Comparative Analysis of Evolving Software Systems Using the Gini Coefficient*

Rajesh Vasa, Markus Lumpe, Philip Branch
Faculty of Information & Communication Technologies
Swinburne University of Technology
P.O. Box 218, Hawthorn, VIC 3122, AUSTRALIA
{rvasa,mlumpe,pbranch}@swin.edu.au

Oscar Nierstrasz
Institute of Computer Science
University of Bern
Bern, CH-3012, SWITZERLAND
oscar@iam.unibe.ch

Abstract

Software metrics offer us the promise of distilling useful information from vast amounts of software in order to track development progress, to gain insights into the nature of the software, and to identify potential problems. Unfortunately, however, many software metrics exhibit highly skewed, non-Gaussian distributions. As a consequence, usual ways of interpreting these metrics — for example, in terms of "average" values — can be highly misleading. Many metrics, it turns out, are distributed like wealth — with high concentrations of values in selected locations. We propose to analyze software metrics using the Gini coefficient, a higher-order statistic widely used in economics to study the distribution of wealth. Our approach allows us not only to observe changes in software systems efficiently, but also to assess project risks and monitor the development process itself. We apply the Gini coefficient to numerous metrics over a range of software projects, and we show that many metrics not only display remarkably high Gini values, but that these values are remarkably consistent as a project evolves over time.

1. Introduction

What is the inherent nature of software? Do software systems form "perfect" societies with an equal distribution of responsibilities, or are they *polarized*, where some parts have to shoulder most of the load and others are just simple service providers? These are questions of more than passing interest. By understanding what typical and successful software evolution looks like, we can identify anomalous situations and perhaps take action earlier than might otherwise be possible. However, we are only beginning to understand how change and distribution of functionality affect evolving software systems [23, 29, 30].

A standard technique [8, 15, 16] to answer these questions is to identify a number of characterizing properties, collect corresponding software metrics, and render the obtained data into meaningful information that can assist both developers and project managers in their decision making [13, 27]. Unfortunately, software metrics data are, in general, heavily skewed [7, 12, 30], which makes precise interpretation with standard descriptive statistical analysis difficult. Summary measures like "average" or "mean" assume a *Gaussian* distribution to capture the *central tendency* in a given data set. However, when applied to non-Gaussian distributions, central tendency measures become increasingly more unreliable the greater the distance is between a given distribution and a normal distribution.

The shortcomings of central tendency measures are amplified when we wish to compare skewed distributions. Any meaningful comparison requires additional effort to fit the distributions in question to a specially-designed third model distribution [1, 26]. This transformation is not only cumbersome but also expensive and may not yield the desired result. Moreover, additional problems may arise due to changes in both the degree of concentration of individual values and and the total value of a distribution. Consider, for example, the high-performance text search engine library Lucene. The median of the heavily-skewed distribution for cyclomatic complexity [19] at class level increased from 5 to 8 as new classes were added to the system. The change in the median suggests that the overall cyclomatic complexity of Lucene increased significantly. But this interpretation is incorrect. The newly added classes had actually the opposite effect. What made the median increase was the growing population size (*i.e.*, the number of classes in the system), which resulted in a new middle value for cyclomatic complexity.

Interestingly, an approach to cope with and meaningfully interpret unevenly-distributed data sets has already been widely adopted in the field of economics. In 1912,

*In Proceedings of the 25th International Conference on Software Maintenance (ICSM 2009), pp. 179–188, IEEE Computer Society, Los Alamitos, CA, USA, 2009.

the Italian statistician Corrado Gini proposed the so-called *Gini coefficient*, a single numeric value between 0 and 1, to measure the *inequality* in the distribution of income or *wealth* in a given population (cf. [9, 24]). A low Gini coefficient indicates a relatively equal wealth distribution in a given population, with 0 denoting a perfectly equal wealth distribution (*i.e.*, everybody has the same wealth). A high Gini coefficient, on the other hand, signifies a very uneven distribution of wealth, with a value of 1 signaling perfect inequality in which *one* individual possesses all of the wealth in a given population. Today, the Gini coefficient is a widely used social and economic indicator to ascertain an individual's ability to meet financial obligations or to correlate and compare per-capita GDPs [28].

We can adopt this technique and consider software metrics data as income or *wealth distributions*. Each metric that we collect for a given property, say the number of methods defined by all classes in an object-oriented system, is summarized as a Gini coefficient, whose value informs us about the degree of concentration of functionality within a given system. Moreover, since the Gini coefficient is both *population size independent* and *bounded*, we obtain a tool for comparative analysis for any two software systems, an aspect of particular interest for the study of evolving systems.

To test and refine our technique, we conducted an exploratory study in which we applied the Gini coefficient to more than fifty object-oriented software systems developed with Java and C#. We observed the following:

- Gini coefficients across multiple software systems are all strongly bounded. The typical overall range of Gini coefficients (for multiple software metrics) is between 0.45 and 0.75, which suggests that developers favor solutions in which a few large and complex abstractions do the bulk of the work, whereas the rest of the system acts as service or data providers.

- Selected software metrics exhibited Gini coefficients greater than 0.85. Such a high value is a strong indicator for the presence of machine-generated code or code that is structured like machine-generated code. Inspection of the system being analyzed always confirmed this.

- Gini coefficients change little between adjacent releases, with the exception of occasional spikes that revealed significant changes like architectural shifts or the introduction of machine-generated code. Once developers commit to a specific solution approach, it seems they seldom tamper with it afterwards.

