Automated Inequality Analysis of Evolving Software Systems

Olga Goloshchapova

Submitted for a Degree of Master of Science(IT) (by Research)

2013
Abstract

Software measurement is a promising technique for evaluating the efficiency of a software development process and the quality of a software development product. The key component of software measurement is an application of software metrics (static code metrics) to retrieve information regarding specific characteristics of a development process and a product in order to understand their nature. This approach is especially beneficial when studying evolutionary changes in a software system, which allows for making predictions about its future states.

However, software metric data alone is often non-descriptive. It needs to be summarised to be interpreted. A common solution for this problem is the usage of standard summary statistics, such as mean or standard deviation. But, a software system is an outcome of a logical rather than a random process. Hence, software metrics data does not follow normal (Gaussian) distribution, which implies that central tendency statistics are unable to capture such information effectively.

In this thesis, socio-economic inequality indexes are offered as a viable alternative to existent aggregation techniques when being used to understand the nature of software. In particular, inequality indexes, like the Gini coefficient and the Theil index, have the advantage of showing how metrics distributions change as a software system evolves.

Both inequality indexes are, initially, applied in a controlled environment governed by a well-understood distribution principle to understand the rules with which they capture inequality. Then, these indexes are employed to evaluate evolving software systems in order to determine their usefulness as a metrics aggregation technique.

As a result, this work shows that two indexes capture inequality in a software system with separate level of granularity: the Gini coefficient offers a macro-level (architectural) analysis, whereas the Theil index is more sensitive to structural (micro-level) changes in a software system. Hence, when used in combination, the Theil index and the Gini coefficient offer a greater amount of information about system’s evolution than when employed separately.
Acknowledgment

I would like to acknowledge with particular gratitude the assistance of my supervisors, Dr. Markus Lumpe and Dr. Clinton Woodward. I am also indebted to a number of other people, in particular, Shannon Pace, Dr. Rajesh Vasa, and Dr. Samiran Muhmud who collaborated with me on various research topics and provided much needed support. Thanks also to the various developers of open source software systems for releasing their software with non-restrictive licensing. I am grateful to my educational provider, Swinburne University of Technology, for offering the resources and support to pursue a research higher degree.

Finally, I would like to thank my family and Thomas for their loving forbearance during the period of time required to conduct my research and write up this thesis.

Olga Goloshchapova, 2013
Declaration

I declare that this thesis contains no material that has been accepted for the award of any other degree or diploma and to the best of my knowledge contains no material previously published or written by another person except where due reference is made in the text of this thesis.

Olga Goloshchapova, 2013
# Contents

1 **Introduction**  
1.1 Background ................................. 1  
1.2 Research Questions .......................... 3  
1.3 Contributions ............................... 4  
1.4 Thesis Structure ............................. 4  

2 **The Analysis of Software Metrics Data**  
2.1 Motivation ................................. 8  
2.2 Software Evolution ........................... 11  
2.3 Metrics and Data ............................ 14  
2.4 Summary ................................ 16  

3 **Software Metrics Extraction and Analysis Tools**  
3.1 jCT - Java Code Tomograph .................. 18  
3.1.1 The Need for jCT .......................... 18  
3.1.2 jCT Organisation .......................... 20  
3.1.3 Metrics Definition Example ............... 23  
3.2 Auxiliary Tools .............................. 25
CONTENTS

3.3 Summary ................................................. 27

4 Inequality-based Software Analysis .............................................. 29
  4.1 Inequality Indexes ........................................... 30
    4.1.1 Lorenz Curve ........................................... 31
    4.1.2 Gini Coefficient ....................................... 32
    4.1.3 Theil Index ............................................. 33
  4.2 The Causes of Inequality ........................................... 37
  4.3 Economics Metaphor for Software Engineering ......................... 38
  4.4 Summary ................................................. 41

5 The Behaviour of the Gini Coefficient and the Theil Index ............... 43
  5.1 The Pigou-Dalton Experiment ................................... 44
  5.2 Gini, zGini, and GiniDif .................................... 48
  5.3 Summary ................................................. 51

6 Analysis of Inequality Indexes on Helix Repository ....................... 53
  6.1 Typical Ranges ........................................... 54
  6.2 Correlation Analysis ........................................ 56
  6.3 Indexes Behaviour ......................................... 58
  6.4 Inequality Indexes Application ................................ 68
  6.5 Summary ................................................. 76

7 The Economics Metaphor in Software Engineering .......................... 77
  7.1 Zero Elimination in Software Inequality Analysis ...................... 77
7.2 Summary ....................................................... 84

8 Conclusion ...................................................... 87
  8.1 Summary ................................................... 87
  8.2 Observations ............................................. 90
  8.3 Future Work ............................................. 91

A jCT Output File Examples .................................. 101

B The Theil Index and the Gini Coefficient Behaviour Analysis 105
List of Figures

3.1 *hsqldb-2.0.0* Type Dependency Graph .......................... 22

4.1 Lorenz curve example ............................................. 31

4.2 Gini coefficient with respect to equality line and the Lorenz curve .... 32

5.1 Mean-invariant transfers in a distribution with 100, 1000, and 10000 elements - the Gini coefficient vs. the Theil index ......................... 45

5.2 Mean-invariant transfers in a distribution with 100, 1000, and 10000 elements - the Theil index vs. the normalised Theil index ......................... 46

5.3 Scatter plot of the Gini coefficient vs. the normalised Theil index for 1,000-member experiment ......................................................... 47

5.4 Scatter plot of the Gini coefficient vs. the zGini coefficient for 1000-member experiment ......................................................... 49

5.5 The Gini Difference scatter plot for 1000-member experiment ............ 50

6.1 Inequality measures ranges on Helix repository for *setters* and *rfc* metrics .............................. 55

6.2 The zGini Coefficient and the normalised Theil index scatter plot for *freemaker* system on *cbo* metric ................................. 57

6.3 The zGini Coefficient and the normalised Theil index over RSN for *pmd* system’s *outdegree* metric ........................................... 57

6.4 “Step down” behaviour example (*kolmafia* evolution changes for *getters* metric) ........................................... 61
# List of Figures

6.5 "Down" behaviour example (*checkstyle* evolution changes for *nom* metric) 61

6.6 "Divergent" relationship example (*kolmafia* evolution changes for *nom* metric) 63

6.7 Predominate behaviour types of the *zGini* coefficient and the normalised *Theil* index 64

6.8 "Parallel" relationship samples 70

6.9 "Divergent" relationship examples 72

6.10 "Crossing" relationship examples 73

6.11 The sample distribution to present the normalised *Theil* index influences 75

7.1 The *Gini Difference* (*GiniDif*) rages across considered metrics 78

7.2 The *zGini* coefficient and the *Gini coefficient* relationship samples 84

A.1 *cpp* Metric Output File for *hsqldb-2.0.0*. 101

A.2 *isummary* Metric Output File for *org.hsqldb.jdbc.JDBCNClob* class in *hsqldb-2.0.0*. 102

A.3 *gssummary* Measure Output File *hsqldb-2.0.0* (partial). 103

A.4 *nom* Measure Output File for *hsqldb-2.0.0* (partial). 104

B.1 The *outdegree* histogram for every release of *ant* system 105

B.2 The *indegree* histogram for every release of *findbugs* system 106

B.3 The *cbo* histograms for every release of *activebpel* system 106

B.4 The *rfc* histogram for every release of *jchempaint* system 107

B.5 The *cbo* histogram for every release of *tapestry* system 108

B.6 The *outdegree* histogram for every release of *pmd* system 109
List of Tables

2.1 Lehman’s Laws of Software Evolution as defined in [44] .......................... 13
2.2 Selected static code measures ................................................................. 15
3.1 "Card" view of a compiled Java Class File structure as specified in [47] 21
4.1 Sample partitioned distribution A ............................................................... 35
4.2 Inequality in the distribution A per group .................................................. 36
4.3 Recommended mapping between factors affecting parametric variations in economics to software engineering equivalents ................................. 38
4.4 Rules of contribution to the Theil index value [21] ....................................... 40
6.1 Values ranges of the Gini coefficient, the zGini coefficient, the Theil index, and the normalised Theil index .......................................................... 54
6.2 Symbolic representation of overall index behaviour patterns (all column in Table 6.4) ................................................................. 58
6.3 Symbolic representation of "between indexes” behaviour patterns (b/w column in Table 6.4) ................................................................. 59
6.4 The Gini coefficient and the normalised Theil indexbehavioural trends 65
6.5 Table 6.4 summary ................................................................................. 67
6.6 Number of classes per release for pmd system ............................................ 74
LIST OF TABLES

7.1 The Pearson correlation coefficient values between GiniDif and RSN per metric ........................................... 80

7.2 Symbolic representation of behaviour patterns between the Gini coefficient and the zGini coefficient ........................................... 81

7.3 The Gini coefficient and the normalised Theil index behavioural trends ........................................... 81

7.4 Table 7.3 summary ........................................... 83
Chapter 1

Introduction

1.1 Background

Many practical problems can effect software development; they include exceeding production and operation costs, delayed product deliveries, or poor system performance once released [17]. In light of an increasing reliance of societies on software, these issues are required to be reduced or, ideally, eliminated [10, 17]. For this reason, software quality is an important topic in software engineering research.

Software quality directly influences various aspects of the software development process. It implies an increased product satisfaction level [33], improved methods for the software delivery process [34], and reduced associated maintenance costs [13]. To achieve software quality, one needs to continuously evaluate and advance the associated process and product during development time [34]. In other words, one requires to maintain a constant control over the production process.

However, as DeMarco [31] has noted, “we can not control something we can not measure”. To address this problem, a number of software quality evaluation methods have been developed [10, 13] that aim at fostering software measurement as a core unit of modern software development [33, 34].

Software measurement involves an application of software metrics – methods whose aim is to evaluate different program attributes [10,33]. These metrics are often applied repeatedly to various systems in order to relate gathered information with associated qualitative
properties (empirical studies) [80]. Thus, metrics and empirical results are means to examine certain software phenomena using quantitative data [34].

Software measures can capture a variety of system attributes; these includes:

- system size (i.e., Lines of Code or LOC [33, 98]),
- complexity (i.e., cyclomatic complexity [56, 89]),
- quality (i.e., FunctionPoints [33]), or
- design (i.e., Chidamber and Kemerer object-oriented metrics suite or CK [19])

However, metrics data alone is often non-descriptive and insufficient for solid conclusions [42, 54, 85, 86] and, hence, requires appropriate interpretation [54, 85, 86].

Many different approaches exist for metrics data analysis. Some techniques rely on graphical representation such as box plots [70], histogram plots [90], cumulative plots [79], and so on. Even though such methods offer an intuitive representation of analysed characteristics, they are not amenable to analytical reasoning [77]. Thus, additional techniques for summarising data are needed, which often accompany graphical representations. A simple example is the employment of standard summary statistics, such as mean or standard deviation. While useful in some cases, they are not suitable for studying data which does not follow normal (bell-curved) distributions [61]. In such cases, the interquartile range can be applied as a suitable alternative [58], albeit it is insensitive to the exact values of the scores in a distribution [61].

Other metrics aggregation techniques can be more involved and complicated. For example, Vasa et al. [90] employed the Bhattacharya measure, a statistical distance measure which evaluates the relative closeness between proportional information of two data samples. However, it does not explain or give an indication about the inner structure of considered data samples themselves.

In other work by Vasa et al. [85, 86], the Gini coefficient was utilised – a socio-economic measure that captures inequality in distribution of values in a population [36]. Due to its properties and nature, this index is especially beneficial when applied on highly skewed distributions, such as software metrics data [86]. The Gini coefficient, however, is not the only economics measure that can be used for metrics data interpretation; the Theil index is another inequality measurement technique that was introduced for software metrics data evaluation by Serebrenik et al. [70, 71, 91].


1.2 Research Questions

In this thesis, the usage of socio-economic inequality measures for metrics data aggregation is studied. In particular, the Gini coefficient and the Theil index are considered along with existent inconsistencies with their application for software analysis. In particular, this thesis explores the Theil index inability to process zero instances sensitively and the corresponding impact the Theil index’s effectiveness to summarise software metrics data. The objectives of this work are to clarify the claim of interchangeableness of these two indexes [92] and their suitability for software metrics analysis [71, 85, 86].

To understand the behaviour of both indexes, their respective performance is evaluated in a highly controlled environment. The goal is to define how each measure captures inequality and responds to asset reallocation in distributions. Also an experimental study is performed to analyse the effects zero values in a distribution have on inequality analysis.

These experiments, however, are a learning mechanism rather than a tool offering conclusive findings. To define how both indexes perform on real-life data we applied the Gini coefficient and the Theil index as metrics aggregation techniques in empirical software analysis of evolving software systems. The same approach was taken to define the value of involving zero instances into inequality studies.

The results of the analysis show that both indexes capture inequality differently. Specifically, the Gini coefficient offers us an ability to reason about inequality at a population level, whereas the Theil index captures inequality at an individual level. Also, it is shown that these two indexes should not be seen as a substitute of one another. Although, in the majority of cases they do exhibit similar behaviour, in others they yield very different pictures. In particular, the Gini coefficient evaluates architectural properties of a software system, while the Theil index provides a notion of its modular or allocation changes. Interestingly, such information reveals itself only when both indexes are used together. Thus, a result of this work is to recommend the employment of both measures for software studies in order to retrieve a greater and more detailed amount of information.

It is also determined that it is not a good practice to completely ignore zero instances when performing software inequality analysis. Zeros in metrics data need to be considered since they have a great influence on the degree of inequality in a software system.
1.3 Contributions

The key findings presented in this thesis are as follows:

- The Gini coefficient is a population-level aggregation function;
- The Theil index is an individual-level aggregation function;
- The Theil index can not be considered as a substitute to the Gini coefficient since they capture different information;
- The Theil index fails to account for zero values while performing inequality analysis of functionality dispersion;
- The Theil index can be used as a summary statistic for software metrics data; however, it is inadvisable to use it as a sole summary measure due shortcomings listed;
- Including zeros into inequality analysis is necessary since, unless these values are accounted for, the inequality of functionality dispersion is underestimated;
- A naive application of inequality measures for software analysis can lead to incorrect or unreliable results.

1.4 Thesis Structure

Chapter 2 presents the state-of-art and the motivation for this research. It focuses on the application of inequality indexes for software analysis and how such approach can benefit researchers and developers. This chapter also describes the selected software repository and metrics collection.

Chapter 3 describes the tools developed or used to support the research. It focuses on the utilised metrics extraction engine, Java Code Tomograph (jCT), and presents an example of metrics definition in this engine. The supporting information for this chapter is listed in Appendix A. This chapter also lists additional tools developed or used to assist post-processing activities required for software analysis.

Chapter 4 presents a detailed examination of each selected inequality measure along with a study of their properties and origins. In addition, it introduces a new inequality measure especially developed to assess the impact of zero values when using inequality-based software analysis, called the Gini Difference.

Chapter 5 presents a number of experiments which were ran to study the performance of
the Gini coefficient and the Theil index, along with the performance of the Gini Difference. This chapter encloses each experiment definition and corresponding results; however, it does not discuss or imply how do these inequality measures perform when applied to real-world data.

Chapter 6 analyses the Gini coefficient and the normalised Theil index when applied to the Helix repository. Initially, it describes typical values of these two indexes. It also studies whether the Gini coefficient and the normalised Theil index are correlated and evaluates the relationships between these two indexes. Finally, it provides an overview of how these indexes score as metrics data summary statistics and epitomises the benefits and drawbacks of each index.

Chapter 7 studies the Gini Difference (GiniDif) performance when applied to real-world software systems. In particular, it lists the typical GiniDif ranges and studies the growth patterns of this measure by evaluating the correlation between GiniDif and release sequences. This chapter also examines the relationships between the Gini coefficient and the zGini coefficient in order to determine the level of possible data distortion when applying the economics metaphor to software analysis.

Finally, Chapter 8 summarises the key observations of this work and presents possible directions for future work in this area.
Chapter 2

The Analysis of Software Metrics Data

The intention of this work is to understand the nature of software development to improve its quality and performance at acceptable costs. One of the ways to achieve this is via software measurement. This approach, however, requires meaningful metrics data interpretation, which can be challenging. Thus, to address this issue inequality measures are selected that offer a viable alternative to other statistical operations, such as central tendency statistics, when applied for software analysis. In particular, they have the advantage of showing how metrics distributions change as software systems evolve. Such an approach is especially beneficial for this study, since this work focuses on empirical analysis of evolving software systems.

In this chapter, Section 2.1 presents the motivation for a selection of inequality measures as the means to interpret metrics data. It also briefly discusses some of the inequality indexes that are used in this work as well as possible limitations of their application. Section 2.2 provides an overview of software evolution processes and their relation to this study. It lists the laws that govern software development growth, and suggests possible limitations to comparative evolutionary analysis, along with a possible solution to this issue. Section 2.3 elaborates the notion of measurement and validation in software engineering. Moreover, it lists the criteria used for the data and metrics selection in this study. Finally, Section 2.4 summarises this chapter.
CHAPTER 2. THE ANALYSIS OF SOFTWARE METRICS DATA

2.1 Motivation

Software development is a very labour intensive activity which often coincides with tight time and resources constraints [33]. Such pressures noticeably influence the quality of an outcome along with associated expenses [33]. Hence, there is a need for a technique that, at acceptable costs, may indicate as to where, in software systems, issues or inconsistencies occur and, thus, can assist developers in their decision making process. Therefore, an understanding of the underlying properties of software development is the key driver to acquire such a technique.

This is the domain of software metrics [33], which offer a way to evaluate a software system by comparing the actual product with desired quality outcomes [80]. However, metrics-based software analysis is not without issues, as software systems are often highly complex and their underlying properties are rarely completely understood. To advance the knowledge in this field a technique is needed that entails within it a means to manage increasing software complexity as well as a robust and powerful analysis technique that assists in making sound data-driven decisions [80].

Software measurement is an act, or a process, of assigning numbers to phenomena according to clearly defined criteria [74]. Fenton et al. [33] state that by manipulating, rearranging, comparing, and analysing measurements (and not the system information itself) we are able to make judgements regarding system properties, predict the future state of a system, detect inconsistencies, or recommend possible corrections to increase product quality. Typically, more information can be obtained by combining different types of measurement.

Software measurement can be direct or indirect [33, 42]. Direct measurement focuses on capturing just one property of a software system [33, 42]. Indirect measurement, on the other hand, acquires information about multiple attributes which often requires some level of data aggregation [33, 42]. In this work those two measurement types are mixed, which relate to size or complexity of a software system, or both. Thus, the goal of this work is to develop a model which reflects the reality and allows for building theories to reason about software systems and their corresponding development processes.

