comments and suggestions for improving this paper.

References

[1] A. Barabasi. *Linked: the New Science of Networks*. Perseus Press, New York, 2002.

[2] A. L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[3] D. W. Clark and C. C. Green. An empirical study of list structure in lisp. *Commun. ACM*, 20(2):78–87, 1977.

[4] C. Collberg, G. Myles, and M. Stepp. An empirical study of Java bytecode programs. Technical Report TR04-11, Department of Computer Science, Univeristy of Arizona, 2004.

[5] S. Dieckmann and U. Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *13th European Conference on Object-Oriented Programming*, pages 92–115, 1999.

[6] K. Ehrlich and E. Soloway. *Human factors in computer systems*, chapter An empirical investigation of the tacit plan knowledge in programming, pages 113–133. Ablex Publishing Corp., Norwood, NJ, USA, 1984.

[7] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. PWS Publishing Company, second edition, 1997.

[8] J. Frederick P. Brooks. Three great challenges for half-century-old computer science. *J. ACM*, 50(1):25–26, 2003.

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

[10] J. Y. Gil and I. Maman. Micro patterns in Java code. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 97–116, New York, NY, USA, 2005. ACM Press.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(tm) Language Specification*. Addison-Wesley, 2000.

[12] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 241–255, New York, NY, USA, 2001. ACM Press.

[13] J. Heering. Quantification of structural information: on a question raised by Brooks. *SIGSOFT Softw. Eng. Notes*, 28(3):6–6, 2003.

[14] S. Hunston. *Corpora in Applied Linguistics*. Cambridge University Press, 2002.

[15] C. Jones. *Programming productivity*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[16] D. E. Knuth. An empirical study of FORTRAN programs. *Software–Practice and Experience*, 1(2):105–133, 1971.

[17] P. Kruchten. *The Rational Unified Process: An Introduction, Second Edition*. Addison-Wesley, United States of America, 2000.

[18] J. Laherrere and D. Sornette. Stretched exponential distributions in nature and economy: "fat tails" with characteristic scales. *The European Physical Journal B*, 2:525, 1998.

[19] H. Melton and E. Tempero. An empirical study of cycles among classes in Java. Technical Report UoA-SE-2006-1, Department of Computer Science, University of Auckland, 2006.

[20] H. Melton and E. Tempero. Identifying refactoring opportunities by identifying dependency cycles. In V. Estivill-Castro and G. Dobbie, editors, *Twenty-Ninth Australasian Computer Science Conference*, Hobart, Tasmania, Australia, Jan. 2006. Proceedings published as "Conferences in Research and Practice in Information Technology, Vol. 48".

[21] M. E. J. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, 46(5):323–351, Sept. 2005.

[22] J. Noble and R. Biddle. Visualising 1,051 visual programs module choice and layout in the nord modular patch language. In *CRPITS '01: Australian symposium on Information visualisation*, pages 121–127, Darlinghurst, Australia, Australia, 2001. Australian Computer Society, Inc.

[23] J. Noble and R. Biddle. *Software Visualization*, chapter Visual Program Visualisation. Kluwer, 2003.

[24] Object Management Group. Unified Modeling Language (UML) 1.5 specification, 2004.

[25] A. Potanin, J. Noble, and R. Biddle. Checking ownership and confinement. *Concurrency - Practice and Experience*, 16(7):671–687, 2004.

[26] A. Potanin, J. Noble, M. Frean, and R. Biddle. Scale-free geometry in OO programs. *Commun. ACM*, 48(5):99–103, 2005.

[27] S. Purao and V. Vaishnavi. Product metrics for object-oriented systems. *ACM Comput. Surv.*, 35(2):191–221, 2003.

[28] R. V. Solé, R. Ferrer-Cancho, J. M. Montoya, and S. Valverde. Selection, tinkering, and emergence in complex networks. *Complex.*, 8(1):20–33, 2002.

[29] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[30] S. Valverde, R. Ferrer-Cancho, and R. V. Solé. Scale-free networks from optimal design. *Europhysics Letters*, 60(4):512–517, Nov. 2002.

[31] S. Valverde and R. V. Solé. Hierarchical small-worlds in software architecture. Under review, *IEEE Transactions in Software Engineering*. An earlier versino is available as Sante Fe Institute Working Paper 03-07-044, 2005.

[32] W. Weibull. A statistical distribution function of wide applicability. *ASME Journal Of Applied Mechanics*, pages 293–297, Sept. 1951.

[33] G. M. Weinberg. *The Psychology of Computer Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1985.

[34] R. Wheeldon and S. Counsell. Power law distributions in class relationships. In *Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM03)*, 2003.

Appendix A: Formal definitions for Metrics

This appendix contains more formal definitions of the metrics we compute as computed from the byte code (the `.class` files). As mentioned in Section 3, the definitions assume one *top-level*[11] type declaration per source file (`.java` file).

Let $\mathcal{S}$ = the set of *source files* in the application under consideration. Under our assumption, every top-level class C is declared in a source file `C.java` that in turns generates a file `C.class` (which is what is used to compute the metrics). We express many of the metrics in terms of source files because it makes some definitions easier to explain.