This last observation also supports the appropriateness of the laws "Conservation of Organizational Stability" and "Conservation of Familiarity" of software evolution as formulated by M. M. Lehman [14]. Consequently, we should be able to use this inherent property of change to help improve the risk management practices as any substantial variation in Gini coefficients can be used as a trigger for a more thorough investigation and retrospection.

The rest of the paper is organized as follows: in Section 2 we motivate our work with a brief overview of related work. We proceed by developing an economic metaphor and how it can assist us to overcome the statistical challenges for comparative software analysis. Section 3 presents the experimental method for the investigation of our technique. In Section 4 we discuss possible interpretations and consequences of our observations. We conclude in Section 5 with a summary of the expressive power of the Gini coefficent for comparative software analysis and some remarks about future work.

2. Towards an Economic Metaphor

Software metrics typically exhibit highly skewed distributions, which makes the use of usual analysis tools that assume Gaussian or other regular distributions inappropriate. In this section we review background and related work, and we motivate our proposal to adopt the wealth-based Gini coefficient to analyze software metrics.

2.1 Typical Metric Data Distributions

Real-world software systems typically contain hundreds of abstractions (*e.g.*, classes in object-oriented systems). The sheer volume of data available can make it difficult to understand the nature of these systems and how they have evolved [5]. A common approach [8] to reducing the complexity of the data is to apply some form of some simple summarization such as the *mean*, *median*, or *standard deviation*. Unfortunately, these simple statistics provide little useful information about the distribution of the data, particularly if it is skewed, as is common with many software metrics.

Additional statistics such as the *skew*, measuring the asymmetry of the data, and *kurtosis*, measuring the peakedness of the data, may be useful, but cannot easily be used to compare systems with different population sizes. Also, these measures are unbounded [22], making relative comparisons difficult. There is a need for a more general statistical measure that provides a way of comparing systems with quite different first order statistics, yet still manages to capture the nature of the software metrics.

Software systems tend to exhibit asymmetrically-shaped metrics data distribution profiles. Typically [29, 30] software systems follow a simple pattern: a few software entities contain much of the complexity and functionality,

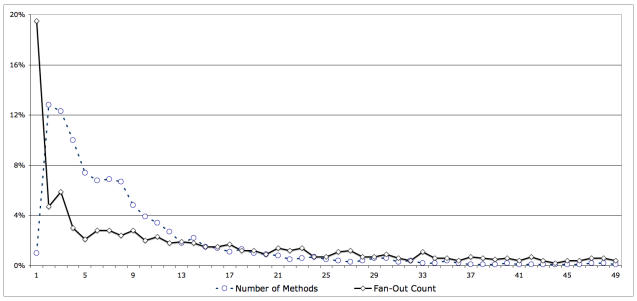

Figure 1. Positively skewed metrics data.

whereas the others define simple data abstractions and utilities. Most metrics are skewed, but skewed in different ways, making comparisons based on average and standard deviation largely meaningless. For example Figure 1 shows the distributions, in percent, for the metrics *Number of Methods* and *Fan-Out Count* for release 2.5.3 of the Spring framework (a popular Java/J2EE light-weight application container). In both cases the distributions, although significantly skewed, are quite different. We would like a simple statistic that provides a synthesis of the skew, kurtosis, mean, and variance statistics of the data.

Others have proposed fitting the metric data to simple skewed distributions such as the Lognormal, Exponential, or other power laws [1, 26]. Unfortunately, there is no widely-accepted distribution that captures consistently and reliably software metric data. Certainly the two metrics described in Figure 1 do not appear to map easily to well-known skewed distributions without additional effort. But more importantly, we are not required to fit a given software metric to particular distributions in order to interpret it. We are interested in simple measures that can be used to characterize a particular property.

Given this situation, it is not surprising that metrics use in industry is rare. Simple statistics, such as median and variance are likely to be misleading. Comparison of different distributions may provide some insight, but require skill to interpret, particularly given the huge number of metrics that might be used and the different population sizes that might be encountered. Consequently, we have investigated simple higher-order statistics [20] that capture key aspects of the data, that are bounded, and that effectively summarize the metrics. We argue the Gini coefficient has great potential to be a useful, simple measure to monitor the stability of software properties over time.

2.2 Lorenz Curve and Gini Coefficient

One of the key pieces of information we wish to obtain from software metrics is the allocation of functionality within the system. Understanding whether the system has a few classes that implement most of the functions or whether functions are widely distributed gives us an insight into how the system has been constructed, and how to maintain it [3]. Allocation of some attribute within a population has been studied comprehensively by economists who are interested in the distribution of wealth [31]. Key to this analysis is the *Lorenz curve* [17], an example of which is shown in Figure 2. A Lorenz curve plots on the y-axis the proportion of the distribution assumed by the bottom x% of the population. The Lorenz curve gives a measure of inequality within the population. A diagonal line represents perfect equality. A line that is zero for all values of $x < 1$ and 1 for $x = 1$ is a curve of perfect inequality.

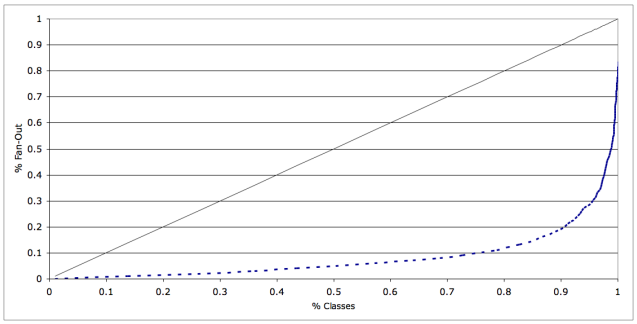

Figure 2. Lorenz curve for Fan-Out Count in Spring framework in release 2.5.3.