Nevertheless, it is not immediately clear as to how to interpret metrics values [54, 86]. Software metrics are extracted to understand the underlying properties of software systems which, consequently, allows us to produce techniques aimed to improve produc-
activity and quality of software development. As a consequence, we require to construe collected data to achieve meaningful interpretation and inference [86]. To achieve that, standard summary statistics, such as median, standard deviation, or any other central tendency techniques, can be used [61]. Still, raw metrics data is heavily skewed towards y axis [86]. Thus, the simplest solution, in this case, fails since central tendency measures produce unsatisfactory results when applied on non-Gaussian distributions [46,54,77,86]. There is no "probable error of a mean" in software [75]. Software development is not random [54]. In fact, it is a planned process that is governed by a frame of reference [84] to actuate the state of the current version of a system along with its future versions [85,86]. In addition, many researchers have recently discovered the presence of power laws in software [49,55,64,97,98] – a phenomenon that occurs when distributions of software attributes (e.g., object dependencies, token frequencies, and artefact size) exhibit long fat tails [49] and there exists a log-linear relationship between the scale and frequency of the measured attributes [20,59]. Thus, there is a need for a metrics interpretation approach that does not assume normal (Gaussian) distribution of data.

A powerful alternative to central tendency statistics is inequality analysis [86]. For over a century economists have been examining inequality in the distribution of socio-economic quantities, such as income [7,27,36,48,68,69,81]. Max Otto Lorenz was one of the first to realise the necessity of knowing whether the current distribution in a population becomes more or less unequal and the importance of comparing distributions over different epochs and countries [48]. Instead of using central tendencies statistics or percentile analysis, which he considered fallacious [48], he developed his own methodology for capturing inequality in distributions, known as the Lorenz curve [48].

The Lorenz curve is a graphical representation of inequality in a distribution [48]. Essentially, it is a plotted graph between a cumulative percentage of a population, ordered from poorest to richest, and a cumulative percentage of allocated resources [48]. Since it captures inequality information graphically, this approach offers an accurate and intuitive representation of a distribution’s nature [54,86]. However, it is often more effective to summarise data numerically [86]. One example is the Gini coefficient which is a ratio of the area between the line of the perfect equality (45 degree line) and the Lorenz curve [36,68]. Thus, this coefficient is a socio-economic entropic index that reflects the level of inequality numerically [86].

In fact, the Gini coefficient has been successfully applied for software analysis of features such as the information hiding principle via the distribution of getter and setter meth-
ods [54], class popularity via the study of in- and outdegree measures distribution [85,86], or component interactions by examining incoming and outgoing transitions in an automaton [51, 55]. The Gini coefficient has also been extensively used for empirical software analysis to identify system development trends on software evolution data [85].

The Gini coefficient, however, is not the only existent inequality index; there are a number of alternatives available [24]. In this thesis, though, only one more inequality measure is going to be considered – the Theil index. This index is derived from the notion of entropy in Shannon’s information theory [68, 72]. Its underlying principle states that “the more unlikely the event, the more interesting it is to know what has really happened” [68]. In other words, the Theil index captures the “surprise effect” in a distribution.

Even though both the Gini coefficient and the Theil index were developed with similar underlying goals, there are differences in their understanding and interpretation. For example, the Gini coefficient is a size and scale insensitive measure. Also, it ranks inequality at the highest level of granularity - population. Thus, we refer to this index as a macro level measure since it provides a large-scale distribution analysis. The Gini coefficient is not decomposable; meaning it lacks the ability to split data into consistent subgroups. Decomposability allows for inequality of a population to be presented as a sum of inequalities between groups, formed in a population, and within these population groups on an individual level [21, 24, 71]. The Theil index, on the other hand, supports decomposability – the inherited property from information theory [22]. This index is size-dependent, which was viewed by its creator - Henry Theil - as an advantage [81]. It captures inequality on an individual level, implying that the main contributory factor to the Theil’s value is the probability of an occurrence of an individual value in a distribution [81]. Thus, we refer to this index as a micro level measure that captures inequality between individuals and groups of individuals.

Although inequality indexes can be employed for software studies [54,55,70,71,85,86,91,92], a naïve application of inequality measures for these purposes is not without risk. First, there is an issue relating to the interchangeableness of the Gini coefficient and the Theil index. When applied for software analysis, both indexes are often strongly correlated [92]. However, they do not convey the same information – the Gini coefficient and the Theil index capture different inequality facets. Thus, even though these two indexes appear to behave similarly, it is inappropriate to assume that they are identical.

Second, the fields of economics and software engineering study systems using differ-
ent underlying principles, in particular in the way the occurrences of zero instances are
treated. This dissimilarity is especially evident upon a closer inspection of the Theil in-
dex. The computation of the Theil index contains a logarithm as a sub-term and the
logarithm of 0 is undefined. In economics there is no notion of zero value when it comes
to income inequality analysis. An individual with zero income “vanishes” from the pop-
ulation [38, 81] because their influence is assumed to be (effectively) nothing. Still, it is
natural for software metrics data to contain zeros. In this case, a direct application of the
Theil index is possible but undesirable. Even though there exists a special rule to handle
zero values for this index, it will distort the actual inequality value which leads to an eco-
logical fallacy in data interpretation [63]. Also, arbitrarily adding one to all elements [48]
or replacing 0’s with indefinitely small value [92] has been shown to be an invalid data
transformation technique as it leads to falsification of the inequality level in the distribu-
tion [48]. The Gini coefficient, on the other, is not affected by the ample presence of zero
instances. Thus, this inequality measure can be applied for metrics data analysis directly,
without much precaution.

Nevertheless, the Theil index can still be used for software studies. Its application, how-
ever, is conditional. This measure can be employed for software metrics analysis only
if one is prepared to lose some information by eliminating zero values. Thus, we are
making a conscious decision to remove this valuable data. If we require to perform an
inequality analysis using both the Theil index and the Gini coefficient then, in addition to
calculating the Gini coefficient on a full data set, we should compute the Gini coefficient
ignoring zeros to facilitate meaningful comparison between the Gini coefficient and the
Theil index.

2.2 Software Evolution

In biology, the term *evolution* emerged from the studies of Charles Darwin on the princi-
iples of natural selection and, hence, is defined as the exclusive driving force of *random
mutations* sorted out by *natural selection* from one generation to the next [95]. Natural
selection, in turn, was formalised by Darwin as a process of preservation of favourable
variations and rejection of less-favourable variations of species with a consequent extinc-
tion of the former ones [28]. Thus, evolution is the *entropic* process, since it is built
upon a balance between randomly occurring events, causing chaotic behaviours, and a
normalisation process of Darwin’s theory which eliminates created disorder.
The term “evolution”, however, is not exclusive to the field of biology. Multiple disciplines are applying it with a similar fundamental idea: the preservation of favourable properties and elimination of less usable ones. In software engineering, the term “evolution” refers to the continuous progression on the development of a software system [44]. This is due to the existence of multiple system versions, where each consequent version is a modification of a previous one [85]. Modifications are often governed by user preferences, defects fixes, or superior development solutions, aspects collectively called software evolution.

Vasa [85] noted that there are differences in the way “software evolution” is conceptualised across various sub-disciplines of software engineering. Often it is either a substitute for software maintenance [12], a process of introducing discrete changes to software, or an outcome of the changes. For the purposes of this work we agree with the definition by Bennet and Rajlich [12] which states that software evolution is a process enclosing the initial development of a system and continuous updates to this system driven by system environmental changes.

Software evolution is not a chaotic process. In fact, it is governed by a set of empirically derived generalisations called Lehman’s Laws of Software Evolution [43, 45] defined by Lehman et al. [43, 45] and listed in Table 2.1. These laws establish that - regardless of domain, size, or level of maturity - as software systems evolve, they gain in complexity and require more resources in order to retain their value. Lehman’s Laws of Software Evolution describe natural characteristics of incremental transformation experienced by evolving software systems along with the social environment in which these systems are developed [85]. Thus, these laws capture not only the change taking place, but also the context in which this change occurred [85].

Multiple factors influence software evolution. These include the rationale for software change, user feedback, implementation approach, or capability of a current system design [12, 14, 85, 88]. Bennett et al. [12] argue that the two most common reasons for a system to evolve are requirement changes and maintenance operations. Nevertheless, the quantity and quality of these factors, along with possible marketing pressures, affect the rate with which a system grows [33, 54].

However, the Laws of Software Evolution depict size and complexity growth of evolving software systems, but not the nature of these changes [85]. In particular, one might need to know at which rate software systems evolve and what affect such growth rate has on an
Table 2.1: Lehman’s Laws of Software Evolution as defined in [44]

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Continuing Change</strong></td>
</tr>
<tr>
<td></td>
<td>A system that is used must be continually adapted or it becomes progressively less satisfactory.</td>
</tr>
<tr>
<td>2</td>
<td><strong>Increasing Complexity</strong></td>
</tr>
<tr>
<td></td>
<td>As a system evolves its complexity increases unless work is done to maintain or reduce it.</td>
</tr>
<tr>
<td>3</td>
<td><strong>Self Regulation</strong></td>
</tr>
<tr>
<td></td>
<td>A system evolution process is self regulating with close to normal distribution of product measures and process attributes.</td>
</tr>
<tr>
<td>4</td>
<td><strong>Conservation of Organisational Stability (invariant work rate)</strong></td>
</tr>
<tr>
<td></td>
<td>The average effective global activity rate on an evolving system is invariant over the product lifetime.</td>
</tr>
<tr>
<td>5</td>
<td><strong>Conservation of Familiarity</strong></td>
</tr>
<tr>
<td></td>
<td>During the active life of an evolving system, the content of successive releases is statistically invariant.</td>
</tr>
<tr>
<td>6</td>
<td><strong>Continuing Growth</strong></td>
</tr>
<tr>
<td></td>
<td>The functional content of a system must be continually increased to maintain user satisfaction over their lifetime.</td>
</tr>
<tr>
<td>7</td>
<td><strong>Declining Quality</strong></td>
</tr>
<tr>
<td></td>
<td>The quality of a system will appear to be declining unless it is rigorously maintained and adapted to a changing operational environment.</td>
</tr>
<tr>
<td>8</td>
<td><strong>Feedback System</strong></td>
</tr>
<tr>
<td></td>
<td>Programming processes constitute multi-level, multi-loop feedback systems and must be treated as such to be successfully modified or improved.</td>
</tr>
</tbody>
</table>

empirical software analysis of evolving software systems [85].

Vasa [85] has already discussed these questions. He stated that one common assumption is that software systems advance at a single growth rate over the entire evolution history. In this model, the size growth is frequently specified as sub-linear, linear, or super-linear. However, such undeviating growth rate is rather exceptional. Upon a closer inspection, it becomes evident that software systems “exhibit segmented and uneven growth pattern” [85]. They can advance at variant growth rates over separate time periods. Also, the growth pace may vary dramatically in between system’s modules. Such findings indicate that developers routinely restructure and reorganise their solution in order to prevent or eliminate possible issues related to size or complexity of a system.

Vasa [85] also noted that the segmented nature of software development places certain constraints onto evolutionary analysis techniques. To compare a systems evolutionary trends we require a reliable and consistent measurement of time. One approach to capture this data is via a collection of an actual release date for each system version. But, due
to segmented nature of software growth a comparative analysis on such level might be misleading. The progression rates between two separate systems almost never align [85, 86]. Therefore, Vasa [85] has concluded that there is a need for an approach that would uniquely identify a specific version independently to the time devoted for its delivery.

To achieve this, Vasa [85] adopted a pseudo-time measure, called Release Sequence Number or RSN [85], that represents constant time intervals between consequent releases of the same system. RSN is allocated incrementally for each version established by its release date. Its main advantage is an ability to pinpoint, directly, a specific version in a history of a software system [85].

Such a time measure is especially beneficial for this work, as it is not concerned with the growth patterns but rather with discrete release (volume and time independent) events of a system. RSN offers an intuitive ordering criterion that can facilitate comparative analysis of release histories between different systems at minimal costs.

### 2.3 Metrics and Data

*Measurement* and *validation* are two fundamental characteristics to the software engineering discipline [80]. Measurement offers a range of approaches aimed to build a better understanding of a system’s solution design and development process [80]. However, measurement often has limited applicability, since we rarely comprehend the relationship between certain quality attributes and collected metrics data [9, 80, 87]. Thus, measuring code alone is insufficient for sound conclusions. We require a model explaining the relationship between collected data and corresponding software traits as well as experiments validating this model [9, 80].

Validation establishes the correctness of extracted information and provides a means for sound conclusions and data interpretation [9, 80]. Though, validation cannot be achieved by solely running a single experiment. The experiments must be repeatable [80]. Moreover, they have to be accompanied by an appropriate measurement approach in order to evaluate claims as either genuine or spurious [9].

To ensure *repeatability* and *reliability* of our studies, this work uses a curated data set of software systems – the *Helix* repository [87] – to perform a range of software investigations. And since this thesis is concerned with characteristics of evolutionary changes
CHAPTER 2. THE ANALYSIS OF SOFTWARE METRICS DATA

in software systems, the selected collection suits the needs perfectly; because the Helix repository was setup to explicitly enclose historical advancement of software systems [87]. Helix comprises 42 non-trivial open-source Java software systems. Each system contains at least 100 classes, had been under development for 18 month or longer, and has a minimum of 15 releases during a continuous growth history [85]. In total, the Helix repository represents approximately 1,200 releases and 65,500 classes.

Table 2.2: Selected static code measures

<table>
<thead>
<tr>
<th>Measure</th>
<th>Metric</th>
<th>Description</th>
<th>Association</th>
</tr>
</thead>
<tbody>
<tr>
<td>ck</td>
<td>cbo</td>
<td>Coupling between object classes</td>
<td>Complexity</td>
</tr>
<tr>
<td>ck</td>
<td>rfc</td>
<td>Response for a class</td>
<td>Communication</td>
</tr>
<tr>
<td>ck</td>
<td>wmc</td>
<td>Weighted methods count</td>
<td>Decomposition</td>
</tr>
<tr>
<td>degrees</td>
<td>indegree</td>
<td>Number of classes class C depends on</td>
<td>Popularity</td>
</tr>
<tr>
<td>degrees</td>
<td>outdegree</td>
<td>Number of classes depending on class C</td>
<td>Delegation</td>
</tr>
<tr>
<td>gsummary</td>
<td>getters</td>
<td>Number of getter methods</td>
<td>Exposure</td>
</tr>
<tr>
<td>gsummary</td>
<td>setters</td>
<td>Number of setter methods</td>
<td>Exposure</td>
</tr>
<tr>
<td>nof</td>
<td>noa</td>
<td>Number of attributes</td>
<td>Density</td>
</tr>
<tr>
<td>nom</td>
<td>nom</td>
<td>Number of methods</td>
<td>Size</td>
</tr>
</tbody>
</table>

For the purposes of this work nine software metrics were selected (see Table 2.2), also called static code metrics, that are associated with structure, size, complexity, and design studies of software artefacts. These include:

- three measures from CK object-oriented metrics suite [19] – WMC, RFC, and CBO,
- in-degree and out-degree counts reflecting structural properties of a system,
- getter and setter method counts emulating information hiding principle,
- number of fields (NOF) - system’s state representation, and
- number of methods (NOM) which is a size measure.

Only three out of the six CK metrics were selected. The DIT and NOC measures lack expressive power [76], whereas LCOMs mathematical validity has been challenged [33, 54]. Thus, these metrics were not included in this study.

To collect metrics data from software systems the Java Code Tomograph (jCT) [52] was developed and used. jCT is a metrics extraction tool for the Java programming language that analyses single or a collection of binary files (i.e., .class or .jar) and extracts metrics data from these files based on predefined configuration settings [52]. The tool is especially beneficial for this work, since it incorporates a data extraction mode designed specifically for software metrics studies from the Helix data set [52].
2.4 Summary

Software systems are studied in order to improve productivity and effectiveness of the associated software development process. There are a number of approaches available to do so, such as model-building or empirical analysis using software metrics. In this work we are interested in the later one.

Software metrics data alone does not provide intuitive results. We need to summarise such data in order to interpret and compare evolutionary changes. Determining which summary statistics would be the most beneficial for these purposes is crucial for achieving relevant and viable results. Central tendency statistics, such as median and standard deviation, may be unable to interpret software metrics data meaningfully, since they assume normally distributed data. Software development, though, is not a random process, and thus we cannot expect the result (i.e., the software product) to exhibit randomness either. For this reason, we borrow inequality measures from the field of economics to analyse data dispersion changes. Such approaches are especially useful for evolutionary studies of software systems since they are very sensitive to changes in metrics distributions.

The studies conducted for this thesis utilise the Helix repository – a curated data set which contains release history for 42 non-trivial Java software systems. Nine metrics are collected that cover complexity, design, size, and structural aspects of software systems. To retrieve this data the Java Code Tomograph is used – a powerful metrics extraction tool especially designed to support flexible data mining of curated software repositories like the Helix data set.
Chapter 3

Software Metrics Extraction and Analysis Tools

Empirical software analysis can be a challenging and unreliable task if not supported by an appropriate set of facilitating tools [13, 94]. Although multiple software systems exist designed to simplify and to automate empirical software studies, it is often the case that these systems do not meet expectations or requirements of a new research. Thus, the Java Code Tomograph or jCT was developed to fit the goals of this study. It is a metrics extraction engine that offers researchers the freedom to select their own metrics collection approach. Its main features are precision data mining and extensibility. However, gathered static code metrics need interpretation. Therefore, a number of supporting tools are employed to investigate theories about the software engineering processes.

In this chapter, Section 3.1 motivates the development of jCT. It provides an overview of existing metrics extraction tools, their associated limitations, and how jCT overcomes these issues. This section also describes the organisation of jCT along with its main features, such as the metrics extraction engine organisation and support for large-scale software studies. This is followed by a demonstration of jCT’s extensibility by presenting a jCT measure definition example. In Section 3.2 a list of all auxiliary tools that have been used and developed to assist this work is given including utilized statistical software and any additional tools that were developed to enrich this study. Lastly, Section 3.3 summarises this chapter.
3.1 jCT - Java Code Tomograph

3.1.1 The Need for jCT

Software engineering, similar to other disciplines, relies heavily on a "high level approach for evolving the knowledge" in the area [9]. This approach is defined as a cycle of model building, experimentation, and learning activities [9, 52]. Simple observations resulting in logical conclusions do not suffice here [9, 52]. Thus, software engineering can only succeed if it is provided with a robust experimental component exercised to (i) study theories regarding software nature [86], (ii) investigate development patterns [54, 86, 88], (iii) analyse evolutionary mutations and their causes [90], and to (iv) assess the health of and possible threats to a system [8, 57].

Measurement is an essential component here [14, 19, 79]. Identifying a set of characterising attributes, collecting matching software metrics, and performing an empirical analysis on this data are measurement activities required to understand a software system and its underlying development processes [33, 86].