We define the following notation:

- For top-level classes A and B, A DEPENDS ON B if B's name appears in the constant pool of `A.class`.
- For a type T, and file u, T IS DECLARED IN u is true if and only if there is a declaration for T in u. Note that u is not necessarily `T.java`, it could be the equivalent of B in the example above, and T could also be a inner type declaration.
- $\mathcal{T} = \{T | T \text{ IS DECLARED IN } u, u \in \mathcal{S}\}$. Note that this set is not just the top-level types, but also includes inner types.
- METHODS(T) = the set of methods declared in T (appear in the `.class` files).
- FIELDS(T) = the set of fields declared in T (appear in the `.class` files).
- ISCLASS(T) is true iff T is a class.
- ISINTERFACE(T) is true iff T is an interface.
- ISCONSTRUCTOR(c) is true iff c is a constructor.
- For $C, D \in \mathcal{T}$ where $(\text{ISCLASS}(C) \wedge \text{ISCLASS}(D) \vee \text{ISINTERFACE}(C) \wedge \text{ISINTERFACE}(D))$ is true, C EXTENDS D if D appears in C's `extends` clause in its declaration.
- For $C, I \in \mathcal{T}$ where $(\text{ISCLASS}(C) \wedge \text{ISINTERFACE}(I))$ is true, C IMPLEMENTS I if I appears in C's `implements` clause

The following definitions apply only to top-level type declarations. We will use the conventions that C and D refer to classes, I refers to an interface, T refers any type, u refers to a source file, m refers to a method, and f refers to a field.

Number of Methods $\texttt{nM}(C) = |\text{METHODS}(C)|$

Number of Fields $\texttt{nF}(C) = |\text{FIELDS}(C)|$

Number of Constructors $\texttt{nC}(C) = |\{m : m \in \text{METHODS}(C),\ \text{ISCONSTRUCTOR}(m)\}|$

Subclasses $\texttt{SP}(C) = |\{u : u \in \mathcal{S}, \exists D, D \text{ IS DECLARED IN } u, D \text{ EXTENDS } C\}|$

Implemented Interfaces $\texttt{IP}(I) = |\{u : u \in \mathcal{S}, \exists D, D \text{ IS DECLARED IN } u, D \text{ IMPLEMENTS } I\}|$

Interface Implementations $\texttt{IC}(T) = |\{u : u \in \mathcal{S}, \exists I, \text{ISINTERFACE}(I),\ I \text{ IS DECLARED IN } u,\ T \text{ IMPLEMENTS } I\}|$ if $\text{ISCLASS}(T)$
$|\{u : u \in \mathcal{S}, \exists I, \text{ISINTERFACE}(I),\ I \text{ IS DECLARED IN } u,\ T \text{ EXTENDS } I\}|$ if $\text{ISINTERFACE}(T)$

References to class as a member $\texttt{AP}(C) = |\{u : u \in \mathcal{S}, \exists D, D \text{ IS DECLARED IN } u, C \text{ IS FIELDTYPE OF } D\}|$

Members of class type $\texttt{AC}(C) = |\{T : T \in \mathcal{T}, T \text{ IS FIELDTYPE OF } C\}|$

References to class as a parameter $\texttt{PP}(C) = |\{u : u \in \mathcal{S}, \exists D, D \text{ IS DECLARED IN } u, C \text{ IS PARAMETERTYPE OF } D\}|$

Parameter-type class references $\texttt{PC}(C) = |\{T : T \in \mathcal{T}, \exists m \in \text{METHODS}(C), T \text{ IS PARAMETERTYPE OF } m\}|$

References to class as return type $\texttt{RP}(C) = |\{u : u \in \mathcal{S}, \exists D, D \text{ IS DECLARED IN } u, \exists m \in \text{METHODS}(D), C \text{ IS RETURNTYPE OF } m\}|$

Methods returning classes $\texttt{RC}(C) = |\{T : T \in \mathcal{T}, \exists m \in \text{METHODS}(C), T \text{ IS RETURNTYPE OF } m\}|$

Depends On $\texttt{DO}(C) = |\{u : u \in \mathcal{S}, \exists D, D \text{ IS DECLARED IN } u, C \text{ DEPENDS ON } D\}|$

Inverse of Depends On $\texttt{DOinv}(C) = |\{u : u \in \mathcal{S}, \exists D, D \text{ IS DECLARED IN } u, D \text{ DEPENDS ON } C\}|$

Public Method Count $\texttt{PubMC}(C) = |\{m : m \in \text{METHODS}(C), \text{ISPUBLIC}(m)|$

Package Size $\texttt{PkgSize}(p)$ = number of top-level classes in p.

Method size $\texttt{MS}(m)$ = number of byte code instructions in m.

Note that this is not the number of bytes needed to represent the method.

Appendix B: Corpus details

This appendix provides the details of the part of the corpus used in this study. We use the standard naming scheme for each application, which typically includes some kind of version identification. The domain comes from our assessment based on the application documentation. We identify where we acquired the source code. The column "O/C" refers to whether the application can be considered open or closed source (all applications used here are open source). The column "V" identifies where we have multiple versions (we only used the latest version in this study). Finally, any notes that seem relevant are provided.