For a probability density function $f(x)$ and cumulative density function $F(x)$, the Lorenz curve $L(F(x))$ is defined as:

$$L(F(x)) \quad = \quad \frac{\int_{-\infty}^{x} t\, f(t)\, dt}{\int_{-\infty}^{\infty} t\, f(t)\, dt} \tag{1}$$

The Lorenz curve can be used to measure the distribution of functionality within a system. Figure 2 is a Lorenz curve for the *Fan-Out Count* metric in the Spring framework release 2.5.3. Although the Lorenz curve does capture the nature of distribution, it can be more effectively summarized by means of the Gini coefficient. The Gini coefficient is defined as a ratio of the areas on the Lorenz curve diagram. If the area between the line of perfect equality and Lorenz curve is A, and the area under the Lorenz curve is B, then the Gini coefficient is $A/(A + B)$ [31].

More formally, if the Lorenz curve is $L(Y)$, then

$$G \quad = \quad 1 - 2 \int_{0}^{1} L(Y)\, dY \tag{2}$$

The Gini coefficient is a *higher order statistic* as it is derived from the Lorenz curve, which itself is a summary measure computed over a cumulative probability distribution function. The Gini coefficient has a number of useful

Name	Rationale	Description
Load Instruction Count	Responsibility	Number of read instructions per class
Store Instruction Count	Responsibility	Number of write instructions per class
Weighted Method Count	Complexity	Degree of algorithmic branching
In-Degree Count	Popularity	Number of classes class X depends upon
Out-Degree Count	Delegation	Number of classes depending on class X
Number of Methods	Decomposition	Breadth of functional decomposition
Public Method Count	Interface Size	Exposure of responsibility
Number of Attributes	Information Storage	Density of information stored in class
Fan-Out Count	Delegation	Degree of reliance on others
Type Construction Count	Composition	Number of object instantiations

Table 1. Collected class-level measures for Gini analysis.

properties in that it is bounded between 0 and 1, makes no assumptions as to the distribution of the statistic under investigation, and can be compared between differently sized populations. These properties makes it an ideal statistic for comparing the distribution of metrics between software systems. Moreover, the Gini coefficient provides a simple and intuitive means for qualitative analysis of observed software properties. In the next sections we determine the Gini coefficients for a large selection of software metrics across a range of systems and provide a discussion on the interpretation of specific Gini values.

3. Wealth-based Software Analysis

How useful is the Gini coefficient in analyzing software systems? Does it enable useful comparisons to be made between systems? Can it provide us with any insights as to how developers construct systems? In this section we report on our study of freely available, open-source, object-oriented Java and C# systems.

3.1 The Setup

Our analysis method does not rely on the availability of source code. Both, Java and C# programs are translated to a platform-independent representation consisting of two components: virtual machine instructions, called *bytecode*, and embedded type information, called *metadata*. We can exploit the metadata information to extract software metrics. Together with bytecode inspection, this approach can reveal almost as much about a system as its original source code.

Using this knowledge, we have been developing *JSeat* [10], a software analysis and visualization framework for Java. JSeat consists of a set of tools that can be individually tailored to meet specific metrics extraction and exploration criteria. All metrics data retrieved with JSeat can be exported to a simple, text-based representation allowing for further statistical analysis with third-party products.

We used JSeat to produce the raw data for the longitudinal analysis [4, 11] of Java software systems. We ran the JSeat metrics extraction on 1,200 unique releases originating from 46 suitable Java candidate systems. To ensure that all studied systems are *non-trivial* we applied the following selection criteria to identify suitable candidates:

- A selected system must comprise no less than 100 classes throughout its lifetime.
- A selected system must have undergone active development for at least 18 months.
- At least 15 unique releases must be available for each selected system.

For each release we distilled ten size and complexity measures (see Table 1). We opted for *direct* metrics only (*i.e.*, metrics that measure just one attribute) as they can be considered wealth distributions immediately.

In order to assess assigned responsibilities we use the two metrics *Load Instruction Count* and *Store Instruction Count*. Both metrics provide a measure for the frequency of state changes in data containers within a system. *Weighted Method Count*, on the other hand, records all branch instructions and is used to measure the cyclomatic complexity [19] at class level. The remaining two dynamic measures are *Fan-Out Count* and *Type Construction Count*. The former offers a means to document the degree of delegation, whereas the latter can be used to count the frequency of object instantiations.

The remaining metrics provide *structural* size and complexity measures. *In-Degree Count* and *Out-Degree Count* reveal the coupling of classes within a system. These measures are extracted from the type dependency graph [29] that we construct for each analyzed system. The vertices in this graph are classes, whereas the edges are directed links between classes. We associate *popularity* (*i.e.*, the number of incoming links) with *In-Degree Count* and usage or *delegation* (*i.e.*, the number of outgoing links) with *Out-Degree Count*. *Number of Methods*, *Public Method Count*, and *Number of Attributes* define typical object-oriented size

measures and provide insights into the extent of data and functionality encapsulation.

But do these measures all represent independent characterizing properties? We need to examine, therefore, all selected metrics more closely and check whether there exists a *linear relationship* between any of them. If we discover a relationship linking two measures, we may be able to eliminate one metric when it does not provide additional insights. We computed the Pearson's correlation coefficients for all measures and systems (see Table 2 summarizing our findings for JasperReports 0.3.0, an embeddable Java reporting library) and observed the following:

- There exits a strong positive correlation (*i.e.*, > 0.8 [25]) between some different measures consistently across our entire data set.