However, "measurement is never better than the empirical operations by which it is carried out" [74]. Data precision is paramount [52]. Premature application of summary statistics or an early introduction of some level of abstraction in a data mining process can arbitrarily limit software analysis and even distort collected data, leading to incorrect interpretations [74, 86]. Hence, to avoid these issues, software metrics data should be extracted at the highest possible level of granularity [74]. Appropriate post-processing activities can eliminate potential noise, if necessary [52].

To achieve this, the tools that facilitate and automate rigorous software measurement are needed. There exists already a number of facilitating frameworks and tools aimed for metrics extraction activities, both open source and commercial (e.g., JMetric [1], JSeat [3], Mutations [5], ASM [15], BCEL [2], etc.). However, each of the considered options has its own limitations making it somewhat unsuitable for the empirical software studies discussed in this work.

For example, both JMetric and JSeat systems use a core meta-model which maps directly to Java language semantics. Such a solution design is cumbersome when performing analysis on large software systems such as Netbeans or Eclipse, although it offers some benefits in terms of data precision by virtually restoring a system. Also, neither tool has
a built-in support for automatic processing of large data collections, such as the Helix repository [87]. Relatively slow performance and lack of an automated support for large-scale data mining operations makes those tools inconvenient for this study.

ASM, BCEL, and Javassist [73], on the other hand, come with a light-weight meta-model, making it possible to process large software systems. Each framework was designed to facilitate Java byte code engineering procedures or to support aspect-oriented programming [2,15,73]. Thus, none of them were built to perform metrics extraction activities per se; yet, they can be applied for these purposes. For instance, the ASM framework is used by JSeat and Mutations as a class file parser. Also, both ASM and BCEL build the foundation of FindBugs application aimed for the statistical analysis of Java software [78]. The underlying meta-models in ASM, BCEL, and Javassist allow for software analysis, though they omit some details regarding the language semantics during the extraction process, making these frameworks less suitable for high precision data mining. Therefore, if a study requires data accuracy, which one of the key factors of this work, none of these frameworks are ideal for data collection.

Even though a meta-model can limit data analysis, this type of solution seems to be quite popular in software engineering. Moreover, there are tools that capture a meta-model in a language-independent way. One example is FAMIX [82, 83] which provides a language-independent representation of object-oriented programs. It models software systems at the entity-level, suggesting that relationships between classes, methods, and invocations are captured, but the complete abstract syntax tree is omitted, which allows for large systems to be studied. The language independence of FAMIX provides an opportunity to reason about the programming languages on a more generalised and abstract level [82]. However, even the developers of FAMIX have noted that certain problems require language-specific solutions in order to be resolved [82]. Thus, high precision data mining is difficult with FAMIX also.

Overall, in the review conducted for this work no available tools were identified that completely satisfy our requirements: precision data mining, low maintenance, extensibility, and support for repeatable studies. Therefore, the Java Code Tomograph or jCT was developed. In addition, the process of its implementation allowed for building a solid understanding of the Java programming language prior to commencing the inequality investigation.

jCT is both a high-resolution data miner and a powerful tool for performing quantita-
3. SOFTWARE METRICS EXTRACTION AND ANALYSIS TOOLS

tive analysis of Java code [52]. The key features of jCT are rigorous measurement and an extensible metrics extraction infrastructure [52]. jCT does not rely on a meta-model for internal representation of analysed systems. Instead, it applies the Java-7 specifications [16,67] as a standard for representing classes in memory. No information is omitted during the extraction process. This approach may be considered as the most precise representation of Java code. Such a solution, though, implies that each programming language requires its own extraction mechanism, however it is argued that language specifics have a major influence on software organisation [52].

3.1.2 jCT Organisation

jCT is a stand-alone application which operates on byte code rather than source code. This is due to the findings by Lumpe et al. [54,86], who determined that compiled Java class files contain almost the same information as source files, and hence, byte code can be considered isomorphic to source code. In addition, byte code provides direct access to system’s data since it is not polluted by source-level syntactic sugar and auxiliary information [52].

Byte code representation of a Java software system is essentially a sequence of binary numbers organised into a collection of .class files [47]. Thus, jCT needs to parse and interpret Java byte code in order to study it. It would have been possible to adapt one of the existing frameworks that facilitates byte code analysis, such as ASM [15]. However, neither of the considered options provide a sufficient level of data precision. Hence, jCT incorporates a custom byte code parser that maintains data accuracy.

In jCT, binary data is stored in a dedicated format that is almost identical to the original Java language specification [47,67]. Consider the card view of a compiled Java Class File structure in Table 3.1. Unlike some existing metrics extraction tools, such as FAMIX [82, 83] or JSeat [3], jCT does not use a meta-model to map measured artifacts to language-independent representation or to facilitate the metrics extraction process. Instead, jCT applies language semantics for software analysis, ensuring minimum information loss during the extraction process.

When performing data mining operations, jCT loads the analysed system attributes into an inbuilt data collection which is used to perform software metrics data collection. In addition, for each system jCT constructs a type-dependency graph [88], which is a representation of inter-class relationships using fan-in and fan-out information [86,90]. It
Table 3.1: "Card" view of a compiled Java Class File structure as specified in [47]

<table>
<thead>
<tr>
<th>Field</th>
<th>Access Flags</th>
<th>Name and Descriptor</th>
<th>Attributes</th>
<th>Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fields</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Methods</td>
<td>Access Flags</td>
<td>Name and Descriptor</td>
<td>Attributes</td>
<td>Annotation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attributes</td>
<td>Attributes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

can be used to facilitate graph-based analysis (i.e., clustering studies) of software systems (see Figure 3.1).

Even though jCT includes a number of predefined metrics [53], it does not require any default measures to be implemented per se. Instead, it employs a light-weight extension mechanism to supply desired metrics definitions to it [53]. This mechanism is viewpoint-agnostic, meaning it does not rely on any predefined analysis model. As a result, jCT offers users the freedom to create measures that match an individual researcher’s needs precisely. With jCT one can perform graph analysis, reverse engineering analysis, or statistical analysis with minimal thought as to how a specific approach fits into an overall metrics extraction process.

Every measure in jCT has to be developed with respect to the required recording process – class-by-class. This implies that each measure must implement MetricsCollector class - an extension point that ensures that newly introduced metric definitions are compatible with jCT. MetricsCollector enforces a strict separation of concerns [40], meaning every defined metric class is responsible for extracting data from a specific class only. The application of this metric to all classes is a responsibility of the metrics extraction engine, resembling a custom visitor pattern implementation.

MetricsCollector is an abstract class which contains five core methods; four of those are abstract and have to be implemented by a new measure. The abstract methods
are flushHeader, setup, setOption, and extractDataPass1. The flush-
Header method is responsible for setting up suitable header information for an output
file of a measure. The setup method is used for initialization of any local variables used
in a measure instance. Both flushHeader and setup methods may be empty if no
information of such kind is necessary. The setOption method is responsible for linking
a newly defined measure to the metrics extraction engine. This method must specify an
option name and description for a measure that are used to register it with jCT.

extractDataPass1 is a method responsible for collecting the required information
from a class file. Sometimes, though, measured attributes experience mutual dependen-
cies (e.g., when studying inner classes or coupling between object classes). To resolve
mutual dependencies, a second pass method is used - extractDataPass2. Thus,
extractDataPass1 has a compulsory implementation and, hence, has to be present
in all defined measures, whereas extractDataPass2 can be added to a measure only
if mutual dependencies between examined attributes occur.

Currently there are over 300 measures implemented in jCT, excluding IL (Intermedi-
ate Language) instruction counters. Measures range from the simplest, such as counting
number of classes per package (-cpp option) or number of methods of different types
(-nom option), to more sophisticated ones. For example, reverse engineering opera-
tions (Individual Class File Writer -isummary option) or getter/setter variations analy-
sis (-gssummary option). The resulting data for each measure is often organised into a
set of strictly defined .csv files (see Appendix A): such formatting ensures high compat-
ibility with the majority of statistical analysis software which significantly simplifies the
data interpretation phase of software engineering studies.

One of the key features of jCT is that it facilitates repeatable empirical studies by pro-
viding an inbuilt functionality for processing curated software repositories, such as the
Helix [87] data set. To achieve that, it is equipped with built-in modes, that perform data
mining operations on both software repositories and separate systems or files.

For instance, the Helix repository processing mode is invoked by -helix option. In this
mode, jCT traverses the Helix directory, system by system, and loads each version into
the corresponding jCT data collection. During the data mining process, it also utilises
.versions meta-data file. This file is associated with each system and contains a number
of labels that identify certain properties of a corresponding Java software system and its
releases [85, 87]. To characterise separate versions in this work, the RSN is used. The
.versions file provides the mapping between a specified release and its RSN.

Data extraction with jCT is $O(n)$. This applies to memory consumption and running time,
both of which are found to be linear to the size of a studied system [52]. On average, the
analysis of a single class with jCT takes close to 3ms to complete and requires approx-
imately 20kB runtime memory. These values, however, are estimates only and can vary
depending of the complexity of the metrics being collected and the nature of analysed
classes and systems.

3.1.3 Metrics Definition Example

jCT provides a user with the freedom to perform any empirical data collection operations.
To demonstrate this, the sample jCT measure implementation is provided which defines
CBO (Coupling Between Objects) metric from the CK (Chidamber and Kemerer) Object
 Oriented Design metrics suite [19]. In fact, this metric is one of the selected measures
used to support this work (see Section 2.3).

CBO is a metric that represents a number of other classes to which the studied class
is coupled to [19]. A class is coupled to another class if methods of this class are using methods or instance variables (fields) defined in another class. In other words, two classes are coupled if one acts on the other [19]. This definition is quite similar to the fan-out count which captures the number of classes the current class uses [88].

To provide this metric with implementation we need to (a) build a data set that depicts inter-class dependencies in a system and (b) once the dependencies are evaluated, output the number of outgoing relations of each class. Since the data mining process with jCT is performed on a class-by-class basis, to implement CBO measure two data passes are required: one to build dependencies, and the second one to output collected data.

jCT is equipped with all abstractions necessary to implement the CBO measure. It offers a type-dependency graph which is explicitly designed to capture fan-in and fan-out relations between classes. In addition, it supports two-pass data processing. The possible CBO metric implementation in jCT is shown in Listing 3.1.

### Listing 3.1: CBO.java Class

```java
public class CBO extends MetricsCollector {
    private TypeDependencyGraph fTG;

    public void setOption( Measure aMeasure ) {
        aMeasure.setOptionName( "cbo" );
        aMeasure.setDescription( "CK’s CBO" );
    }

    protected void flushHeader() {
        getWriter().println( "Class,CBO" );
    }

    protected void setup() {
        fTG = new TypeDependencyGraph();
        fTG.populate( getDataSet() );
    }

    protected void extractDataPass1( JavaClassFile aClassFile ) {
        if ( aClassFile.containsNonSyntheticCode() )
            fTG.addNode( aClassFile );
    }

    protected void extractDataPass2( JavaClassFile aClassFile ) {
        TypeNode tn = fTG.lookup( aClassFile );
    }
}
```
When collecting CBO metrics data with jCT, the result of this metric is listed in a `.csv` file. Depending on a processing mode, there might be one or multiple associated output files. For instance, when performing data mining operation on the Helix repository, each version of each system will have its own file containing the associated class names and their CBO values. The format of an output file is presented in Listing 3.2.

### Listing 3.2: jCT output for the CBO metric

```java
Class, CBO
com.oreilly.servlet.MailMessage, 1
com.oreilly.servlet.MailPrintStream, 1
org.apache.tools.ant.BuildEvent, 3
org.apache.tools.ant.BuildException, 1
org.apache.tools.ant.BuildListener, 0
org.apache.tools.ant.DefaultLogger, 6
org.apache.tools.ant.DesirableFilter, 0
...
```

The CBO example shows that, potentially, jCT is capable of incorporating various data mining procedures, as long as the class-by-class metrics extraction process is maintained.

### 3.2 Auxiliary Tools

By collecting software metrics jCT allows us to rationalise a system’s design and its underlying processes. However, metrics alone can be misleading [80]. Thus, static code metrics retrieved with jCT should be rearranged, summarised, or both to enable us to evaluate theories about corresponding software engineering features. For this reason, a number of auxiliary tools were developed and used to facilitate automatic post-processing of metrics data and to provide additional information regarding inequality indexes usage for this thesis.

To facilitate inequality indexes behaviour studies, in particular inequality indexes experimentation (Chapter 5), *Inequality Experiments Runner* was developed. The main purpose of this tool is to simulate a specific, and well-understood, distribution of values, to perform a number of updates to this distribution (the Pigou-Dalton distribution principle [68]), and
to calculate the Gini coefficient and the Theil index values for each change in a data set. The aim of such experimentation is to determine how the Theil index and the Gini coefficient perform in a controlled environment. This tool can be configured with an output file location and the size of a data set to be rendered.

The Inequality Experiments Runner was built to extract inequality measures on a specific distribution only. However, it lacks the ability to perform analysis on collected software metrics. Thus, Inequality Calculator was designed which is another tool supporting this work that calculates the Gini coefficient and the Theil index on data produced by jCT. jCT measures are distilled to .csv files. So, the Inequality Calculator uses corresponding header fields to define which metric should be used for distribution studies. Thus, it is configured with a metrics file path, an output file path, and a list of columns to be included in calculations. In addition, if a user would like to consider multiple columns as one and calculate inequality measures on such summary column, Inequality Calculator will combine specified columns automatically and calculate the indexes on resulting values.

Sometimes the default output format of a software metric (as defined by a jCT measure) can cause difficulties in post-processing activities. To eliminate possible data reorganisation issues, we developed CSVProcessor. The main purpose of this tool is to read data from all input files, line by line, and output it in a specified format. Thus, CSVProcessor is configured with an input file path, an output file path, and a list of columns to be included in an output file. Similar to the Inequality Calculator tool, CSVProcessor is able to combine data from multiple metrics into one.

One of the aims of empirical software analysis, and software evolution studies in particular, is to be able to compare collected information across systems or releases to develop and support theories about software engineering activities. Therefore, there is a need for tools that allow us to visualise collected data, compare it, and perform further statistical analysis if necessary. Fortunately, there are various applications which correspond to these requirements. For this work we used R [6], the project for statistical computing, and STATA [50], the data analysis and statistical software. Both tools support .csv files imports and can execute any statistical calculations operations necessary for the purposes of this thesis.
Chapter 3. Software Metrics Extraction and Analysis Tools

3.3 Summary

There is a clear need for reliable software analysis tools to develop and evaluate theories regarding software engineering processes and outcomes. Thus, a number of software systems and frameworks have been developed by the community to achieve that, such as JSeat [3] or Mutations [5]. However, even though there are multiple metrics extraction tools available, it is often difficult to find one that fits individual research goals precisely. To facilitate this study a separate tool was developed called jCT, or the Java Code Tomograph.

jCT is both a high-resolution data miner and a powerful tool for performing quantitative analysis on Java byte code. Its main features are precision data collection, an extensible metrics extraction engine, and built-in support for recently emerged curated repositories, such as Helix. jCT offers users a freedom to perform virtually any type of software analysis. It also exhibits a consistent (linear) running time and memory consumption profiles.

However, a metrics extraction tool alone does not suffice for software studies. We require to perform post-processing activities to accumulate and compare collected software metrics in order to support or arise theories about software development. Thus, a number of auxiliary tools were created for this work. These include the CSVProcessor, Inequality Experiment Runner, and Inequality Calculator tools. Some statistical analysis tools are also employed, such as R and STATA, to aid metrics data interpretation.
Chapter 4

Inequality-based Software Analysis

The technique of measuring inequality is over a century old [48] and has been employed to examine population welfare and economical development. The application of this approach, however, is not limited to economical studies. For instance, inequality measurement has been adopted by multiple researchers in the software engineering discipline [54, 71, 85] to interpret software metrics data.

However, a naïve application of inequality measures for software engineering studies does not suffice. To gain a better understanding of what these measures capture and how to map collected information to software concepts, we need to determine:

- Why do inequalities occur?
- What affects inequality in distribution?
- Can inequality measures be applied directly for software studies safely?
- How does the nature of inequality measures influence software examination results?

This chapter covers all of these concepts.

In this chapter, Section 4.1 presents the required for this work inequality indexes. These include the Lorenz curve, the Gini coefficient, and the Theil index. Section 4.2 lists the causes of inequality in the field of economics and recommends a potential mapping of those causes to software engineering principles. Next, possible implications of a direct application of these indexes for software analysis are considered, in particular zero issues, and an alternative way to define the impact of these issues onto the inequality value is proposed (Section 4.3). Finally, this chapter is summarised in Section 4.4.
4.1 Inequality Indexes

In the field of economics, the measurement of inequality has been successfully employed to explain numerous socio-economic topics [7, 21, 48, 68]. The main application of this technique, however, is analysing the distribution of wealth (income) in a population to determine its current state or the success of existent policies such as taxation [7, 48].

The relation between economics and social welfare is a close one [18, 68] – in fact they are mutually dependant [68]. It is evident that high inequality causes, in one way or another, extreme poverty, economical issue prioritisation, political and social conflicts and egalitarian policy performance [18]. But, it is also important to realise that all those issues, and more, also influence the associated inequality levels in a society. Thus, economists found themselves in need to study the nature, meaning, and causes of inequality to evaluate associated living standards in a society [18, 68].

Inequality measurement is a pursuit of meaningful comparison of income distributions in terms of factors derived from ethical principles, mathematical constraints, or simple intuition [24]. The principle motivation for measuring inequality has traditionally been normative, for instance, to guide socio-economic policies [7, 41, 48]. If an inequality level in a population has been growing, then it could be an indication for a need to stabilise income dispersion levels [41]. For example, in the case of a taxation policy management, inequality may imply a need to extend redistribution in the tax and transfer system to even income allocation [41]. Inequality analysis is not limited to existent socio-economic systems, and may be used to verify a policy proposal since it provides a means to predict the effect a certain policy may have [41].

Multiple approaches have been proposed to examine inequality [48]; only a few, however, are considered to be effective [7]. In particular, when studying income distributions economists give preference to the ranking measures, such as the Lorenz curve [48], the Gini coefficient [36], and the Theil index [81], due to their indifference to the underlying distribution profile [7, 24].

The following sub-sections present overviews of each inequality index and their underlying calculation criteria.
4.1.1 Lorenz Curve

Lorenz [48] has been an early critic of the arbitrary usage of central tendency statistics to summarise unequal distributions. In his seminal paper [48] he considered all existent approaches and concluded that unless changes in wealth allocation and population are studied simultaneously, thus comparing two diverse distributions, it is quite difficult to accumulate reliable results.

For this reason, Lorenz [48] proposed his own method, now called the *Lorenz curve*, which is a graphical representation of the inequality in a distribution. This method plots on the x-axis the cumulative percentage of the population, sorted from poorest to richest, and along the y-axis the cumulative percentage of resources held by the corresponding population share [48].