Application	*#*	*Domain*	*Origin*	*O/C*	*V*	*Notes*
aglets-2.0.2	280	Framework for developing mobile agents	Sourceforge	O	N	Donated by IBM
ant-1.6.5	700	Java build tool	Apache	O	Y	
antlr-2.7.5	209	Parser generator	`antlr.org`	O	N	
aoi-2.2	415	3D modelling and rendering	Sourceforge	O	N	
argouml-0.18.1	1251	UML drawing/critic	`tigris.org`	O	Y	
axion-1.0-M2	237	SQL database	`tigris.org`	O	N	
azureus-2.3.0.4	1650	P2P filesharing	Sourceforge	O	Y	
colt-1.2.0	269	High performance collections library	`hoschek.home.cern.ch`	O	Y	
columba-1.0	1180	Email client	Sourceforge	O	N	
compiere-251e	1372	ERP and CRM	Sourceforge	O	N	
derby-10.1.1.0	1386	SQL database	Apache Jakarta	O	N	Donated by IBM
drjava-20050814	668	IDE	Sourceforge	O	N	
eclipse-SDK-3.1-win32	11413	IDE	`www.eclipse.org`	O	Y	Donated by IBM
fitjava-1.1	37	Automated testing	`fit.c2.com`	O	N	
fitlibraryforfitnesse-20050923	124	Automated testing	Sourceforge	O	N	
galleon-1.8.0	243	TiVo media server	Sourceforge	O	N	
ganttproject-1.11.1	310	Gantt chart drawing	Sourceforge	O	N	
geronimo-1.0-M5	1719	J2EE server	Apache	O	N	
glassfish-9.0-b15	582	J2EE server	`dev.java.net`	O	N	
hibernate-3.1-rc2	902	Persistence object mapper	Sourceforge	O	N	
hsqldb-1.8.0.2	217	SQL database	Sourceforge	O	N	
ireport-0.5.2	347	Visual report design for JasperReports	Sourceforge	O	N	
jag-5.0.1	208	J2EE application generator	Sourceforge	O	N	
jaga-1.0.b	100	API for genetic algorithms	`jaga.org`	O	N	
james-2.2.0	259	Enterprise mail server	Apache	O	N	
jasperreports-1.1.0	633	Reporting tool	Sourceforge	O	N	
javacc-3.2	125	Parser generator	`dev.java.net`	O	N	
jboss-4.0.3-SP1	4143	J2EE server	Sourceforge	O	N	

Continued on next page

Table 4 – continued from previous page

Application	*#Classes*	*Domain*	*Origin*	*O/C*	*V*	*Notes*
jchempaint-2.0.12	612	Editor for 2D molecular structures	Sourceforge	O	N	
jedit-4.2	234	Text editor	Sourceforge	O	N	
jeppers-20050607	20	Spreadsheet editor	Sourceforge	O	N	
jetty-5.1.8	327	HTTP Server/servlet container	Sourceforge	O	N	
jext-5.0	353	IDE	Sourceforge	O	N	
jfreechart-1.0.0-rc1	469	Chart drawing	Sourceforge	O	N	
jgraph-5.7.4.3	50	Graph drawing	Sourceforge	O	Y	
jhotdraw-6.0.1	300	Graph drawing	Sourceforge	O	Y	
jmeter-2.1.1	560	Java performance testing	Apache	O	Y	
joggplayer-1.1.4s	114	MP3 player	`joggplayer.webarts.bc.ca`	O	N	
jparse-0.96	69	Java compiler front-end	`www.ittc.ku.edu/JParse`	O	N	
jre-1.4.2.04	7257	JRE	sun.com	O	N	
jrefactory-2.9.19	211	Refactoring tool for Java	Sourceforge	O	N	
jtopen-4.9	2857	Java toolbox for iSeries and AS/400 servers	Sourceforge	O	N	Donated by IBM
jung-1.7.1	454	Graph drawing	Sourceforge	O	Y	
junit-3.8.1	48	Unit testing framework	Sourceforge	O	N	
lucene-1.4.3	170	Text indexing	Apache	O	Y	
megamek-2005.10.11	455	Game	Sourceforge	O	N	
netbeans-4.1	8406	IDE	netbeans.org	O	Y	Donated By Sun
openoffice-2.0.0	2925	Office suite	openoffice.org	O	N	Donated By Sun
pmd-3.3	375	Java code analyser	Sourceforge	O	N	
poi-2.5.1	480	API to access Microsoft format files	Apache	O	N	
rssowl-1.2	189	RSS Reader	Sourceforge	O	N	
sablecc-3.1	199	Compiler/Interpreter generating framework	Sourceforge	O	N	
sandmark-3.4	837	Software watermarking/security	`www.cs.arizona.edu/sandmark`	O	N	
scala-1.4.0.3	654	Multi-paradigm programming language	`scala.epfl.ch`	O	N	
sequoiaerp-0.8.2-RC1-all-platforms	936	ERP and CRM	Sourceforge	O	N	
tomcat-5.0.28	892	Servlet container	Apache	O	N	