- The strength of the relationship varies across systems and versions. For example, the measures *Load Instruction Count* and *Fan-Out Count* are strongly correlated in JasperReports 0.3.0, but this relationship is not as strong in other systems and versions.

- Across all systems, the measure *In-Degree Count* is only weakly correlated to other metrics if at all. This implies that the popularity of a class is not a function of its size or complexity.

- *Load Instruction Count* and *Store Instruction Count* are consistently strongly correlated. This signifies that data containers require a pairwise read and write.

We decided therefore that none of our selected measures, except *Load Instruction Count* and *Store Instruction Count*, qualify for potential exclusion from our analysis. Even if two metrics are correlated at one point, the strength of this relationship is not necessarily maintained at the same level while the system under investigation evolves. Moreover, we found repeatedly that the Gini coefficients for metrics that may be considered correlated, diverge independently of the underlying correlation.

Though *Load Instruction Count* and *Store Instruction Count* are strongly correlated, we need to analyze them separately also. We observed in some instances (*e.g.*, CheckStyle 3.0 to 3.1 *Load Instruction Count* Gini coefficient changes from 0.87 to 0.80 while *Store Instruction Count* remains stable) that the *Load Instruction Count* Gini coefficients changed independently of the *Store Instruction Count* Gini value. As a consequence, all selected measures have to be considered significant as they can reveal different, sometimes surprising, dimensions of the software system, which may be lost by eliminating certain metrics.

3.2 Gini Coefficients in Java Software

After extracting the desired metrics raw data with JSeat, we computed the Gini coefficients for our measures. Much to our surprise, the computed Gini coefficients for all measures were distributed over a very narrow range between 0.45 and 0.75 (see Figure 3). Moreover, individual measures showed a remarkable stability between versions with a typical delta of less than 0.01. This fluctuation decreased even further as systems were maturing.

There were, however, some noticeable exceptions in eight systems: Checkstyle, Hibernate (version 3.0b1 and higher), PMD, Groovy, ProGuard, FreeMarker, JabRef (version 1.4.0 and higher), and JasperReports. In these systems we observed persistent occurrences of Gini values for *Weighted Method Count* greater than 0.8, surpassing the value 0.91 in CheckStyle version 2.1.0. We discovered, upon further inspection, that all systems contained machine-generated code, which yields an extremely uneven functionality distribution.

The only systems that produced rather puzzling values were Xerces2 and Xalan. In both the Gini coefficient for *Weighted Method Count* is between 0.75 and 0.82. These high values result from hand-written parsers that produce functionality distribution profiles similar to machine-generated code. These were the only instances in which we observed such high values for *Weighted Method Count* without the presence of machine-generated code.

3.3 Gini Coefficients in C# and .NET

Does the programming language or the execution platform impact metric distribution profiles? In July 2000, C# [6], a new *managed* language for the .NET platform, was announced. Like in Java, C# programs are compiled into a machine-independent, language-appropriate representation defined by the *Common Language Infrastructure* [21]. Moreover, C# and Java are very closely related and we therefore asked ourselves whether programs written in C# exhibit distribution profiles similar to the ones we observed in Java. Unfortunately, the number of freely-available, open-source systems developed in C# framework that met our selection criteria is rather limited. So, we began our study with systems that were originally written in Java and had been ported to the .NET platform in order to take advantage from the knowledge gained in the analysis of their respective Java counterparts.

For the .NET metrics extraction, we used CLI [18], an assembly reader library that provides access to both the metadata and byte code. We added a small wrapper for the computation of the Gini coefficients and stored the resulting data in a text file for further processing with JSeat.

We collected metrics data from four .NET systems:

Load Instruction Count (LIC)	–									
Store Instruction Count (SIC)	**0.93**	–								
Weighted Method Count (WMC)	0.80	**0.81**	–							
In-Degree Count (IDC)	0.18	0.24	0.15	–						
Out-Degree Count (ODC)	**0.86**	**0.82**	0.77	0.08	–					
Number of Methods (NOM)	0.71	0.66	0.44	0.26	0.63	–				
Public Method Count (PMC)	0.21	0.20	0.09	0.27	0.18	0.75	–			
Number of Attributes (NOA)	0.70	0.79	0.48	0.23	0.50	0.53	0.16	–		
Fan-Out Count (FOC)	**0.96**	**0.87**	0.80	0.09	**0.85**	0.62	0.16	0.67	–	
Type Construction Count (TCC)	0.78	0.78	0.53	0.11	0.68	0.56	0.16	**0.82**	**0.84**	–
Metric	LIC	SIC	WMC	IDC	ODC	NOM	PMC	NOA	FOC	TCC

Table 2. Pearson's correlation coefficients – JasperReports 0.3.0.