Mathematically, for a set of non-negative values \( X \), such that \( \{x_k \in X | 1 \leq k \leq n \ and \ x_k \leq x_{k+1} \} \), the Lorenz curve is a continuous linear function connecting the points \((F_i, L_i)\), \(0 \leq i \leq n\) where \(F_0 = 0, L_0 = 0\), and for all \(i\) in \(1 \leq i \leq n\):

\[
F_i = \frac{i}{n} \quad \text{and} \quad L_i = \left( \frac{\sum_{j=1}^{i} x_j}{\sum_{j=1}^{n} x_j} \right)
\]

---

**Figure 4.1:** Lorenz curve example

The Lorenz curve is a non-decomposable, macro-level aggregation function. Its shape reflects the degree of inequality in a distribution [48]. For example, if the Lorenz curve is a straight line, then it indicates *perfect equality* as shown in Figure 4.1. If a distribution is
not equal, then the Lorenz curve will deviate to a certain degree also shown in Figure 4.1. For perfect inequality (maximum), the Lorenz curve encloses the complete area beneath the straight, equality line. Thus, the rule of interpretation states as “the bow” is bent, the concentration increases [48].

Another important observation made by Lorenz [48] is on the effects that additive and multiplicative operations have on the level of inequality in a distribution. Lorenz [48] noticed that all additive operations change distribution thus affecting the corresponding inequality level. Inequality, however, is invariant to multiplicative data transformations. Hence, the shape of the Lorenz curve remains the same after multiplication operations are applied [48].

### 4.1.2 Gini Coefficient

The Lorenz curve is a powerful tool for visually representing inequality. Nevertheless, it is often more practical to operate with a single numeric value, especially when performing comparative studies. For these purposes, the Gini coefficient [36, 68] can be used, since it is defined as the area between the line of equality and the Lorenz curve. Figure 4.2 shows this concept in detail.

The Gini index can be computed for both continuous and discrete distributions. In this work we employ the later form. Thus, for a set \( X \) of \( x_i \) values, where \( 1 \leq i \leq n \) and \( n \) is the size of the set, indexed in an ascending order (\( i.e. \ x_i \leq x_{i+1} \)), the discrete Gini
coefficient \((G)\) is calculated as follows:

\[
G = \left(\frac{1}{2n^2 \mu}\right) \sum_{i=1}^{n} \sum_{j=1}^{n} |y_i - y_j| = \frac{\sum_{i=1}^{n} (2i - n - 1) x_i}{n \sum_{i=1}^{n} x_i} \tag{2}
\]

The Gini coefficient is not a simple aggregation function. It calculates inequality using relative mean differences between asymmetric pairs of values [66, 68]. The result of the Gini coefficient lies within the interval \([0, 1]\) [68]. If all values in a set are the same (so \(\bar{x}\) is the mean of set \(X\) and \(\bar{x} \neq 0\)), then the Gini coefficient is at minimum (0) which presents the perfect equality in a distribution. The upper bound of the Gini coefficient approaches 1; and if all values in the set are zeros except for one, then the Gini coefficient is at maximum and equals to \(1 - \frac{1}{n}\) [11].

The Gini coefficient, similar to the Lorenz curve, is a size-indifferent macro-level aggregation function [86] with an intuitive interpretation. Moreover, due to its high readability, in economics the Gini coefficient is commonly referred as “the gold standard” [96] for inequality measures.

### 4.1.3 Theil Index

Shannon [72] studied the surprise effect of an event occurrence in information content using event probability [72]. In 1967, Henry Theil [81] used Shannon’s information theory [72] to define a new measure that used the entropic notion of disorder to capture inequality in a distribution. He applied Shannon’s theory utilising income shares instead of probabilities and considered this data in relation to the maximum entropy value in a population [22, 25, 81]. This resulted in a measure with a puzzling construction and non-intuitive nature; nevertheless, it provides a simple way to aggregate grouped data [68].

Theil defined inequality as the difference between the total equality and the population entropy. Knowing that, for a discrete set of positive values \(x_i, 1 \leq i \leq n\), where \(n\) is a number of elements in the set, the individual Theil index, denoted as \(T_I\), is computed as follows:

\[
T_I = \sum_{i=1}^{n} \frac{x_i}{X} \log \left( \frac{x_i}{X} / \frac{1}{n} \right) \tag{3}
\]

where \(X = \sum_{i=1}^{n} x_i\).
The Theil index is size dependant [81]. Its value falls in the interval \([0, \log(n)]\) [22, 35, 38]. Henry Theil considered the size dependency a crucial property. He argued that an inequality measure should not only quantify inequality in a distribution, but also "qualify" it to some degree [81]. Such reasoning implies that inequality in a population with a thousand individuals, where one individual holds all resources, is more severe than inequality in a population of two individuals [81]. However, this can also create a misconception: inequality measures capture the degree of values dispersion, but they cannot explain it. To explain the causes of inequality, we are required to identify the parametric variations [7] that are affecting it.

The Theil index is a micro-level statistic. It captures dependencies between individual values in a distribution and their response to each other [21, 81]. Still, the tight relationship of the Theil index to the number of elements in a set makes this index hard to interpret.

There is, however, a normalisation function for this index. This normalisation transforms the Theil index value making it comparable with the Gini coefficient. So, if the individual Theil index is \(T_I\), then the normalised Theil index, denoted as \(T^n_I\), is given by:

\[
T^n_I = 1 - e^{-T_I} = 1 - \frac{1}{e^{T_I}}
\]  

The value range for the normalised Theil index, \(T^n_I\), is \([0, 1]\), almost identical to the Gini coefficient. However, this does not imply the existence of the monotonic relationship between the Gini coefficient and the normalised Theil index, since they both use a different way to measure inequality in a distribution.

Although the Theil index has been accepted by many as a valid inequality measure in the field of economics, there are some that are critical. For example, Sen has stated in [68] that the Theil index is an unintuitive and arbitrary inequality measure and is not entirely suitable for economic analysis. Though, the relation of this measure to the concept of entropy does not allow for its dismissal yet [68]. Also, one of the major benefits of the Theil index is its high sensitivity to distribution shifts when inequality is already high [22]. However, this effect diminishes once normalisation occurs; hence, lessening the Theil index’s value for distribution examination.

Still, the Theil index is a powerful tool for hierarchical data aggregation [68]. Its decomposable nature allows us to reason about inequality not only at an individual level but also at a group level. The partitioned Theil index’s computation rules, however, differ from that of the individual Theil index. Specifically, the partitioned Theil index becomes a sum
CHAPTER 4. INEQUALITY-BASED SOFTWARE ANALYSIS

of “between-group” (\(T^b_G\)) and “within-group” (\(T^w_G\)) components [21].

Thus, the partitioned Theil index, denoted \(T_G\), is calculated as follows:

\[
T_G = T^b_G + T^w_G
\]

such that

\[
T^b_G = \sum_{g=1}^{m} X_g \log \left( \frac{X_g}{X} \frac{n_g}{n} \right)
\]

(6)

and

\[
T^w_G = \sum_{g=1}^{m} X_g \left( \frac{n_g}{X_g} \sum_{i=1}^{n_g} \frac{x_{gi}}{X_g} \log \left( \frac{x_{gi}}{X_g} \frac{1}{n_g} \right) \right)
\]

(7)

where \(X = \sum_{i=1}^{n} x_i\), \(X_g = \sum_{i=1}^{n_g} x_{gi}\), and \(x_{gi}\) is a value of a discrete element in a group.

To assist the understanding as to how the partitioned Theil index is computed, consider a sample distribution \(A\) where values are split into four groups as shown in Table 4.1. The individual Theil index, \(T_I\), and the partitioned Theil index, \(T_G\), for set \(A\) are the same (0.536434). Hence, partitioning the Theil index does not alter its value. Inequality in the distribution is invariant to decomposition.

Table 4.1: Sample partitioned distribution \(A\)

<table>
<thead>
<tr>
<th>Group</th>
<th>Members</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 to 10</td>
<td>1, 2, 5, 10</td>
<td>4</td>
</tr>
<tr>
<td>11 to 110</td>
<td>100, 101</td>
<td>2</td>
</tr>
<tr>
<td>111 to 1010</td>
<td>120, 290, 450, 510, 890, 1000, 1010</td>
<td>7</td>
</tr>
<tr>
<td>1011 and over</td>
<td>1011, 1012</td>
<td>2</td>
</tr>
</tbody>
</table>

Even though \(T_I\) and \(T_G\) are the same, \(T_G\) provides more information about inequality in a distribution. In particular, it allows us to distinguish inequality within groups (\(T^w_G = 0.10845\)) and inequality between groups (\(T^b_G = 0.42799\)). Moreover, the partitioned Theil index enables an access to the within group inequality for each group (see Table 4.2), and their corresponding subgroups if necessary [23].

To examine this property consider Table 4.2 which lists the within group inequality, \(T^w_G\), for each group in the population \(A\). The within group inequality is the smallest for the groups ”11 to 110” and ”1011 and over”. Such an effect is due to the low number of
elements in each group (2 per group) and the minimal difference between element values (see Table 4.1). Note that the last group has the lower within inequality than the second one, indicating that the values range contributes to $T_G$ result also. The highest within group inequality is in the ”111 to 1010” group. This group is characterised by the relatively large number of elements (7 elements) and values diversity. Therefore, the within group inequality is affected by the number of elements in a distribution and the level of variety of element values.

<table>
<thead>
<tr>
<th>Group</th>
<th>$T_G^w$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 to 10</td>
<td>0.0008270795</td>
</tr>
<tr>
<td>11 to 110</td>
<td>0.0000003820</td>
</tr>
<tr>
<td>111 to 1010</td>
<td>0.1076214157</td>
</tr>
<tr>
<td>1011 and over</td>
<td>0.0000000380</td>
</tr>
</tbody>
</table>

Mathematically, the partitioned Theil index has a fractal or self-similar structure. It can be hierarchically decomposed to almost any degree [21]. In fact, it enjoys properties similar to the Koch snowflake - a fractal that is achieved by an infinite recursive alternation of each line segment in the original triangle, and thus resulting in a mathematical curve with infinitely self-similar segments, each of which are a scaled down twin of the original [62].

However, the deeper decomposition is, the more involved actual computation becomes. There is a trade-off between decomposition and information gain. So, prior to the usage of the partitioned Theil index for a study, the following points should be considered:

- To which degree should the hierarchical data be decomposed in order to gain the most out of Theil’s partitioning ability?
- Is this information needed?

These questions, though, will not be answered in this work but rather left for a future research. The reason is due to the nature of the conducted study – to determine what kind of benefits the Gini coefficient and the Theil index can offer when applied for software examination. To achieve that the comparison of two indexes is required. Since the Gini coefficient is not decomposable, it has to be compared to the individual Theil index to avoid fallacious deductions.

Nevertheless, the thorough examination of the Theil index’s background enables us to recommend employing the partitioned Theil index to explore how exogenous factors affect structural relationships within a population. If, though, only simple aggregation oper-
ations are required, the complex computation rules and non-intuitive nature of this approach may become an unnecessary distraction.

4.2 The Causes of Inequality

Inequality analysis evaluates the effective imbalances between the amount of resources (i.e., income) received by a population share and the size of that population share [48]. This definition, though, does not include an explanation of the causes of inequality occurrence nor the reasons for shifts in a distribution. Atkinson [7] proposed that economic inequality is due to *parametric variations* - external factors that govern the relationships between income allocation and individual achievements or freedoms. Such relationships are not constant and often change [69]. Furthermore, Sen [69] states five major sources for parametric variation. These are as follows:

1. **Personal heterogeneities**: unequal dispersion of natural abilities of individuals in the population in terms of physical characteristics such as disability, illness, age, or gender;
2. **Environmental diversities**: disparity of environmental conditions in particular climatic circumstances;
3. **Variations in social climate**: difference in social climate or the nature of community relationships such as public health care, public education arrangements, or predominance or absence of crime and violence in the area;
4. **Differences in relational perspectives**: difference in commodity requirements of established patterns of behaviour between communities in relation to social conversions and customs; for example, being a relatively poor individual living in a rich environment may prevent this person from achieving his or her basic functionings in this community;
5. **Distribution within the family**: income dispersion and usage between family members.

To determine what causes inequality in software engineering, we need to translate the factors influencing parametric variations in the field of economics to equivalent software engineering concepts.

Though, Pressman [65] states that “software is a logical rather than a physical system element”. As a result, it is may be difficult to identify the characterising properties that
truly reflect the nature of software and the process used to deliver this software. This work recommends a mapping between the source of parametric variations identified in economics to corresponding software attributes (see Table 4.3). Such an interpretation incorporates the need to capture (i) size- and complexity-related aspects of software systems [85,86], (ii) programming language idioms and conventions usage [54,86], and (iii) the commonly employed software development practices [85]. Taking into account all these components may allow researchers to reason about the state of software systems, since each component has a significant influence on solution design.

This mapping, though, is based on purely quantitative software attributes and does not incorporate social metrics, which model the sociology of the programmer working on the code [37]. Social metrics capture attributes related to the name (or an anonymous id) of developers and some knowledge about their social context (e.g., their reputation, who they email, who they are co-located with, the modules they interact with, etc.). However, this developer knowledge is absent in many repositories.

Nevertheless, such a mapping between causes of inequality in the fields of economics and software engineering provides sufficient concept detail, enabling the development of a measurement model that permits collection of suitable static code metrics [54] and effective inequality-based analysis of evolving software systems [85,86].

Table 4.3: Recommended mapping between factors affecting parametric variations in economics to software engineering equivalents

<table>
<thead>
<tr>
<th>Economics</th>
<th>Software Engineering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Personal heterogeneities</td>
<td>Development tools, coding practices, and a programming language</td>
</tr>
<tr>
<td>Environmental diversities</td>
<td>System architecture and solution domain</td>
</tr>
<tr>
<td>Variations in social climate</td>
<td>Skill level (training), time pressures, and use of third-party artefacts</td>
</tr>
<tr>
<td>Differences in relational perspectives</td>
<td>Development model and development practices</td>
</tr>
<tr>
<td>Distribution within the family</td>
<td>Business reputation, responsibilities, and ownership</td>
</tr>
</tbody>
</table>

4.3 Economics Metaphor for Software Engineering

Inequality measures can be and have been successfully applied for metrics data interpretation in software engineering [54, 55, 70, 71, 85, 86, 91, 92]. However, the fields of economics, where inequality measurement originated, and software engineering are very
different. In particular, the diversities are present in the data collection principles associated with both fields.

In the field of economics data sets used for income inequality analysis typically do not contain zero values [81]. This happens for various reasons: some consider this information unimportant, saying that individuals with no income do not contribute to the economic development of a country [81]; others simply do not have access to such data because their studies are drawn from sources, such as tax records, which omit this information [29]. Regardless of the reason, individuals with zero incomes are often removed or, according to Theil [38], "vanish", from economics data sets [81], and thus some associated measures are not designed to cope with zero values sensibly [23, 38, 81].

The situation is quite different in the field of software engineering – in particular, empirical software analysis. Zero instances often occur in metrics data, sometimes having a significant influence on data dynamics (e.g., in getters or setters measures). Some might argue that this information has no value since the corresponding features are not included in a system implementation. However, software systems are products of a logical thought process [65]. Hence, the absence of a certain functionality in a class is as much of an indicator of certain design preferences as the presence of this functionality. Knowing why developers chose not to implement certain feature in a class is as valuable as understanding why other features were included. Zeros should not be ignored. Hence, prior to utilising economic measures for software engineering examination, consequences of such differences between the underlying data collection rules in the fields of economics and software engineering ought to be investigated.

To achieve that, the effects that the presence of zero values in data might have onto results of the Gini coefficient and the Theil index should be considered. Such analysis starts with a close inspection of the Theil index and the Gini coefficient computation rules.

When examining the Theil index, $T_i$, in particular, a few inconsistencies emerge. For example, an individual can contribute to the final Theil value either positively or negatively [21,38]. To be precise, if $w_i$ is an income share of an individual, $n_i$ is a population share of this individual, and $T_i$ is a contribution of this individual to the Theil result, then the general rules of contribution [21] of this individual to the Theil index are defined as in Table 4.4.

If an individual has no income (i.e., $w_i = 0$) then his or her contribution to the final Theil value should be negative, since their population share ($n_i = 1/n$) is greater than their
income share \((1/n > 0)\) (see the third rule in Table 4.4). However, the computation of the Theil index includes a logarithm as a sub-term (see Equation 3). In the case of an individual having no income, this sub-term becomes \(\log(0)\) which is undefined. Since economics data sets often do not contain zero instances, Henry Theil assumed that if an individual has zero income share then their contribution becomes zero to the Theil index result \([38, 81]\):

\[
\frac{x_i}{\mu_p} \ln \left( \frac{x_i}{\mu_p} \right) = 0, \quad \text{if } x_i = 0
\]  

(1)

Interestingly, this goes against the rules of contribution listed in Table 4.4 \([21]\). Such a mathematical inconsistency casts doubt on the Theil index result’s reliability. More importantly, when applied to software metrics interpretation in cases where the data set contains a vast amount of zero values \((i.e.,\) getters variations \([54]\)) the Theil index may distort the resulting value severely and, thus, may not be usable at all.

The Gini coefficient does not suffer from this issue. It calculates inequality using relative mean differences between asymmetric pairs of values \([66, 68]\). Therefore, if an individual has zero income, then it does not imply that this individual has no contribution towards the resulting inequality value. Also, since the Gini coefficient formula contains no sub-term that will be affected by a specific value in a distribution as shown in Equation 2, this index can be applied directly to any data set, including ones containing zeros. Therefore, if there is a need to maintain zero instances in metrics data or to study the distribution of functionalities that are not declared in software, then the Gini coefficient is a more suitable choice than the Theil index.

This statement, though, does not imply a complete dismissal of the Theil index for software analysis. Instead, to minimise zero issues and to maintain data integrity, the Theil index could be applied to data where all zero instances has been removed. Such an approach loses some valuable information about functionality distribution, and devalues the
comparison of the Gini coefficient computed on the full set to the Theil index computed on a set with eliminated zeros. Thus, for the purposes of this work we collect the Gini coefficient computed on a zero-free data set to ensure its comparability to the Theil index and the Gini coefficient computed on a full data set as means to retain information.

Still, calculating the Gini coefficient twice – on data containing and excluding zero instances – does not explicitly indicate how much information is lost when computing inequality on a zero-free data. Thus, this work investigates the effects zeros have on inequality analysis using the Gini coefficient as a utility measure, since the Theil index is unsuitable for such role due to its calculation dispute (see Equation 1).