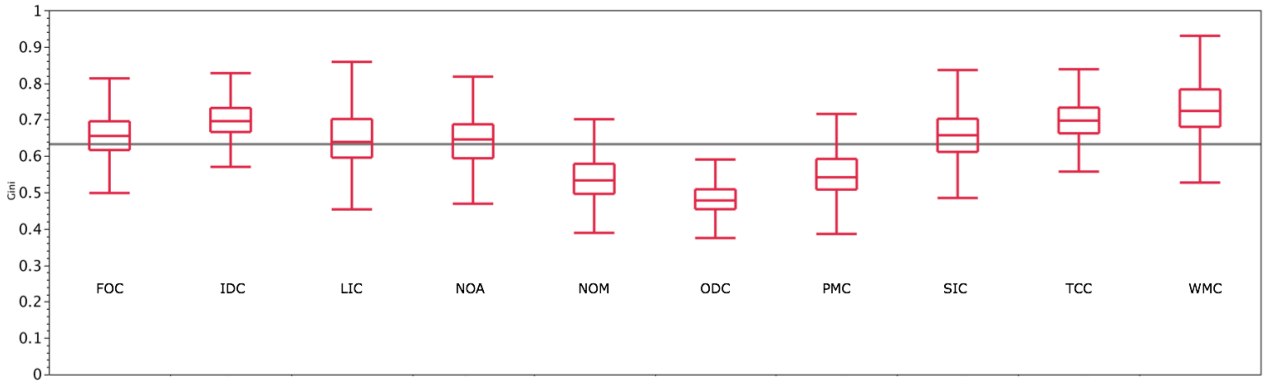

Figure 3. Box plot of Gini coefficients across all analyzed systems.

iTextSharp, NHibernate, SharpDevelop, and NAnt. The analysis of our 10 measures produced Gini coefficients equivalent to the ones determined for Java systems. However, there were also exceptions. We observed a shift exceeding 0.4 (*i.e.*, individual Gini coefficients doubled in value) for almost all measures in NAnt version 0.8.3-rc1. The Gini coefficients stayed high until version 0.84-rc1, where they assumed "normal" values again. An inspection of the developer logs provided an explanation: in version 0.8.3-rc1, the NAntContrib project was integrated into the NAnt distribution. This project defines a number of utilities whose metrics exhibit very uneven distribution profiles caused by a centralization of event handling in a few classes. In version 0.84-rc1, the developers removed NAntContrib from NAnt resulting in a change by -0.4, returning the Gini coefficients for NAnt to their previous values.

4. Discussion

4.1 The Value of the Gini Coefficient

We discovered in our analysis that Gini coefficients normally change little between adjacent releases. However, changes do happen and may result in significant fluctuations in Gini coefficients that warrant a deeper analysis (see Figure 4 showing selected Gini profiles for 51 consecutive releases of the Spring framework). But why do we see such a remarkable stability of Gini coefficients?

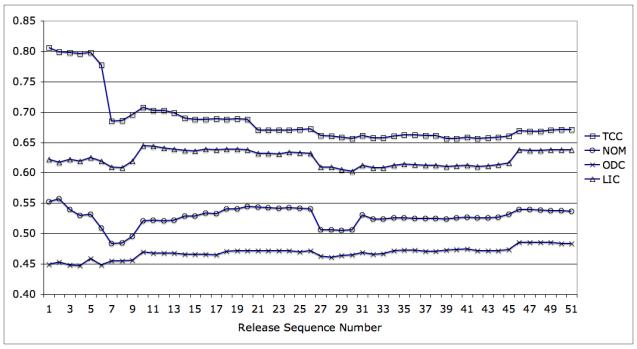

Figure 4. Selected Gini profiles in Spring.

Developers accumulate system competence over time. Proven techniques to solve a given problem prevail, where untested or weak practices have little chance of survival. If a team has historically built software in a certain way, then it will continue to prefer a certain approach over others. Moreover, we can expect that most problems in a given domain are similar, hence the means taken to tackle them

would be similar, too. Tversky and Kahneman coined the term "decision frame" [27] to refer to this principle in which *decision-makers* proactively organize their solutions within well-established and strong boundaries defined by cultural environment and personal preferences. These boundaries manifest themselves also in the software systems.

When developers are making decisions, they weigh the benefits of using a large number of simple abstractions against the risk of using only a few, but complex, ones in their solution design. Our findings indicate that developers favor the latter. In particular, we learn (see Figure 3) that the Gini coefficients of most metrics across all investigated systems assume bounded values that range just between 0.60 and 0.75. These values mark a significant inequality between the "richest" and the "poorest". For example, in the Spring version 2.5.3 approx. 10% of the classes possess 80% of the *Fan-Out Count* wealth (see Figure 2).

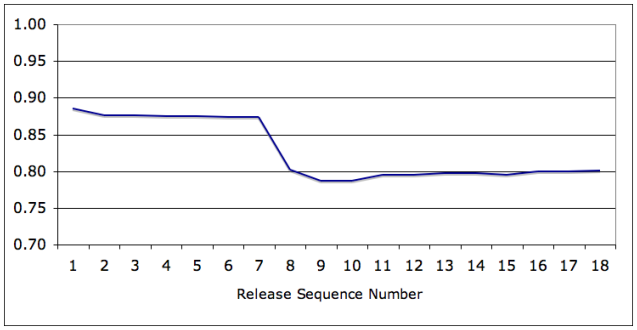

Figure 5. IDC Gini profile for Struts.

Another observation we made is that it is rare to see Gini coefficients greater than 0.8. For example, we noticed that in Struts, a Web application framework, the Gini coefficient for *In-Degree Count* initially moves beyond this threshold, only to fall back below it within a few releases (see Figure 5). This interesting behavior reveals that, in order for software systems to sustain evolution, responsibilities have to be distributed across abstractions in such a way that developers can maintain their cohesion. For each measure we can clearly identify *lower* and *upper bounds* that appear to define corresponding limits (cp. Table 3).

The existence of both lower and upper bounds for Gini coefficients in software systems can be viewed as a result of a *trade-off* developers use throughout development. Decisions how to allocate responsibilities to classes within a system are a product of the application domain, past experience, preferences, and cultural pressures within the development team. The Gini coefficient is a potential mechanism for characterizing these trade-offs and perhaps more importantly, identifying when they change. In the analyzes that follow we have, somewhat arbitrarily, chosen a difference of greater than 4% between adjacent releases as being significant. The motivation for this is the Pareto principle [2] (also known as the 80–20 rule). We found that Gini coefficients changed by more than 4% in less than 20% of the studied releases.