To do so, this work introduces a new measure called the Gini Difference (GiniDif) which assesses the impact removing zeros have on the inequality value. This measure calculates a rank difference between the Gini coefficient, computed on a complete data set, and the Gini coefficient, computed on the same set with all zero instances removed. If \( A \) is a set of metric data to be analysed, \( A' \) is defined as \( A' = \{ x_a \in A | x_a > 0 \} \), and \( Gini(X) \) is the Gini coefficient function applied to any set \( X \), then GiniDif is defined as follows:

\[
    GiniDif(A) = Gini(A) - Gini(A')
\]  

Further we denote the Gini coefficient calculated on a data collection with all zero instances removed \( (Gini(A')) \) as \( zGini \). Thus, the \( zGini \) coefficient can be seen as a direct economics metaphor of inequality analysis; and GiniDif can be interpreted as the difference between software engineering and economics inequality indexes applications.

### 4.4 Summary

To asses the level of inequality in a distribution we measure it. Multiple approaches exist to achieve that. The ones described in this chapter are the Lorenz curve, the Gini coefficient, and the Theil index. The Lorenz curve present the graphical representation of the degree of inequality in a distribution, whereas the Gini coefficient and the Theil index are numerical inequality measures.

However, being able to measure inequality is only one side of inequality analysis. There is also a need in understating what causes distribution shifts. It is found that, inequality is due to parametric variations – external factors affecting the relationship between available resources and the corresponding entities holding these resources. Parametric variations
are, in turn, caused by five factors: personal heterogeneities, environmental diversities, variations in social climate, differences in relational perspectives, and distribution within the family. Each factor can be mapped to a number of software concepts to explain the occurrences of inequalities in software functionality.

Inequality analysis is originally an economic technique. This may give rise to measurement inconsistencies when applying this approach in the field of software engineering. In fact, some inequality measures are unable to process certain information (zeros in data sets) sensitively. Removing zeros, on the other hand, implies information loss. To determine the amount of information relinquished during zero elimination, a new measure was developed – the Gini Difference. It is a rank difference between the Gini coefficient computed on a data set and the Gini coefficient computed on the same set with all zero instances removed prior to calculation (a.k.a. zGini).
Chapter 5

The Behaviour of the Gini Coefficient and the Theil Index

The Gini coefficient and the Theil index capture inequality differently. The Gini coefficient enables us to reason about inequality on the population or macro level; whereas the Theil index measures inequality on the micro, individual level. But, to what degree are these indexes distinct; and by what means do they measure inequality in a distribution?

To answer these questions, this chapter studies how the Gini coefficient and the Theil index actually aggregate metrics data. Hence, a well-understood approach that would allow us to observe and understand the indexes’ behaviour is required.

For this purpose, a number of mean-invariant experiments were performed each of which is based on the Pigou-Dalton condition, also defined as the Pigou-Dalton distribution principle [22, 27, 68], that is satisfied by both indexes [22, 68]. The Pigou-Dalton principle states that if we transfer assets from a “richer” individual to a “poorer” one, without changing any other attributes, then inequality in a distribution decreases and a corresponding inequality index must assume a lower value, and vice versa. We employ this principle to study the behaviour of the Theil index and the Gini coefficient, along with an analysis of possible issues with a naïve application of inequality measures.

In Section 5.1 the Gini coefficient and the Theil index performance are examined along with some of the inter-relations when both measures are employed on an experimental data set. As a result, some of the properties of both indexes along with their drawbacks are discovered. Section 5.2 discusses what effects zero values have in inequality mea-
measurement. This section also describes an experiment which examines the Gini Difference performance when applied on a set enriched with zero instances. Finally, Section 5.3 summarises this chapter.

5.1 The Pigou-Dalton Experiment

Prior to commencing a study on the Gini coefficient and the Theil index performance, there is a need to consider a few important experiment design aspects. In particular, a careful experimental data selection is required. The Theil index, unlike the Gini coefficient, performs poorly when employed for data collections with a heavy presence of zero instances [81]. Thus, zero values should be eliminated from experiment distributions to avoid data integrity violations (for reasoning, see Chapter 4). Also, the Theil index is a size dependent measure [38], hence the affect of such a property should be analysed by introducing a few investigations with various data size.

So, three experiments were designed for populations containing 100, 1000, and 10000 numerical elements. Each experiment has an element value average of \( \mu = 100 \). The initial data collection contained \( N \) elements, where:

- \((N - 1)\) elements were assigned the value of 1, and
- the \( N \)th element had the value of \( \mu N - (N - 1) \).

Such assets allocation provides an almost perfect inequality in a population. For example, on the data collection with 100 elements, where 99 of them are set to 1 and 1 is 9901, the Gini coefficient equates to 0.9801 and the Theil index is 4.5041 (note \( \log 100 = 4.605 \)) – indicating a near total inequality in a distribution.

However, having one case is insufficient to understand how these measures capture inequality. We employ the Pigou-Dalton condition to add some dynamic changes to our test distributions. In particular, \((N - 1)\) transfers, or iterations, are performed, each of which removes \( \mu - 1 \) assets from the \( N \)th element in a distribution and adds it to the first available element with value 1. Thus, \( \mu = 100 \) is preserved and all requirements of the Pigou-Dalton principle are satisfied.

For each iteration three measures are recorded: the Gini coefficient, the Theil index, and the normalised Theil index. The charts plotting the results for all experiments are presented in Figure 5.1 and Figure 5.2.
Chapter 5. The Behaviour of the Gini Coefficient and the Theil Index

Figure 5.1: Mean-invariant transfers in a distribution with 100, 1000, and 10000 elements - the Gini coefficient vs. the Theil index

With every transfer from a rich element to a poor one, both the Gini coefficient and the Theil index are decreasing monotonically (see Figure 5.1) - an expected behaviour inferred by the Pigou-Dalton condition. The initial inspection of Figure 5.1 reveals that the graphs do not really alter between three experiments. However, a more detailed analysis indicates that although the rate at which the Theil index lessens is the same, its maximum value increases dramatically as the number of elements in a distribution increments. For example, for a population with 10,000 elements, the maximum the Theil index acquired is 9.0623 (see Figure 5.1(c)), whereas for the 100-member experiment it is 4.5041 (see Figure 5.1(a)). Yet, they both represent an almost total inequality in the corresponding distributions. This is due to Theil’s idea that as the size of a distribution increases, so should corresponding inequality index also [81].

In comparison, the Gini coefficient curves vividly demonstrate its size indifference (see Figure 5.1). Neither the growth rate nor the maximum values change as a number of members in an experiment increases. The curves connect to the same point and follow identical paths in all three experiments.
The size dependency of the Theil index influences the value of the normalised Theil index also (see Figure 5.2). When the Gini coefficient and the Theil index graphs experience no changes as the size of a distribution grows (visually), the curves for the normalised Theil index tend to have an increasing protuberance (see Figure 5.2). Such observation indicates that unlike the Gini coefficient, the normalised Theil index is population-size dependent - an inherent property from the Theil index. The normalisation function for the Theil index (see Equation 4) actually magnifies the normalised Theil index’s size-dependency. Once the Theil index value reaches 2.4, $e^{T_i}$ is changing so rapidly that the normalised Theil index reaches 1 – the value for perfect inequality – almost immediately [22].

Figure 5.2: Mean-invariant transfers in a distribution with 100, 1000, and 10000 elements - the Theil index vs. the normalised Theil index

Conceição and Galbraith [22] determined that when inequality is high within a considered population, the Theil index grows monotonically (almost linearly). In particular if the Theil index is greater than approximately 2.4, then it is a sole representation of high inequality in a respective distribution. Thus, this index experiences a distinguished sensitivity to an increasing inequality when a considered set is already unequally dispersed [22]. This range of values, however, is highly compressed by the normalisation
function. Thus, the normalised Theil index loses the Theil index’s sensitivity in the upper bounds.

But how does the normalised Theil index capture inequality? Consider Figure 5.3 showing a scatter plot between the normalised Theil index and the Gini coefficient for an experiment with 1,000 members. Here, when the Gini coefficient is just 0.5, the normalised Theil index is around 0.85. So, the Gini range between 0.5 and 1 is captured in the top 15% from the Theil range. This implies that the normalised Theil index loses its sensitivity when inequality is high; however, it has a better resolution than the Gini coefficient at the lower end.

![Figure 5.3: Scatter plot of the Gini coefficient vs. the normalised Theil index for 1,000-member experiment](image)

Still, it is not clear which index should be used for software inequality analysis and when. The experiments show that both indexes progress and capture inequality as expected. Though, the unintuitive interpretation and population-size dependency of the Theil index makes data interpretation and comparison much harder with this measure, if not impossible. The normalised Theil index is not a complete solution to these issues and also requires an additional consideration. These aspects make the Theil index and the normalised Theil index troublesome for software metrics data analysis, as the confounding effects of size may lead to resulting data distortion [32].

The experiments also show that the Gini coefficient is more stable alternative to the Theil index since it is not affected by distribution size. Nevertheless, it has been argued in the software engineering community that the Gini coefficient limits empirical analysis and the preference should be given to the Theil index as a data aggregation technique [71].

Experiments presented in this section are very different to real-life data sets. Their main
purpose is to capture the progression of both indexes between two extremes - from perfect inequality to perfect equality. Further understanding of the Gini coefficient and the Theil index performance should be gained from the real-life data analysis.

5.2 Gini, zGini, and GiniDif

In software engineering, in particular empirical software analysis, zero values are a representation of functionalities that are absent from a system implementation. This does not imply, though, that when performing software examination these values can safely ignored. On the contrary, it is valuable to analyse this data since every decision made during system’s development is a product of a logical process which is governed by certain system design properties [65]. But, if the necessity of removing zeros from analysis takes place, how much data is going to be lost by taking such approach and what are the effects a na"ïve application of inequality measures have onto result quality.

To answer these questions, an experiment has been developed where the performance of the Gini coefficient, the zGini coefficient, and the Gini Difference measures are examined. As the basis for this work, similar to the Theil index and the Gini coefficient performance study, we utilise the Pigou-Dalton condition [22, 27, 68]. However, unlike the previous experimentations in Section 5.1, zero instances are included in the data to examine their distribution patterns.

Since the Gini coefficient is a core measure for this study, which is size indifferent, one experiment containing 1,000 elements is sufficient for zero influence investigation. This experiment is organised as a series of transfers from rich to poor driven by the Pigou-Dalton condition, similar to ones in Section 5.1. The experiment is also mean-invariant with \( \mu = 1.001 \). So, if \( N = 1,000 \) is the number of elements in a population, then the analysed distribution is organised as follows:

- \((N - 1)\) elements were assigned the value of 0, and
- the \( N\)th element had the value of \( N + 1 \).

Such an allocation of values replicates a perfect inequality in a distribution. Here the maximum Gini coefficient is 0.999 and the maximum Theil index is 6.9078.

Overall, 999 or \((N - 1)\) transfers were performed each of which removes 1 from the \( N\)th or 1000th element in a distribution and adds it to the first available element with 0 assets.
For each iteration the Gini coefficient, the zGini coefficient, and the Gini Difference are recorded. Figure 5.4 presents the values changes for the Gini coefficient and the zGini coefficient only.

Both the Gini coefficient and the zGini coefficient are decreasing with each iteration. Such behaviour is expected as per the Pigou-Dalton principle. Though, both indexes evolve at separate rates. The Gini coefficient produces the curve almost identical to the one shown in the previous section (see Figure 5.1(b)). Whereas the zGini coefficient chart (Figure 5.4) presents contradicting information. At first zGini experiences a rapid growth, but once it reaches its maximum (0.9378), it starts to decrease linearly until zGini attains the value of 0. The linear decrease can be explained by the correspondence to the Pigou-Dalton condition and zero instances exclusion. The rapid growth of the zGini coefficient, on the other hand, requires a deeper analysis of the Gini coefficient characteristics.

Interestingly, Deltas [30] determined that the value of the Gini coefficient is highly biased toward zero when applied on small distribution samples. Since the zGini coefficient captures inequality on the set where all zero instances has been removed prior to analysis, the data collection size becomes equal to a corresponding iteration index. For example, for the first iteration the considered data collection is of a form “1001”; for the second iteration it is “1000,1”; for the third “999,1,1”; and so on. So, for these cases the number of elements in a considered data collection are insufficient for the Gini coefficient to perform as intended since it is downwardly-biased in small populations [30]. Thus, as the sample size increases, so does the zGini coefficient up to a point where the size of a considered set becomes sufficient to satisfactorily reduce the bias [30].
To study the effects zero values have on inequality measurement the Gini Difference is employed. This measures is a distance between the Gini coefficient and the zGini coefficient. So, its changes must reflect ones of those two indexes. To examine that, the Gini Difference behaviour for the Pigou-Dalton experiment with 1000 members is presented in Figure 5.5.

![Figure 5.5: The Gini Difference scatter plot for 1000-member experiment](image)

The Gini Difference transpires as predicted – by echoing the distances between the Gini coefficient and the zGini coefficient values. The disparity is large at the start and tends to decrease for about 30 iterations due to the biased zGini value for small data samples. Once the zGini coefficient is stabilised (GiniDif = 0.06023), the Gini Difference increases up to iteration 500 and diminishes to 0 towards the end of the experiment. Such behaviour is explained by the change dynamics of both the Gini coefficient and the zGini coefficient as shown in Figure 5.4. The Gini coefficient experiences an exponential decrease, whereas the zGini coefficient graph lessens almost linearly. Thus, the Gini Difference reflects the distances between the Gini and the zGini curves which is presented in the corresponding curve bulge cf. Figure 5.5.

The experiment presented in this section is distanced from real-life data. Thus, to study the causality zero instances have on inequality analysis of software systems, we are required to apply these measures to unchanged metrics data collections, which is discussed in Chapter 7.
5.3 Summary

To summarise, a number of experiments were performed to determine typical behaviour patterns of the Gini coefficient and the Theil index. Each experiment was designed to replicate the inequality decrease from perfect inequality to perfect equality using the Pigou-Dalton condition as the main rule governing overall distribution.

The experiments showed that the Theil index is a size-dependent measure with an invariant progression rate. This implies that once plotted over iteration its curves have an identical shape but varied maximum values. Interestingly, the normalised Theil index inherited this size-dependency property. Although the normalised Theil index always ranges between 0 and 1, inclusively, its curves tend to bend to the right as the size of an experimental set increases.

The Gini coefficient, on the other hand, ranges in the same interval \([0, 1)\) and grows at the same rate regardless of data size. When compared to the normalised Theil index, the Gini coefficient tends to be more sensitive in the upper bounds of inequality, whereas the normalised Theil index captures inequality with a better precision when it is still relatively low.

The experiments also showed that the Gini Difference captures the rank differences between the Gini coefficient and the \(z\)Gini coefficient as intended. More on this topic and on the evaluation of inter-dependencies between the Theil index and the Gini coefficient will be discussed in the following chapter.
Chapter 6

Analysis of Inequality Indexes on Helix Repository

Thus far, this work has presented an analytic evaluation of the Gini coefficient and the Theil index of their suitability for software studies. Although such an approach is valuable to build an understanding of how these indexes capture inequality, it does not reflect the actual indexes’ behaviour when applied to software analysis. To understand that, both inequality measures can be employed in the analysis of real-world software systems.

The chapter is organised as follows: Section 6.1 presents the motivation for analysing the normalised Theil index and the zGini coefficient only and lists their associated typical ranges when computed on the Helix repository. Section 6.2 presents correlation analysis between these two inequality measures. Section 6.3 lists common behaviour patterns (relationship trends) of both indexes with an associated analysis of the frequency of occurrence of each behaviour trend per metric. This section also lists major relationship trends between indexes observed when studying the Helix repository. It is followed by Section 6.4 which presents the differences in the way both indexes capture information, along with a hypothesis regarding the occurrences of identified relationship patterns. It also provides guidelines indicating how and when the indexes should be used in order to retrieve more information. Finally, this chapter is summarised in Section 6.5.
6.1 Typical Ranges

We begin the evaluation of the Gini coefficient and the Theil index with an analysis of their typical ranges when computed on the selected data and metrics collection listed in Section 2.3. Table 6.1 lists all metrics and their ranges for the Gini coefficient, the zGini coefficient, the Theil index, and the normalised Theil index computed on the Helix repository.\(^1\)

**Table 6.1**: Values ranges of the Gini coefficient, the zGini coefficient, the Theil index, and the normalised Theil index

<table>
<thead>
<tr>
<th>Measure</th>
<th>Metric</th>
<th>Gini</th>
<th>zGini</th>
<th>Theil</th>
<th>Normalised Theil</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Min</td>
<td>Max</td>
<td>Min</td>
<td>Max</td>
</tr>
<tr>
<td>ck</td>
<td>cbo</td>
<td>0.415</td>
<td>0.675</td>
<td>0.374</td>
<td>0.579</td>
</tr>
<tr>
<td>ck</td>
<td>rfc</td>
<td>0.497</td>
<td>0.729</td>
<td>0.467</td>
<td>0.704</td>
</tr>
<tr>
<td>ck</td>
<td>wmc</td>
<td>0.545</td>
<td>0.911</td>
<td>0.395</td>
<td>0.926</td>
</tr>
<tr>
<td>degrees</td>
<td>indegree</td>
<td>0.552</td>
<td>0.796</td>
<td>0.455</td>
<td>0.795</td>
</tr>
<tr>
<td>degrees</td>
<td>outdegree</td>
<td>0.415</td>
<td>0.675</td>
<td>0.374</td>
<td>0.579</td>
</tr>
<tr>
<td>gssummary</td>
<td>getters</td>
<td>0.655</td>
<td>0.946</td>
<td>0.274</td>
<td>0.643</td>
</tr>
<tr>
<td>gssummary</td>
<td>setters</td>
<td>0.752</td>
<td>0.981</td>
<td>0.251</td>
<td>0.597</td>
</tr>
<tr>
<td>nof</td>
<td>noa</td>
<td>0.591</td>
<td>0.902</td>
<td>0.324</td>
<td>0.617</td>
</tr>
<tr>
<td>nom</td>
<td>nom</td>
<td>0.545</td>
<td>0.785</td>
<td>0.427</td>
<td>0.723</td>
</tr>
</tbody>
</table>

Overall, removing zero instances during analysis causes inequality to drop. This is evident from the differences between the Gini coefficient and the zGini coefficient ranges in Table 6.1. Thus, an underestimated value for population-level inequality is obtained when eliminating zeros from data. This effect is stronger for measures with a profuse amount of zero instances, for example *setters* or *getters* metrics, and almost inconspicuous for measures containing a few zeros (e.g., *rfc*). The observation is also supported by graphical representation of the Gini coefficient and the zGini coefficient ranges for *setters* and *rfc* metrics as presented in Figure 6.1.

In addition, the data from Table 6.1 indicates that, in general, the Theil index does not experience extremely high values. None of the considered metric ranges exceed the critical threshold of 2.4, discussed in Chapter 5. Nevertheless, there are exceptions to this observation. For instance, the Theil index value for the *wmc* metric calculated on *jchempaint-3.1.2* is 4.557, which is the largest Theil result in the sample. In fact, out of 11,745 calculated values, consisting of 9 measures calculated on 1,305 releases, only 109 of them are over the threshold value, which is less than 1%. Such results may not be difficult to interpret, since they imply a presence of machine-generated code causing high

---

\(^1\)The range values were defined using the interquartile range statistic due to the presence of extreme outliers which caused severe data distortion.
inequality due to its convoluted nature and inefficiency [26]. It is, though, unclear how to effectively compare such data between systems since the computed inequality level is highly reliant on data size. Thus, the Theil index normalisation is a necessary factor for software inequality analysis.