Metric	Minimum	Maximum
Load Instruction Count	0.60	0.64
Store Instruction Count	0.63	0.68
Weighted Method Count	0.68	0.72
In-Degree Count	0.62	0.72
Out-Degree Count	0.45	0.49
Number of Methods	0.48	0.56
Public Method Count	0.47	0.56
Number of Attributes	0.57	0.67
Fan-Out Count	0.62	0.66
Type Construction Count	0.66	0.81

Table 3. Gini value ranges in Spring.

To illustrate the effectiveness of this threshold, consider again Figure 4 showing selected Gini coefficients from the Spring framework. We see a jump of 0.092 in *Type Construction Count* from the 6th (*i.e.*, version 1.0m4) to the 7th (*i.e.*, version 1.0r1) release. Upon further inspection we discovered that this change was caused by the removal of just one, yet very rich, class — *ObjectArrayUtils*, which also affected the Gini value for *Number of Methods*. This class provided 283 utility methods to map primitive arguments to an object array. These methods contained a total of 1,122 object instantiations. The next biggest class in terms of type constructions defined just 99 object instantiations. This concentration of type constructions in *ObjectArrayUtils* caused the high values for *Type Construction Count* of approx. 0.8 up to the 6th release. A similarly rich class was never added to the Spring framework again.

We notice another major fluctuation of 0.0347 in *Number of Methods* between the 26th (*i.e.*, version 1.2.6) and the 31th (*i.e.*, version 2.0m5) releases. In the 27th release, the developers of the Spring framework decided to remove a set of large template classes from the core functionality. After a period of 6 months (*i.e.*, from version 2.0m5), however, these classes were brought back causing the Gini value for *Number of Methods* to return to its original state. Though an explanation as to why this was done has not been included in the release notes, our approach detected this anomaly.

4.2 The Sensibility of the Gini Coefficient

The stability of the Gini coefficient indicates that developers rarely make modifications to systems that result in a significant reallocation of functionality within the system. Moreover, the likelihood for such events to occur is greater in earlier versions. Managers can, therefore, use this knowl-

System	Measure	Release	Gini	Explanation
PMD	*Type Construction Count*	1.01	0.81	User interface code refactored into multiple smaller classes.
		1.02	0.73	
Checkstyle	*In-Degree Count*	2.4	0.44	Plug-in based architecture introduced.
		3.0b1	0.80	
Proguard	*Type Construction Count*	3.8	0.78	2,500 line obfuscation instruction mapping class introduced.
		4.01	0.90	
JabRef	*Weighted Method Count*	1.3	0.75	Machine generated parser introduced.
		1.4	0.91	
WebWork	*Fan-Out Count*	2.1.7	0.51	A large utility class and multiple instances of *copy-and-paste* code introduced.
		2.21	0.62	
Xerces2	*In-Degree Count*	2.0a	0.59	Abstract syntax tree node referencing changed.
		2.0b	0.78	
JasperReports	*Public Method Count*	1.0.3	0.58	Significant design approach change with introduction of a set of new base classes.
		1.1.0	0.69	

Table 4. Sample of observed significant changes to Gini coefficients in consecutive releases.

edge to define project-specific triggers both to help detect substantive shifts in the code base and to ask directed questions about the reasons for their occurrences. For example, in our study we were able to detect a major change in the *Type Construction Count* of Proguard, a Java byte code obfuscator, between release 3.8 and 4.0 (cf. Table 4). A detailed analysis disclosed that the developers had added a new, large single auxiliary class to centralize the obfuscation mapping of instructions — a change, so significant, as to warrant an appropriate notification to both internal and external team members.

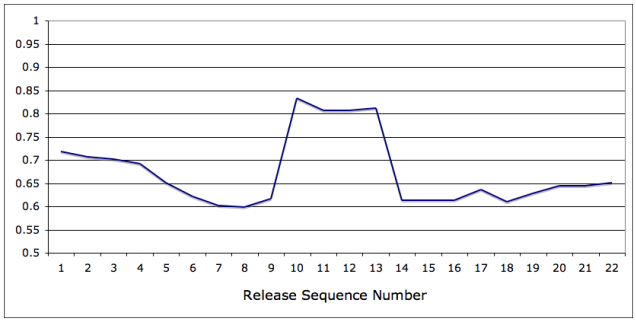

Figure 6. WMC Gini profile for NAnt.

Architects often face a situation in which they have to select a third-party component. They need to consider the desired functional requirements as well as project-specific and institutional principles governing local development practices and styles. In other words, we need to consider risks arising from mismatches between existing team habits and newly adapted third-party preferences. Therefore, before a final decision is made, architects should inspect past or ongoing projects and compare the responsibility distribution profiles (captured in the respective Gini coefficients). This method focuses on fitness rather than prescriptive rules to proactively avert emerging development risks. NAnt provides a vivid example of what can happen if incompatible profiles are mixed. The integration of NAntContrib into the code base of NAnt in version 0.8.3-rc1 (*i.e.*, release 10) caused a serious disruption in the Gini coefficient for *Weighted Method Count* as shown in Figure 6 as well as other measures.

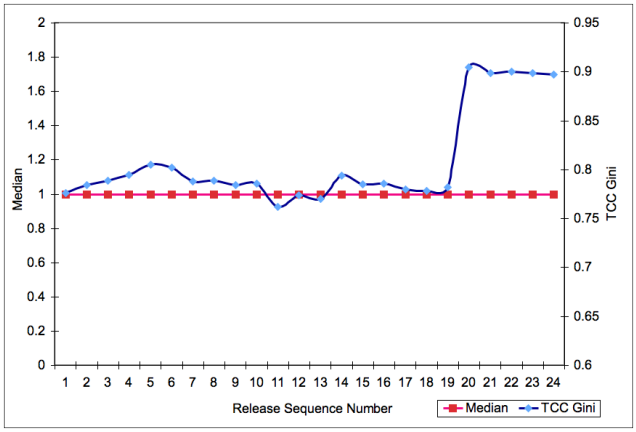

Figure 7. TCC for ProGuard.

The true benefit of the Gini coefficient is its ability to precisely capture and summarize changes in both the degree of concentration and the population size. When analyzing metrics data, crucial aspects to consider are the width of distribution and its underlying dispersion [17]. Standard average measures like median are blind for this dimension. A system that strikingly demonstrates the problem with standard averages is ProGuard (see Figure 7). The median for *Type Construction Count*, for example, stays firm at 1 for 340 weeks of development, suggesting a fairly routine evolution of the code base over a period of 6.5 years. The Gini coefficient for *Type Construction Count*, on the other hand,

moves from 0.776 to 0.897 indicating that the arrival of new classes results in a less equitable concentration of object instantiations. In case of ProGuard, the changes to the system occur at the upper end of the *Type Construction Count* measure. While the median for *Type Construction Count* remains the same, the changing Gini coefficient reflects correctly the occurrence of a dramatic architectural shift.

4.3 Detecting Machine-generated Code

In our study we noticed that certain systems consistently exhibited *Weighted Method Count* Gini values above 0.85. An investigation into those unusually high values revealed the presence of *machine-generated code*, specifically parsers and expression processors. It turns out that machine-generated code structures have unique distribution profiles, whose Gini coefficients move very close to 1 (see Figure 8 showing the *Weighted Method Count* Gini profile for JabRef). Code generators often employ *greedy* strategies to map a given specification to target code. Therefore, generated code exhibits a much higher structural and algorithmic density. As noted earlier, human developers rarely write code in which Gini coefficients for specific measures go past 0.80. But if they do, their code is very similar that produced by a corresponding code generator.

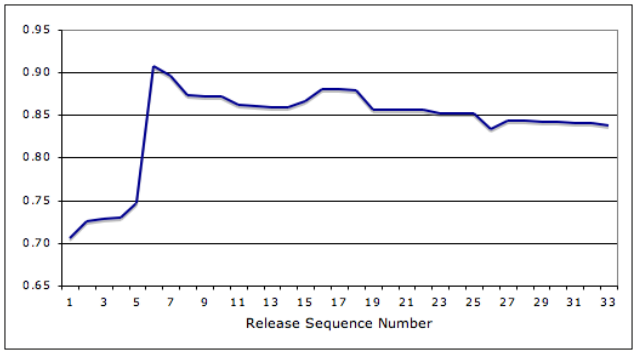

Figure 8. WMC Gini profile for JabRef.

In a large J2EE commercial system that we investigated as a part of our experiments we encountered a number of different measures, whose Gini coefficients were well above 0.95. Upon closer investigation, we found that this phenomenon was caused by a substantial number of machine-generated *stub-classes* that did not contain any functionality.[1] In essence, this large number of "poor" classes made those that actually possessed functionality look unusually rich. As a simple solution to compensate for this distortion, we eliminated all 0-valued data points from the underlying raw measures, a technique, that made the Gini measures return to their typical ranges.

[1] No open-source Java system showed similar behavior.

Knowing that Gini coefficients have strong boundaries can be used as an effective and qualitative detection method for identifying machine-generated code. Our method of detecting machine-generated code has already been tested in commercial code audits aimed at assessing viable strategies for a repositioning of an existing product. The presence of machine-generated code signals the possible need for additional expertise in order to maintain or enhance an existing code base and to meet strategic objectives.

5. Conclusion

Software metrics are known to exhibit erratic and skewed distributions. As a consequence it can be hard to draw sensible conclusions when comparing metrics of different projects, or even of a single project as it evolves over time. We have proposed to use a higher-order statistic, known as the Gini coefficient, to exploit this commonly observed phenomenon, and to study software metrics the same way that economists study the distribution of wealth. We have studied a large number of open-source software systems and compared their metric values using the Gini coefficient.

Much to our surprise, developers prefer to organize their solutions around a fairly limited set of design options as indicated by our analysis. The Gini coefficient for all metrics hover over a very tightly-bounded value space. This fact gives rise to the speculation that developer decisions are driven by some form of *cognitive preference* that makes developers choose solutions with certain typical profiles. As a consequence, the perceived practical design options occur in a narrow and, hence, predictable design space.

What are the reasons for this observed phenomenon? We do not have a direct answer yet, but it appears that there exists a *comfort range* for programmers similar to the ideal temperature for an organism. This comfort range is domain-independent (the Gini coefficients across all analyzed systems fall within a tightly-bounded range) and language-neutral (Java or C# do not, in any way, limit developers to choose between design alternatives or forces them to construct software with typical distribution profiles).

Another interesting facet surfaced in all studied systems are *God*-like classes that shoulder most of the work. Developers are not afraid to construct, maintain, and evolve very complex abstractions. Contrary to common belief developers can actually manage complexity in real projects, but at what price? Classical software engineering shies away from tighly arranged and centralized abstractions. However, we find that it is well within the developers's mental capacity to organize and control the structure of software. Furthermore, our analysis suggests that there has to exist a certain degree of inequality, which defines the *equilibrium* that drives sustainable software evolution.