Table 6.1 also shows that the normalised Theil index tends to be lower than the zGini coefficient and its values are more dispersed. For example, the minimum interquartile range of the getters metric for both the normalised Theil index and the zGini coefficient are 0.274 and 0.130 respectively, with maximum values 0.643 and 0.556. Thus, the zGini coefficient experiences slightly higher values than the normalised Theil index. The normalised
Theil index has the 0.426 range, whereas zGini is distributed over 0.369, indicating that the prior index has a wider values spread. The observation is true for all considered measures.

### 6.2 Correlation Analysis

The typical range analysis lacks the ability to provide sufficient evidence for validating the theories studied in this work, which are (i) the Gini coefficient and the Theil index capture inequality from different perspectives and (ii) the combined usage of both of these indexes may offer more information than each measure used individually. Thus, we proceed with evaluation of behavioural patterns of both indexes using evolutionary data collected from the Helix repository.

While the Gini coefficient is computed on data including zero instances, the Theil index is calculated on a set of strictly positive values ($\{x \in X | x > 0\}$). This arrangement risks the occurrence of an ecological fallacy [63] – a comparison between two measures calculated on distinct data sets. In addition, the Theil index is not really comparable to the Gini coefficient in its original form, since this index’s value is highly dependent on considered data size [38]. Therefore, to maintain data integrity [63] and to ensure the indexes’ comparability, the zGini coefficient and the normalised Theil index are the only studied indexes in this investigation.

One of the first questions arising here is whether there is a relationship between these two measures. To answer this question, Spearman’s rank correlation coefficient [61] is used to analyse the evolution behaviours of both indexes. The idea is to define in how many cases both indexes are not related. For the purpose of this investigation, we are interested in strong dependencies only. Hence, using the definition of Spearman rank correlation coefficient, listed in [61], the subjective threshold value of 0.6 was selected, indicating that the correlation lower than 0.6 implies weakened relationship.

Overall, out of 378 cases (9 measures with 42 systems each) only 22 experience a correlation value smaller than 0.6, with just four of them below 0 – an indication that the indexes are negatively correlated (while one index tends to grow, the second one decreases) [93]. Thus, in the considered data collection around 5.8% of observations experience weak or no relationship. Nevertheless, when considering systems with low correlation each can be explained. For instance, the freemarker correlation value for the cbo measure is
-0.3137 which implies the lack of linear relationship between the two inequality indexes. However, the scatter plot between the zGini coefficient and the normalised Theil index depicting this observation (see Figure 6.2) demonstrates that this weak relationship is due to a quite small range for both indexes.

![Figure 6.2: The zGini Coefficient and the normalised Theil index scatter plot for freemaker system on cbo metric](image)

Another example of a low correlation result is the outdegree correlation calculated on the pmd system. The associated value is -0.8. Still, when studying the scatter plot of both the zGini coefficient and the normalised Theil index against RSN, depicted in Figure 6.3, it is evident that once two curves intersected they begin to develop at a similar rate. In fact, the correlation value after RSN 7 increases dramatically to 0.5786; after RSN 8 it is 0.6890; after RSN 9 it becomes 0.8058.
For 22 cases of low correlation between the normalised Theil index and the zGini coefficient three main causes of such behaviour were distinguished, which are (i) too small a value range, (ii) outlier occurrences (often causing a slight relationship weakening), and (iii) the normalised Theil and the zGini curve intersections. These events are rather exceptions than rules, however they are sufficient to prevent the establishment of a monotonic relationship between the normalised Theil index and the zGini coefficient. Thus, the existence of low correlation values should not be ignored.

### 6.3 Indexes Behaviour

"Correlation is no proof of causation" [39]. Thus, the existence of a strong linear relationship between the zGini coefficient and the Theil index does not imply that these indexes capture inequality in the same way. To define the major behaviour patterns for both indexes and their relationship to each other a table is presented, which lists the main trends among and between the zGini coefficient and the normalised Theil index on a per system basis (see Table 6.4). In this table, the symbolic representation of indexes’ behaviour patterns is used which is defined in Table 6.2 and Table 6.3.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>↑</td>
<td>Apex</td>
<td>Indicates the presence of a sudden increase in index values with a consequent restoration of these values to a level aligned with the original ones.</td>
</tr>
<tr>
<td>↓</td>
<td>Drop</td>
<td>Indicates the presence of a sudden drop in index values with a consequent restoration of these values to a level aligned with the original ones.</td>
</tr>
<tr>
<td>→</td>
<td>Straight</td>
<td>Indicates that the normalised Theil index and the zGini coefficient values do not experience any significant changes as a system evolves.</td>
</tr>
<tr>
<td>←</td>
<td>Step up</td>
<td>Indicates the presence of a sudden increase in index values without a consequent restoration of these values to a previous level.</td>
</tr>
<tr>
<td>→</td>
<td>Step down</td>
<td>Indicates the presence of a sudden drop in index values without a consequent restoration of these values to a previous level.</td>
</tr>
<tr>
<td>/</td>
<td>Up</td>
<td>Indicates that the normalised Theil index and the zGini coefficient values gradually increase as a system evolves.</td>
</tr>
<tr>
<td>\</td>
<td>Down</td>
<td>Indicates that the normalised Theil index and the zGini coefficient values gradually decrease as a system evolves.</td>
</tr>
</tbody>
</table>
Table 6.3: Symbolic representation of "between indexes" behaviour patterns (b/w column in Table 6.4)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>≡</td>
<td>Parallel</td>
<td>Indicates that the normalised Theil index and the zGini coefficient evolve in parallel.</td>
</tr>
<tr>
<td>⊳</td>
<td>Divergent</td>
<td>Indicates that either the normalised Theil index or the zGini coefficient vary during system evolution, preventing the establishment of a parallel relationship.</td>
</tr>
<tr>
<td>⌁</td>
<td>Crossing</td>
<td>Indicates that the normalised Theil index and the zGini coefficient evolve at different rates and their associated evolution trends cross at a certain point(s) in system’s history.</td>
</tr>
</tbody>
</table>

For the purpose of this investigation the strictness with which the zGini coefficient and the normalised Theil index are considered to follow a certain behaviour pattern is of a little importance. The aim is to define key behaviour trends of both indexes. The proportion of occurrences of every behaviour pattern per measure is presented in Table 6.5.

On average, across all systems and metrics the most followed behaviour pattern by both the normalised Theil index and the zGini coefficient is "step up" (24.1%), indicating a relatively rapid increase in inequality index values during system evolution. However, such performance is not preponderant in the Helix repository. There are other trends which typically have values close to that associated with the $\uparrow$ symbol. These include:

- the "up" ($\searrow$) trend with an average proportion of 19.9%,
- the "apex" trend ($\nearrow$) with an average of 17.9%,
- the "straight" trend ($\rightarrow$) with an average of 15.3%, and
- the "drop" trend ($\downarrow$) with an average of 15.1%.

Examples of each behaviour pattern are presented in Figure 6.7.

For these five behaviour trends ($\leftarrow$, $\leftarrow\rightarrow$, $\rightarrow\leftarrow$, $\rightarrow$), the proportions of each is relatively evenly distributed amongst metrics (see Table 6.5). There are, though, outliers. For instance, the "step up" behaviour pattern for the indegree metric experiences the highest occurrence rate along all considered metrics and trends (47.6%) which is well above the corresponding average listed in Table 6.5. The indegree metric, however, has another outlier – there is no "straight" behaviour pattern observed when examining inequality evolution for this metric. Such findings, though, may be explained by the nature of the indegree metric: it captures the popularity of a class. The class’s popularity tends to increase over time [86], however, since it is not a monotonic function [86], the value growth
Another intriguing observation here is that both getters and setters metrics a relatively low proportion of occurrence for the "drop" and "apex" behaviour patterns. The getters metric has a 9.5% rate for the "apex" trend and 7.1% for the "drop" trend. The setters metric experiences a 4.8% and 9.5% rate for the "apex" and "drop" behaviours respectively. In both cases these correspond to the lowest proportions for two considered trends. This observation may be explained by the way developers employ getters and setters functionality – they introduce such methods responsibly within clearly defined decision frames [54].

So far the considered trends (→, ↓, └, ┐, and /) can be characterised as ones where inequality is increasing or experiencing almost no change as a system evolves. Even though the “drop” trend indicates a decrease in inequality value, it is associated with a consequent restoration of inequality to a previous level. Such an observation can be taken as supporting Lehman’s Laws of Software Evolution [44] presented in Table 2.1. In particular, it upholds the Conservation of Familiarity (the fifth law) and Continuing Growth (the sixth law).

However, there are exceptions to this observation. Two index behaviour trends – "step down” and "down” – are also displayed by inequality indexes, although not as often as others. The maximum proportion of occurrences for the "step down" trend is 14.3%, which is present when studying the getters metric. As for the "down" trend, it is very uncommon with a maximum of only 7.1% (wmc metric). Overall, using information presented in Table 6.5 it can be deduced that both behaviour patterns are relatively abnormal.

The "step down” pattern, although uncommon, is present in each measure (see Table 6.5). It is consistently captured by the zGini coefficient and the normalised Theil index across all considered metrics. This behaviour is characterised by a sudden drop in index values during system evolution, which may indicate a restructuring event, though there is no documentation available supporting this statement. An example of such behaviour pattern is shown in Figure 6.4.

The "down” behaviour trend, as opposed to the "step down” one, is characterised by a more gradual decrease of values over time (i.e., one listed in Figure 6.5). This behaviour is much less frequently followed than others. Out of nine considered metrics, five of them – rfc, indegree, getters, setters, and noa – have no systems where the normalised Theil index and the zGini coefficient experience progressive decrease over time. The rest of metrics have maximum of only three systems following such trend (see Table 6.5).
So far, this section has covered overall index behaviour patterns. However, we are also interested in between index relationships. To analyse such trends, Table 6.4 presents between index behaviour patterns also and Table 6.5 lists a summary of their occurrences across metrics.

The most prevalent relationship between the normalised Theil index and the zGini coefficient is "parallel" (▁▁▁▁), indicating that two indexes evolve at the same or a very similar rate. This trend is present in average of 67.7% of observations. Interestingly, each measure has approximately 80% occurrences of this relationship except in three cases:
CHAPTER 6. ANALYSIS OF INEQUALITY INDEXES ON HELIX REPOSITORY

- the wmc metric, where 50% of systems present such a relationship between the two indexes,
- the indegree metric where only 11.9% of systems experience this relationship,
- the nom metric, where 52.4% of systems evidence the "parallel" relationship.

A representative sample for this behaviour trend coincides with an example of the "up" behaviour pattern, presented in Figure 6.7(b). There, both indexes experience continuous growth over time characterised by a close growth rate.

The normalised Theil index and the zGini coefficient, though, do not always evolve in parallel. In some cases the indexes’ values deviate over time – a relationship which is defined as "divergent" (
\[ \neq \]) in Table 6.3. On average, 19.3% of observations experience such a relationship. However, there is a strong outlier: evolution changes for the nom metric demonstrate the highest proportion of divergent trends occurrences, which equates to 40.5%. The "divergent" relationship is characterised by the presence of an overall similar index development pattern with a clear difference in the rate at which either index evolves. An example of such behaviour is presented in Figure 6.6, which illustrates evolutionary changes of both indexes computed on kolmafia for the nom metric. In this figure, both the normalised Theil index and the zGini coefficient curves pursue similar behaviour patterns. However, the normalised Theil index experiences much more intense value fluctuations than the zGini coefficient.

Another non-parallel pattern between the considered inequality indexes is "crossing" (\[ \times \]). An example of this relationship has already been illustrated in Figure 6.7(c) and Figure 6.7(e). The main attribute of this trend is a crossing between the normalised Theil index and the zGini coefficient curves. On average, the indexes exhibit such behaviour in 13.0% of observations as listed in Table 6.5. Notably, no systems experience such a growth pattern when inequality is computed on the noa metric, and the largest proportion of occurrences of the the "crossing" relationship is seen when examining the indegree metric, which is as high as 52.4%.

Overall, the "divergent" and "crossing" relationships are not as frequently observed as "parallel", but are still present. Such a finding raises the question whether these two indexes are, in fact, similar or not, and why these behaviour trends occur. In the observed cases, it is the normalised Theil index that reacts more sensitively to an unknown evolution event (i.e., in Figure 6.6). It might be the case that these two indexes are affected differently by some emergent properties during software evolution, though no evidence
for this statement has been presented so far. This hypothesis is going to be investigated in the next section of this chapter.

Figure 6.6: "Divergent" relationship example (kolmafia evolution changes for nom metric)
Chapter 6. Analysis of Inequality Indexes on Helix Repository

(a) "Straight" behaviour example (freemarker evolution changes for cbo metric)
(b) "Up" behaviour example (saxon evolution changes for rfc metric)
(c) "Apex" behaviour example (activemq evolution changes for wmc metric)
(d) "Drop" behaviour example (xalan evolution changes for outdegree metric)
(e) "Step up" behaviour example (tapestry evolution changes for cbo metric)

Figure 6.7: Predominate behaviour types of the zGini coefficient and the normalised Theil index
Table 6.4: The Gini coefficient and the normalised Theil index behavioural trends

<table>
<thead>
<tr>
<th>No</th>
<th>System</th>
<th>Measure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>cbo</td>
</tr>
<tr>
<td></td>
<td></td>
<td>all</td>
</tr>
<tr>
<td>1</td>
<td>acegi</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>activebpel</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>activemq</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ant</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>axis</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>azureus</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>castor</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>checkstyle</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>cocoon</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>columba</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>compass</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>findbugs</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>flow4j</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>freemarker</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>groovy</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>hibernate</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>ibatis</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>itext</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>jabref</td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>No</th>
<th>System</th>
<th>cbo</th>
<th>rfc</th>
<th>wmc</th>
<th>indegree</th>
<th>outdegree</th>
<th>getters</th>
<th>setters</th>
<th>noa</th>
<th>nom</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>all</td>
<td>b/w</td>
<td>all</td>
<td>b/w</td>
<td>all</td>
<td>b/w</td>
<td>all</td>
<td>b/w</td>
<td>all</td>
</tr>
<tr>
<td>20</td>
<td>jasperreports</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>jchempaint</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>jena</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>jfreechart</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>jmeter</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>jruby</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>jung</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>kolmafia</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>lucene</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>pmd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>proguard</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>rssowl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>saxon</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>soapui</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>spring</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>stripes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>struts</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>tapestry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>webwork</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
### Table 6.4 – Continued from previous page

<table>
<thead>
<tr>
<th>No</th>
<th>System</th>
<th>cbo_all</th>
<th>rfc_all</th>
<th>wmc_all</th>
<th>indegree_all</th>
<th>outdegree_all</th>
<th>getters_all</th>
<th>setters_all</th>
<th>noa_all</th>
<th>nom_all</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>b/w</td>
<td>b/w</td>
<td>b/w</td>
<td>b/w</td>
<td>b/w</td>
<td>b/w</td>
<td>b/w</td>
<td>b/w</td>
<td>b/w</td>
</tr>
<tr>
<td>39</td>
<td>wicket</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>xalan</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>xerces</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>xwork</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 6.5: Table 6.4 summary

<table>
<thead>
<tr>
<th>Measure</th>
<th>Overall</th>
<th>Between measures</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>⊲ ⊲ ⊲ ⊲</td>
<td>⊳ ⊳ ⊳ ⊳</td>
</tr>
<tr>
<td>cbo</td>
<td>21.4</td>
<td>19.0</td>
</tr>
<tr>
<td>rfc</td>
<td>21.4</td>
<td>9.5</td>
</tr>
<tr>
<td>wmc</td>
<td>23.8</td>
<td>19.1</td>
</tr>
<tr>
<td>indegree</td>
<td>23.8</td>
<td>7.1</td>
</tr>
<tr>
<td>outdegree</td>
<td>16.7</td>
<td>23.8</td>
</tr>
<tr>
<td>getters</td>
<td>9.5</td>
<td>7.1</td>
</tr>
<tr>
<td>setters</td>
<td>4.8</td>
<td>9.5</td>
</tr>
<tr>
<td>noa</td>
<td>21.4</td>
<td>16.7</td>
</tr>
<tr>
<td>nom</td>
<td>19.0</td>
<td>23.8</td>
</tr>
<tr>
<td>Mean</td>
<td>17.9</td>
<td>15.1</td>
</tr>
</tbody>
</table>
6.4 Inequality Indexes Application

Even in cases where the zGini coefficient and the normalised Theil index diverge or cross as a system evolves, there is always a clear behaviour pattern present defining overall inequality changes in the system. This observation implies that inequality analysis yields the same results regardless of the index applied. However, the existence and popularity of various inequality measures in the field of economics [24], along with the observations regarding their varied performance in a controlled environment (presented in Chapter 5), signify the importance of the usage of various inequality measures to achieve certain goals. So, what kind of benefits does each studied inequality index offer to software analysis?

Consider the relationship trends between the normalised Theil index and the zGini coefficient discussed in Section 6.3 – "divergent", "parallel", and "crossing". The "parallel" behaviour pattern indicates that two indexes evolve at the same rate. The "divergent" and "crossing" relationships state that two indexes experience a somewhat-different-from-parallel relationship where they either diverge or cross over time.

Thus, to identify the exact benefits of each considered index, an investigation is conducted that analyses between indexes relationship types. The trends where the zGini coefficient and the normalised Theil index deviate – "divergent" and "crossing" – are of the most value here, since they indicate that two indexes evolve separately. In addition, even thought the "parallel" relationship lacks the descriptive power for such an investigation, it was also included in order to determine how it differs from other relationship types.

For this investigation an overall examination of the differences between the normalised Theil index and the zGini coefficient was performed. To identify what affects inequality indexes a number of histograms were developed depicting each metric distribution associated with every release in the Helix repository. The investigation groups findings by the relationship type.

Due to the large number of observations, it is difficult to present all findings in this work. Thus, to illustrate the corresponding discoveries two cases of each relationship were chosen. The investigation is supported by a number of histograms depicting metric values distribution for selected cases listed in Appendix B. Not all associated histograms are presented in this work, meaning some releases’ information is omitted due to space constraints. However, these histograms depict the overall behaviour and major distribution
shifts effectively. The representative samples of each behaviour patterns, used for this investigation, are as follows:

- "Parallel" relationship:
  - \textit{cbo} metric calculated on \textit{activebpel} system,
  - \textit{outdegree} metric calculated on \textit{ant} system,

- "Divergent" relationship:
  - \textit{rfc} metric calculated on \textit{jchempaint} system,
  - \textit{indegree} metric calculated on \textit{findbugs} system,

- "Crossing" relationship:
  - \textit{cbo} metric calculated on \textit{tapestry} system,
  - \textit{outdegree} metric calculated on \textit{pmd} system.

Two samples have been chosen to evaluate this the "parallel" relationship (see Figure 6.8): inequality changes of the \textit{cbo} metric for \textit{activebpel} (Figure 6.8(a)) and inequality changes of the \textit{outdegree} metric computed on \textit{ant}(Figure 6.8(b)).

The sample shown in Figure 6.8(a) experiences almost no change over system evolution, though some slight fluctuation is present. The associated histogram collection is shown in Figure B.3 which presents the graphical representation of the \textit{cbo} metric distribution for 5 out of 16 releases of the \textit{activebpel} system available in the Helix repository. The histograms indicate that, overall, the \textit{cbo} metric for \textit{activebpel} has a consistent profile across all available releases of this software, with a tendency for slightly wider values spread and lower density (frequency divided by the width of an interval) over time. However, no dramatic differences between the histograms are observed.

Inequality of the \textit{outdegree} metric, calculated on \textit{ant}, increases over time and both indexes evolve at the same rate. In this case when evaluating associated histograms it can be seen that there is a difference in associated distributions, but the change is very gradual. The main observation is that the \textit{outdegree} values tend to be more skewed to the right as the system evolves and no significant outliers emerge, presenting an overall equable behaviour.

Thus, the "parallel" behaviour trend analysis for the normalised Theil index and the \textit{zGini} coefficient shows that where both indexes evolve at the same or a very similar rate no rapid distribution shifts are present and no extreme data outliers emerge. The investigation moves to the "divergent" relationship analysis to discover how it differs from the "parallel" one.
CHAPTER 6. ANALYSIS OF INEQUALITY INDEXES ON HELIX REPOSITORY

The first representative sample for the "divergent" relationship is evolution changes of both inequality indexes for the rfc metric computed on jchempaint shown in Figure 6.9(a). Here, the zGini coefficient and the normalised Theil index evolve at the same rate until RSN 22, after which the normalised Theil index appears to be more sensitive to a certain event than the zGini coefficient. The diversion between the two indexes’ evolution patterns, which happens between RSN 22 and RSN 23, is characterised by almost changeless evolution of the zGini coefficient and a significant increase of the normalised Theil index. When examining the histograms of these two releases, presented in Figure B.4, they show that as the system evolves some changes are introduced to the overall system’s profile which is mostly characterised by a change in density. However, the profiles change dramatically between RSN 22 and RSN 23 which corresponds to a relatively significant
density decrease from 0.06 to 0.015 and an emerged utmost outlier. Both events had no predisposition and possibly play the role of an indicator to a restructuring event in RSN 23.

Another example of the ”divergent” relationship between the normalised Theil index and the zGini coefficient is indegree metric inequality analysis for the findbugs system presented in Figure 6.9(b). In this case, both indexes evolve similarly up to RSN 14, with a very slight difference from RSN 1 to RSN 3. The major deviation from such an evolution trend of the two inequality measures happens from RSN 14 to RSN 15, where the normalised Theil index reaches almost the same value as the zGini coefficient. After RSN 15 the indexes appear to evolve at the same rate, with the normalised Theil index retaining its position in relation to the zGini coefficient. This may indicate that the property that emerged at RSN 15 holds for all available consequent releases. The histogram study for the indegree metric computed on the findbugs system (see Figure B.2) shows that the metric values appear to spread slightly wider from RSN 1 to RSN 3. Then from RSN 3 to RSN 14, inclusively, the profile experiences a very slight change mostly associated with density fluctuations. RSN 15, though, presents an absolutely different distribution to the previous releases. Here, the relative distance between the histogram intervals is twice as large as that of RSN 14, and the overall density also decreases notably. A similar profile persists for the rest of the releases, confirming that the potential restructuring event in RSN 15 is maintained in the remaining versions.

The ”crossing” relationship is now considered. The selected samples for this study are the tapestry inequality evolution for the cbo metric, shown in Figure 6.10(a), and the pmd inequality evolution of the outdegree metric, presented in Figure 6.10(b).

The tapestry system experiences an interesting inequality behaviour over time (see Figure 6.10(a)). The normalised Theil index and the zGini coefficient differ significantly as the system evolves. The major deviation here, however, is associated with just one event emerging in RSN 13, where the normalised Theil index experiences a significant increase. After RSN 13, the normalised Theil index maintains a similar inequality level with few fluctuations, which eventually causes this index to match the zGini coefficient. Analysis of associated histograms, presented in Figure B.5, indicates the occurrence of a major restructuring event in RSN 13. This is due to a strong outlier arising in RSN 13 and a large density drop from approximately 0.4 in RSN 12 to 0.15 in RSN 13. A similar distribution profile persists in the remaining tapestry releases with the outlier getting more pronounced over time. The zGini coefficient was not completely indifferent to this
potential restructuring activity, however its response is much less severe than that of the normalised Theil index.

Another interesting case is the evolution of inequality measures for the *outdegree* metric computed on *pmd* (see Figure 6.10(b)). In this example, the normalised Theil index starts off higher than both the zGini coefficient and, more surprisingly, the Gini coefficient. It experiences a rapid decrease from RSN 1 to RSN 12, where it reaches its minimum. From RSN 13 and further, though, the normalised Theil index follows a very similar behaviour trend to that of the zGini coefficient, with a slight variation between RSN 19 and 27. When cross referencing this cognizance to associated histograms, shown in Figure B.6, it is observed that the overall distribution pattern of the *outdegree* metric is stable, with
To answer these questions, a number of classes per pmd release were examined, listed in Table 6.6. The initial 12 releases of the pmd system are characterised by an increase in the number of classes per version. In the first couple of releases, the number of classes is less than 150, indicating that the presence of a strong outlier would affect inequality values dramatically if a considered index is sensitive to the size of a data set (i.e. the normalised Theil index). From RSN 7 to RSN 8, pmd enlarges by almost 100 classes, which coincides with the drop of the normalised Theil index values presented in Figure 6.10(b). The most
prominent normalised Theil index decrease, however, happens between RSN 10 and RSN 12 where the pmd system changes in size from 329 classes to 617 – a major extension. At RSN 13, though, pmd consists of 295 classes only, which is reflected by the increased normalised Theil index.

Table 6.6: Number of classes per release for pmd system

<table>
<thead>
<tr>
<th>RSN</th>
<th>Version</th>
<th>No of classes</th>
<th>RSN</th>
<th>Version</th>
<th>No of classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pmd-0.1</td>
<td>130</td>
<td>22</td>
<td>pmd-2.1</td>
<td>397</td>
</tr>
<tr>
<td>2</td>
<td>pmd-0.2</td>
<td>131</td>
<td>23</td>
<td>pmd-2.2</td>
<td>404</td>
</tr>
<tr>
<td>3</td>
<td>pmd-0.3</td>
<td>133</td>
<td>24</td>
<td>pmd-2.3</td>
<td>404</td>
</tr>
<tr>
<td>4</td>
<td>pmd-0.5</td>
<td>151</td>
<td>25</td>
<td>pmd-3.0</td>
<td>436</td>
</tr>
<tr>
<td>5</td>
<td>pmd-0.7</td>
<td>159</td>
<td>26</td>
<td>pmd-3.1</td>
<td>439</td>
</tr>
<tr>
<td>6</td>
<td>pmd-0.8</td>
<td>170</td>
<td>27</td>
<td>pmd-3.2</td>
<td>444</td>
</tr>
<tr>
<td>7</td>
<td>pmd-0.9</td>
<td>190</td>
<td>28</td>
<td>pmd-3.3</td>
<td>451</td>
</tr>
<tr>
<td>8</td>
<td>pmd-1.0rc1</td>
<td>289</td>
<td>29</td>
<td>pmd-3.4</td>
<td>454</td>
</tr>
<tr>
<td>9</td>
<td>pmd-1.0rc2</td>
<td>299</td>
<td>30</td>
<td>pmd-3.5</td>
<td>459</td>
</tr>
<tr>
<td>10</td>
<td>pmd-1.0rc3</td>
<td>329</td>
<td>31</td>
<td>pmd-3.6</td>
<td>537</td>
</tr>
<tr>
<td>11</td>
<td>pmd-1.1</td>
<td>357</td>
<td>32</td>
<td>pmd-3.7</td>
<td>531</td>
</tr>
<tr>
<td>12</td>
<td>pmd-1.2</td>
<td>617</td>
<td>33</td>
<td>pmd-3.8</td>
<td>587</td>
</tr>
<tr>
<td>13</td>
<td>pmd-1.3</td>
<td>295</td>
<td>34</td>
<td>pmd-3.9</td>
<td>635</td>
</tr>
<tr>
<td>14</td>
<td>pmd-1.4</td>
<td>323</td>
<td>35</td>
<td>pmd-4.0</td>
<td>664</td>
</tr>
<tr>
<td>15</td>
<td>pmd-1.5</td>
<td>328</td>
<td>36</td>
<td>pmd-4.1</td>
<td>676</td>
</tr>
<tr>
<td>16</td>
<td>pmd-1.6</td>
<td>329</td>
<td>37</td>
<td>pmd-4.2</td>
<td>721</td>
</tr>
<tr>
<td>17</td>
<td>pmd-1.7</td>
<td>333</td>
<td>38</td>
<td>pmd-4.2.1</td>
<td>721</td>
</tr>
<tr>
<td>18</td>
<td>pmd-1.8</td>
<td>338</td>
<td>39</td>
<td>pmd-4.2.2</td>
<td>721</td>
</tr>
<tr>
<td>19</td>
<td>pmd-1.9</td>
<td>340</td>
<td>40</td>
<td>pmd-4.2.3</td>
<td>726</td>
</tr>
<tr>
<td>20</td>
<td>pmd-2.0</td>
<td>341</td>
<td>41</td>
<td>pmd-4.2.4</td>
<td>727</td>
</tr>
<tr>
<td>21</td>
<td>pmd-2.1</td>
<td>388</td>
<td>42</td>
<td>pmd-4.2.5</td>
<td>727</td>
</tr>
</tbody>
</table>

The zGini coefficient is not completely insensitive to the changes taking place during pmd evolution. It is also reflecting the major distribution shifts taking place between RSN 10 and RSN 13 of the pmd system. However, it is capturing information in a more subtle way than the normalised Theil index, focusing on the overall (i.e. architectural) properties of a system rather than the structural (organisational) changes.

As a result, it is shown that the normalised Theil index is more sensitive to the organisational properties of a software system, meaning it captures micro-level changes in functionality allocation. The hypothesis is that developers allocate functionality in accordance to an unknown distributive function $f$. This function is influenced by environmental factors such as domain requirements, time constraints, resource availability, and so on. Function $f$ distributes features into a number of collections grouped by the amount of functionality present in each measured entity. Therefore, the normalised Theil index is influenced by the distances between those groups as well as by the population share allocated to each group.
CHAPTER 6. ANALYSIS OF INEQUALITY INDEXES ON HELIX REPOSITORY

For instance, Figure 6.11 presents a sample distribution to explain the powers affecting the normalised Theil index. In this figure, the y-axis captures population share and the x-axis presents functionality concentration per measured entity. Each group presented in Figure 6.11 is a product of the function $f$; it is characterised by the associated feature concentration $v$ and population share $p$ and is represented as a pair $(v_i, p_i)$. In the sample distribution, perfect equality is achieved if the absolute difference between considered pairs reaches 0, in other words when $|(v_{i+1}, p_{i+1}) - (v_i, p_i)| = 0$. The further the absolute difference from 0, the higher inequality level in a distribution. Thus, the normalised Theil index responds to the widening distance between adjacent concentration groups and their associated population share size. The normalised Theil index grows when a group has a higher functionality concentration than its population share.

Figure 6.11: The sample distribution to present the normalised Theil index influences the $z$-Gini coefficient, on the other hand, focuses on the macro-level changes within a system, reflecting architectural shifts in the software’s evolution. It is much less sensitive to internal reorganisation events, unless such actions affect overall system design [85, 86].

So, which inequality index should be used? In the majority of observations both the normalised Theil index and the $z$-Gini coefficient evolve at the same rate. In these cases neither index offers much benefit over the other, information-wise. However, it is more practical to select an index that has less involved computation and is easier to interpret, which has been shown to be the $z$-Gini coefficient. In addition, the Gini coefficient can be applied safely to analyse data containing zeros, which the normalised Theil index fails to achieve accurately.

However, if detailed analysis is required and, especially, if the structural properties of a software system are of great interest, then the normalised Theil index has been shown
to be a valuable measure. Nevertheless, its changes should not be mistaken for those of an architectural nature. Architectural data is obtained primarily with the zGini coefficient. Thus, this work recommends to use both the normalised Theil index and the zGini coefficient in these cases to retrieve an optimal amount of information safely.

### 6.5 Summary

To conclude, the zGini coefficient and the normalised Theil index performance was evaluated when applied to inequality analysis of evolving software systems. To achieve this a number of techniques were used such as the definition of each index’s typical range and the examination of their usage when applied to software analysis.

As a result, it has been determined that, on average, the normalised Theil index has lower values but a wider typical range than the zGini coefficient across all considered metrics. Also, these two indexes are tightly correlated, implying a strong linear relationship between them. But there are cases when such a statement does not hold. Nevertheless, high correlation does not imply that the normalised Theil index and the zGini coefficient capture inequality in the same way. To evaluate that, a behaviour analysis was performed.

This chapter shows that in the majority of cases the normalised Theil index and the zGini coefficient grow as a system evolve, which corresponds to the Lehman Laws of Software Evolution. Moreover, they change at a similar rate. But these factors do not always hold. There are systems which experience decreasing inequality over time, and there are systems where the indexes do not follow the same behaviour patterns.

The cases where inequality indexes diverge or cross have been evaluated using histogram analysis. The results show that the normalised Theil index is sensitive to the structural, or micro-level, adjustments in a software system, whereas the zGini coefficient reflects architectural, or macro-level, changes. Thus, if the interest lies in definition of the organisational evolution of a software system, then both indexes should be applied to retrieve such data safely. However, if this information is of no value for a specific task, the Gini coefficient can be used since it is easier to compute and to interpret.
Chapter 7

The Economics Metaphor in Software Engineering

The fields of economics and software engineering are quite different. As a consequence, some adopted measures are unable to process metrics data adequately. In particular, the Theil index fails to evaluate distributions containing zeros appropriately. To solve this issue, all zero instances are removed from a distribution prior to an application of the Theil index; still, this approach implies that some information regarding inequality level in a distribution would be lost. Thus, this chapter focuses on analysis of the amount of information loss and the influence zero elimination has on inequality analysis.

In this chapter, Section 7.1 discusses the typical ranges of the Gini Difference – a novel measure capturing the rank difference between the Gini coefficient (software engineering inequality measure) and the zGini coefficient (economics metaphor) – and how this measure evolves over time. In addition, it evaluates the relationships between the Gini coefficient and the zGini coefficient and provides some guidelines regarding how to maintain data integrity without neglecting the economics metaphor usage for software analysis. Finally, this chapter is summarised in Section 7.2.

7.1 Zero Elimination in Software Inequality Analysis

To evaluate the effect zero elimination from metrics data has on inequality analysis, the Gini Difference (GiniDif) measure is used. This measure reflects the rank difference
between the Gini coefficient and the zGini coefficient, allowing the examination of the disparity between a direct application of inequality measures (Gini) and the economics metaphor applied (zGini) for software analysis.

Initially, a typical range analysis of the Gini Difference is performed. Since some measures are more predisposed to contain zero instances than others (i.e. setters metric in comparison to nom), the range values are grouped by metric. Figure 7.1 presents typical GiniDif ranges for each considered metric computed on the Helix repository.

![Figure 7.1: The Gini Difference (GiniDif) ranges across considered metrics](image)

This figure demonstrates that the effect zero instances removal has on inequality analysis is different for various measures. The three metrics that are the most influenced by zero elimination are noa, getters, and setters, which is evident from their relatively large GiniDif ranges. As a result in these cases excluding zeros from metrics data may cause a noticeable inequality level shift. The least affected metric is rfc, where GiniDif range is less than 0.1. As for the rest of the measures, they have similar Gini difference ranges, which are around twice as large as that for the rfc metric but much less than those of getters, setters, and nom. Thus, the nature of a considered metric is one of the major contributors towards the GiniDif value.

Figure 7.1 displays interquartile ranges of the Gini Difference, however, it hides the outliers. The maximum Gini difference value across all observations is 0.851. It is linked to the setters metric computed on RSN 1 of the lucene system. Considering that both the Gini coefficient and the zGini coefficient have the limit of 1, having a distance between
these two measures as high as 0.851 implies that zero elimination in this case causes a significant inequality level alteration. In the considered data collection, there are three cases where the GiniDif is higher than 0.8, and 81 cases where GiniDif is larger than 0.7. Altogether, these observations are associated with inequality computed on the setters metric. There are also seven releases where the Gini Difference is equal to 0, meaning no zero instances were found to be removed from the corresponding data sets. These belong to the indegree metric computed on the kolmafia system and are presented by RSN 1 to RSN 7, inclusively.

Typical ranges study presents the overall difference between software engineering inequality and economics metaphor. Still, such an approach provides no indication as to how this difference evolves. To answer this question, the GiniDif evolution behaviour for each metric is evaluated by computing the Pearson correlation value [93] between the GiniDif and RSN for each system and each metric.

The choice of the Pearson correlation measure is not random. In the previous chapter, the aim was to determine whether two inequality indexes (the Gini coefficient and the normalised Theil index) exhibit monotonic (not necessary linear) correlation. To achieve this the Spearman rank-order correlation was used as it provides a benchmark for this type of relationship [60]. The correlation study conducted here, however, is interested in evaluating the strength of a linear relationship between the GiniDif and RSN measures. Thus, the Pearson product-moment coefficient was selected for this purpose [61].

This experiment is based on the properties of the Pearson correlation coefficient [93]. Whenever the correlation value is positive it is said that two sets are positively correlated, meaning the values of each set are increasing [93]. If the correlation value is negative, two sets are said to be negatively correlated, implying that the values of one set are growing while the values of the second set are diminishing [93].

The Pearson correlation ($r$) also demonstrates the strength of a linear relationship between two data sets [93]. It has an interval of $[-1, 1]$ [93] where, if $r$ approaches 1 or $-1$ it indicates a perfect linear relationship between two sets, whereas when $r$ reaches 0 no linear relationship can be observed.