Acknowledgments

The authors would like to acknowledge the contributions of Asiri Wanigarathne and Ben Hall. Asiri developed the .NET metrics extraction tool and collected data from .NET software systems. Ben collected data from a commercial software system and interviewed the architects which validated the usefulness of the Gini coefficient in a commercial project. Special thanks go to Tudor Girba for his feedback on earlier drafts of this paper.

Oscar Nierstrasz gratefully acknowledges the financial support of the Swiss National Science Foundation for the project "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 - Sept. 2010).

References

[1] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the shape of java software. *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming Languages, Systems, and Applications*, pages 397–412, 2006.

[2] A. Bookstein. Informetric distributions, part I: Unified overview. *Journal of the American Society for Information Science*, 41(5):368–375, 1990.

[3] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Square Bracket Associates, 2008.

[4] P. Diggle, P. Heagerty, K. Liang, and S. Zeger. *Analysis of longitudinal data*. Oxford University Press, 2002.

[5] K. A. Ericcson, W. G. Chase, and S. Faloon. Acquisition of a memory skill. *Science*, 208(4448):1181–1182, June 1980.

[6] European Computer Machinery Association. *Standard ECMA-334: C# Language Specification*, third edition, June 2005.

[7] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999.

[8] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. Thomson Publishing, second edition, 1996.

[9] C. Gini. Measurement of Inequality of Incomes. *The Economic Journal*, 31(121):124–126, Mar. 1921.

[10] *JSeat*. http://code.google.com/p/jseat, Sept. 2008.

[11] C. Kemerer and S. Slaughter. An empirical approach to studying software evolution. *Software Engineering, IEEE Transactions on*, 25(4):493–509, Jul/Aug 1999.

[12] B. A. Kitchenham. An evaluation of software structure metrics. In *Proceedings of the 12th International Computer Software and Application Conference (COMPSAC 1988)*, pages 369–376. IEEE Computer Society Press, 1988.

[13] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

[14] M. Lehman. Laws of Software Evolution Revisited. In *European Workshop on Software Process Technology*, pages 108–124, 1996.

[15] M. M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 68(9):1060–1076, Sept. 1980.

[16] M. M. Lehman, D. E. Perry, J. C. F. Ramil, W. M. Turski, and P. Wernik. Metrics and Laws of Software Evolution – The Nineties View. In *Proceedings of the Fourth International Symposium on Software Metrics (Metrics '97)*, pages 20–32, Albuquerque, New Mexico, Nov. 1997.

[17] M. O. Lorenz. Methods of Measuring the Concentration of Wealth. *Publications of the American Statistical Association*, 9(70):209–219, June 1905.

[18] M. Lumpe. Using Metadata Transformations to Integrate Class Extensions in an Existing Class Hierarchy. In N. Kobayashi, editor, *Proceedings of the Fourth ASIAN Symposium on Programming Languages and Systems (APLAS 2006)*, LNCS 4279, pages 290–306, Sydney, Australia, Nov. 2006. Springer.

[19] T. J. McCabe. A Complexity Measure. *IEEE Transaction on Software Engineering*, 2(4):308–320, Dec. 1976.

[20] J. Mendel. Tutorial on higher-order statistics (spectra) in signal processing and system theory: theoretical results and some applications. *Proceedings of the IEEE*, 79(3):278–305, 1991.

[21] J. S. Miller and S. Ragsdale. *The Common Language Infrastructure Annotated Standard*. Microsoft .NET Development Series. Addison-Wesley, 2003.

[22] H. C. Picard. A Note on the Maximum Value of Kurtosis. *Annals of Mathematical Statistics*, 22(3):480–482, Sept. 1951.

[23] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Sixth edition, 2005.

[24] G. Pyatt. On the Interpretation and Disaggregation of Gini Coefficients. *The Economic Journal*, 86(342):243–255, June 1976.

[25] G. Succi, W. Pedrycz, S. Djokic, P. Zuliani, and B. Russo. An Empirical Exploration of the Distributions of the Chidamber and Kemerer Object-Oriented Metrics Suite. *Empirical Software Engineering*, 10(1):81–104, 2005.

[26] T. Tamai and T. Nakatani. Analysis of Software Evolution Processes Using Statistical Distribution Models. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 120–123. ACM Press New York, NY, USA, 2002.

[27] A. Tversky and D. Kahneman. The Framing of Decisions and the Psychology of Choice. *Science*, 211(4481):453–458, Jan. 1981.

[28] United Nations Devlopment Programme. Human Development Report 2007/2008. available at http://hdr.undp.org, 2007.

[29] R. Vasa, M. Lumpe, and J.-G. Schneider. Patterns of Component Evolution. In M. Lumpe and W. Vanderperren, editors, *Proceedings of the 6th International Symposium on Software Composition (SC 2007)*, LNCS 4829, pages 235–251, Braga, Portugal, Mar. 2007. Springer.

[30] R. Vasa, J.-G. Schneider, and O. Nierstrasz. The Inevitable Stability of Software Change. In *Proceedings of 23rd IEEE International Conference on Software Maintenance (ICSM '07)*, Paris, France, Oct. 2007. IEEE Computer Society.

[31] K. Xu. How Has the Literature on Gini's Index Evolved in the Past 80 Years? Department of Economics, Dalhouse University, Halifax, Nova Scotia, Dec. 2004.