Hence, by evaluating correlation values between GiniDif and RSN, it is possible to determine how the Gini Difference changes over time. Since RSN always rises, if $r$ is negative then this implies that the Gini Difference decreases, otherwise it increases.
In this experiment, the Pearson value \((r)\) is categorised into three groups: (i) \(r > 0.4\) which corresponds to a fairly strong positive linear relationship between the GiniDif and RSN, (ii) \(0.4 < r < -0.4\) is a “blur” area - no specific tendency can be found, (iii) \(r < -0.4\) which is a group where the Gini Difference and RSN are negatively correlated. For each metric a percentage of systems with correlation values \((r)\) corresponding to each groups are calculated. The results are presented in Table 7.1.

**Table 7.1:** The Pearson correlation coefficient values between GiniDif and RSN per metric

<table>
<thead>
<tr>
<th>Measure</th>
<th>Metric</th>
<th>(% r =&lt; -0.4)</th>
<th>(% -0.4 &lt; r &lt; 0.4)</th>
<th>(% r &gt; = 0.4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ck</td>
<td>cbo</td>
<td>66.7</td>
<td>14.3</td>
<td>19.0</td>
</tr>
<tr>
<td>ck</td>
<td>rfc</td>
<td>54.8</td>
<td>14.3</td>
<td>31.0</td>
</tr>
<tr>
<td>ck</td>
<td>wmc</td>
<td>66.7</td>
<td>14.3</td>
<td>19.0</td>
</tr>
<tr>
<td>degrees</td>
<td>indegree</td>
<td>66.7</td>
<td>14.3</td>
<td>19.0</td>
</tr>
<tr>
<td>degrees</td>
<td>outdegree</td>
<td>66.7</td>
<td>14.3</td>
<td>19.0</td>
</tr>
<tr>
<td>gssummary</td>
<td>getters</td>
<td>52.4</td>
<td>21.4</td>
<td>26.2</td>
</tr>
<tr>
<td>gssummary</td>
<td>setters</td>
<td>64.3</td>
<td>21.4</td>
<td>14.3</td>
</tr>
<tr>
<td>noa</td>
<td>noa</td>
<td>45.2</td>
<td>28.6</td>
<td>26.2</td>
</tr>
<tr>
<td>nom</td>
<td>nom</td>
<td>54.8</td>
<td>19.0</td>
<td>26.2</td>
</tr>
<tr>
<td><strong>Mean</strong></td>
<td></td>
<td><strong>59.8</strong></td>
<td><strong>18.0</strong></td>
<td><strong>22.2</strong></td>
</tr>
</tbody>
</table>

Table 7.1 shows that in the majority of cases (around 60%) the GiniDif value is decreasing as a system matures. This implies that, overall, the amount of classes with a metric value of zero diminishes over time, which may indicate that developers prefer to "fill the gaps" in the system’s implementation and, thus, increase functionality concentration.

Alternatively, in average of 22.2% of observations the Gini Difference value tends to increase over system evolution. The hypothesis regarding this phenomenon is that in these cases developers prefer to introduce new instances without associated functionality rather than increase its usage. The \(rfc\) metric has the highest percentage of observations experiencing growing GiniDif values (31%). This metric represents the level of communication between classes in a system, and hence, its diminishing popularity over system evolution may indicate the weakening dependency between classes.

Finally, in an average of 18% of cases no explicit relationship can be observed between the Gini Difference and RSN. The reasons for such an observation can be two-fold: (a) the Gini coefficient and the \(z\)Gini coefficient evolve at very different rates, indicating that zero elimination causes a pronounced inequality profile change, and (b) this phenomenon is due to a very small range of GiniDif values.

The typical range analysis is a technique used to evaluate the amount of information lost while removing zeros. The correlation analysis is a method to define whether the
amount of zero instances decreases or increases over time. Neither of these approaches, however, illustrate whether upon zero elimination a simple scaling factor or inequality value distortion takes place. To analyse the effect “vanishing” zero instances have on an inequality profile a table was created, presented in Table 7.3, listing types of relationships between the zGini coefficient and the Gini coefficient when calculated on considered metrics data.

In this table a number of symbols are used to represent the relationships between the Gini coefficient and the zGini coefficient. The definition of each symbol is recorded in Table 7.2.

Table 7.2: Symbolic representation of behaviour patterns between the Gini coefficient and the zGini coefficient

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Parallel</td>
<td>Indicates that the Gini coefficient and the zGini coefficient evolve in parallel.</td>
</tr>
<tr>
<td>≃</td>
<td>Semi-parallel</td>
<td>Indicates that, even though the Gini coefficient and the zGini coefficient experience a similar development pattern, there are still dissimilarities causing a slight diversion between the indexes evolution trends.</td>
</tr>
<tr>
<td>⋏</td>
<td>Antithetical</td>
<td>Indicates that the Gini coefficient and the zGini coefficient have no behaviour similarities – develop in different directions over time.</td>
</tr>
</tbody>
</table>

Table 7.3: The Gini coefficient and the normalised Theil index behavioural trends

<table>
<thead>
<tr>
<th>No</th>
<th>System</th>
<th>cbo</th>
<th>rfc</th>
<th>wmc</th>
<th>indegree</th>
<th>outdegree</th>
<th>getters</th>
<th>setters</th>
<th>noa</th>
<th>nom</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>acegi</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
</tr>
<tr>
<td>2</td>
<td>activebpel</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
</tr>
<tr>
<td>3</td>
<td>activemq</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
</tr>
<tr>
<td>4</td>
<td>ant</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
</tr>
<tr>
<td>5</td>
<td>axis</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
</tr>
<tr>
<td>6</td>
<td>azureus</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
</tr>
<tr>
<td>7</td>
<td>castor</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
</tr>
<tr>
<td>8</td>
<td>checkstyle</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
</tr>
<tr>
<td>9</td>
<td>cocoon</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
</tr>
<tr>
<td>10</td>
<td>columba</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
</tr>
<tr>
<td>11</td>
<td>compass</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
<td>≃</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 7.3 – Continued from previous page

<table>
<thead>
<tr>
<th>No</th>
<th>System</th>
<th>Metric</th>
<th>cbo</th>
<th>rfc</th>
<th>wmc</th>
<th>indegree</th>
<th>outdegree</th>
<th>getters</th>
<th>setters</th>
<th>noa</th>
<th>nom</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>findbugs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>flow4j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>freemarker</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>groovy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>hibernate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>ibatis</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>itext</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>jabref</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>jasperreports</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>jchempaint</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>jena</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>jfreechart</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>jmeter</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>jurby</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>jung</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>kolmafia</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>lucene</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>pmd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>proguard</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>rssowl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>saxon</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>soapui</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>spring</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>stripes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>struts</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>tapestry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>webwork</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>wicket</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>xalan</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>xerces</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>xwork</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
So, which behaviour pattern between the Gini coefficient and the zGini coefficient is that most followed? Table 7.4 list the occurrence rate of each trend per metric, and, thus, summarises information from Table 7.3.

**Table 7.4: Table 7.3 summary**

<table>
<thead>
<tr>
<th>Measure</th>
<th>Relationship Occurrence (in %)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>==</td>
</tr>
<tr>
<td>cbo</td>
<td>28.6</td>
</tr>
<tr>
<td>rfc</td>
<td>76.2</td>
</tr>
<tr>
<td>wmc</td>
<td>61.9</td>
</tr>
<tr>
<td>indegree</td>
<td>40.5</td>
</tr>
<tr>
<td>outdegree</td>
<td>35.7</td>
</tr>
<tr>
<td>getters</td>
<td>19.0</td>
</tr>
<tr>
<td>setters</td>
<td>11.9</td>
</tr>
<tr>
<td>noa</td>
<td>9.5</td>
</tr>
<tr>
<td>nom</td>
<td>38.1</td>
</tr>
<tr>
<td>Mean</td>
<td>35.7</td>
</tr>
</tbody>
</table>

Overall, the ”semi-parallel” behaviour pattern is the predominant one. It is found in 55.3% of considered cases. The second-most popular behaviour trend is ”parallel”, which has an average rate of 35.7%. The ”antithetical” relationship is present in the average of 9.0% of observations. Examples of each behaviour pattern are illustrated in Figure 7.2.

The popularity of these three behaviour trends, though, vary between measures. For instance, the coefficients calculated on rfc metric primarily evolve in parallel (76.2%) with no ”antithetical” relationship demonstrated. Thus, removing zero instances, in this case, has no significant effect on the inequality level.

The getters, noa, and setters metrics though, exhibit opposite behaviour to the rfc measure. The most prominent relationship demonstrated by these observations is ”semi-parallel”, which is present in 76.2% of cases for getters, 73.8% of observations for noa, and 66.7% of cases for the setters metric. Also, setters and noa measures experience the highest occurrence rate for the ”antithetical” trend which are 21.4% and 16.7% respectively. Thus, for these metrics, eliminating zero instances from data set causes notable distribution change resulting in inequality level falsification.

Overall, in the majority of cases zero elimination from the metrics data set causes inequality level distortion. Hence, a direct application of the economics metaphor for software analysis is unsafe. The effect, however, differs between metrics. For some, such as rfc, distribution changes slightly when zeros are removed. For other metrics (i.e. getters or noa) zero elimination has a significant influence on the quality of inequality analysis.
Removing zero instances, however, is a valuable technique, especially if the Theil index is required. Though, in this case it is important to calculate economics inequality (eliminating zeros) along with software engineering inequality (including zeros) in order to maintain the integrity of collected inequality data.

### 7.2 Summary

When applying the economics metaphor to software analysis, it is valuable to evaluate the effect such an approach has on results. To achieve that, a facilitating measure – the Gini Difference – was developed. This measure calculates the rank difference between the Gini coefficient (software engineering inequality measurement) and the zGini coefficient (economics metaphor alternative).
Initially, the typical range analysis of the Gini Difference was performed. Here, the high GiniDif values are associated with those metrics that enclose a relatively large amount of zero instances – *setters* and *getters*. The relatively small GiniDif values are present in *rfc* metric observations, since it lacks zeros in its data.

The Gini Difference also evolves. In the majority of cases it decreases, which may indicate the increasing popularity of a specific functionality in a system. There are, however, cases where GiniDif grows. Interestingly, this behaviour is most apparent when studying the *rfc* metric.

Finally, the relationship between the Gini coefficient and the zGini coefficient was analysed. The observed predominant relationship pattern is ”semi-parallel”, representing a slight difference between the Gini coefficient and the zGini coefficient evolution trends. However, this is already an indication that zero removal from metrics data causes distribution distortion, and hence affects corresponding inequality values. The proposed solution here is to apply both the economics metaphor and software engineering inequality measurement techniques, together, in order to employ inequality indexes that are unable to process zeros sensitively and to maintain overall data integrity.
CHAPTER 7. THE ECONOMICS METAPHOR IN SOFTWARE ENGINEERING
Chapter 8

Conclusion

8.1 Summary

Software measurement is the powerful technique for evaluating the effectiveness of a software development process. However, software metrics data alone is ambiguous [77]. Such data needs to be summarised to be interpreted [86]. The crucial factor here is a careful selection of summary statistics used to construe metric results. For this purpose, this work adopts inequality measures, borrowed from the field of economics [24]. The main benefit of this approach is its sensitivity to the changes in metrics data allocation and its computational independence to the nature of a distribution.

Inequality measurement, though, requires software metrics data. Thus, a collection of tools supporting metrics data mining and rendering processes is needed. A number of software systems have been already developed to achieve this (JSeat [3], Mutations [5], or jHawk [4]), though it is often difficult to find one that fits individual research goals precisely. Some tools lack data precision, others are difficult to extend. In addition, to evaluate software systems, we need to understand how they are created, which is best achieved via participating in a software project. Thus jCT, or Java Code Tomograph, was developed to facilitate software metrics collection in this work [52].

jCT is both a high-resolution data miner and a powerful tool for performing quantitative analysis on Java byte code. The main features of this metrics extraction tool are: high-precision data collection, an extensible metrics extraction engine, and built-in support for the Helix [87] repository. However, jCT is a tool for metrics extraction, not interpreta-
tion. To achieve the latter, a number of supporting tools were developed that (i) perform data reorganisation to fit research requirements (CSVProcessor), (ii) execute experiments (Inequality Experiment Runner), and (iii) fulfil post-processing activities using inequality indexes (Inequality Calculator). Statistical analysis tools, such as R and STATA, are also employed to aid metrics data interpretation.

Inequality analysis indicates whether resources are allocated evenly within a collection, and, if not, it shows the strength of a disparity between entities’ wealth [24]. Inequality itself is due to parametric variations [7] – external factors affecting the relationship between available resources and the corresponding entities holding these resources. They are caused by a number of environmental factors [69] such as individual (personal) heterogeneities or sub-group resources distribution.

In this work, the focus lies upon two inequality measurement techniques - the Gini coefficient and the Theil index. One of them, the Gini coefficient, can be employed directly for software analysis since it is not affected by the presence of certain values (zeros) in a distribution. The Theil index, however, by design, is unable to process data containing zeros sensitively, of which a software metrics data often contains a profuse amount. Thus, to avoid inequality measurement distortion when evaluating software systems with this index, the Theil index should be applied on data with all zero instances removed prior to computation. In addition, its values are tightly dependent on the size of a considered data set, thus, to be comparable to the Gini coefficient and to achieve intuitive interpretation of its values, the normalised Theil index is used for software analysis. Still, zeros elimination implies information loss. To determine the amount of data displaced while performing such an operation, a new measure was developed – the Gini Difference. This measure captures the rank difference between the Gini coefficient, computed normally, and the Gini coefficient, computed on the same set with all zero instances removed prior to calculation – the economic metaphor (zGini).

Initially, a number of experiments were performed to determine how both indexes capture inequality. Each experiment represents a highly controlled environment where the distribution values shift in accordance to the Pigou-Dalton condition [27]. As a result, this work shows that the normalised Theil index inherits the size dependency property from the original Theil index, whereas the Gini coefficient remains indifferent to an increased data size. When compared, the Gini coefficient tends to be more sensitive in the upper bounds of inequality, whereas the normalised Theil index captures inequality with a better precision when inequality is still relatively low.
The next step taken in this work includes the application of both inequality indexes for software analysis. To maintain comparability and applicability of both indexes, only the zGini coefficient and the normalised Theil index were analysed. It has been shown that in almost 95% of observations the two indexes are tightly correlated, implying a strong linear relationship between them. But there are still cases when such a finding does not hold. In addition, high correlation does not imply that the normalised Theil index and the zGini coefficient capture inequality in the same way. Thus, the typical behaviours of both indexes were evaluated. In the majority of cases the normalised Theil index and the zGini coefficient grow as a system evolves, corresponding to the Lehman Laws of Software Evolution [44]. Moreover, the indexes tend to evolve at the same rate, but there are exceptions to this observation. The indexes also demonstrate decreasing values over time, as well as the cases where they do not follow a similar evolution pattern (they develop at different rates).

To evaluate what kind of data inequality indexes capture when employed for software analysis, in addition to their behaviour study, histogram analysis was performed. It was determined that the normalised Theil index is sensitive to the organisational, or micro-level, adjustments in a software system, whereas the zGini coefficient reflects architectural, or macro-level, changes. Thus, this work proposes to employ both indexes for software analysis to collect information about both the structural and architectural development of a software system. If such detailed analysis is not required, the zGini coefficient would suffice.

However, both the normalised Theil index and the zGini coefficient are playing the role of economics metaphor application to software analysis. In which case, only part of the associated system information is analysed when these indexes are computed. This raises a question: do their values reflect the overall system’s inequality also? To answer this, the Gini Difference was evaluated, as well as the types of relationships between the Gini coefficient and the zGini coefficient. The experiments have shown that zeros removal from metrics data causes its distribution to change, and hence affects the corresponding inequality values. Thus, economics metaphor employment for software analysis does not reflect the overall system’s profile. The proposed solution here is to apply both the economics metaphor and software engineering inequality measurement techniques, together. This allows researchers to use inequality indexes that are unable to process zeros adequately as well as to maintain overall inequality data integrity.
8.2 Observations

This work presents a number of key contributions to the research community, listed below:

- The Theil index’s value is tightly dependent to a data size which makes its interpretation and comparability difficult. The solution proposed in previous research [71, 91, 92] is the usage of a normalisation function on this index to achieve a value within range $[0, 1]$. However, the findings discussed in this work show that the normalised Theil index is also size dependent – a property which has to be taken into account when using this measure for software evaluation. Nevertheless, such a finding does not prevent researchers from employing this index for software analysis.

- The Gini coefficient and the Theil index capture inequality from different perspectives: the Gini coefficient evaluates macro-level inequality corresponding to the architectural features of a software system, and the Theil index captures micro-level inequality – organisational properties of a software system. Thus neither index can be considered as a substitute to one another nor as a superior inequality measure. These two coefficients simply capture different information.

- When a detailed evaluation of inequality in a software system is of no interest to a researcher, the Gini coefficient would be a better inequality measure than the Theil index. This is so for multiple reasons: the Theil index is a size-dependent measure, it is unable to process zero instances sensitively, and, most importantly, it has more complicated computation process than the Gini coefficient. However, in other cases the application of both inequality measures is advisable. This is to avoid confusion in the interpretation of architectural shifts and restructuring events.

- The Theil index lacks an ability to process zero instances in data adequately. This is due to the presence of $\log(n)$ as a sub-term in the Theil index formula. Thus, it is advised to employ this measure on a data collection with zero instances removed prior to an investigation. However, when computed in combination with the Gini coefficient, ecological fallacy might occur since both measures are calculated on separate data sets, even if one of the data collections is a subset of another. Thus, when comparison to the Gini coefficient is required, the $z$-Gini coefficient should be computed in its place, ensuring that two indexes capture inequality on the same distribution.

- Completely eliminating zero instances from software metrics data causes the associated inequality level to change. Thus, when employing inequality measures that are unable to process zeros adequately, it is valuable to include a secondary mea-
sure, that is computed on a full data collection, in order to compare the inequality value, computed on data excluding zeros, to the original inequality level.

- Overall, a naïve application of inequality measures for software analysis can lead to incorrect or unreliable results. Thus, every measure requires a thorough analysis prior to its usage for software evaluation.

8.3 Future Work

There are opportunities for future research in this area. These are as follows:

- This work covers the usage of the individual Theil index only. However, such an inequality measure supports decomposition analysis also – the group Theil index. Evaluation of between and within groups inequality would allow researchers to study software systems at various decomposition levels. There are, however, issues with such an approach. For instance, it is unknown how deep the decomposition should be for a meaningful software analysis and what value this approach has to the research community. Thus, it is unknown whether the convoluted computation of the group Theil index worth the associated effort, or if the information gain is insignificant.

- Another extension to this work would be to employ inequality measures to create a specific imprint of each system’s functionality and organisation – software “DNA”. The idea is to use this information, similar to the study of human DNA, to define system’s inherent nature, shape, and drawbacks. Thus, the predictions could be made regarding a final product prior to commencing a development process using the ”genetic” information retrieved from the study of either a supporting library, a development team, a development approach, or all together.

- This work employed just two inequality measures for software analysis. However, there exist other techniques to measure inequality in a data collection [24]. All of those are also borrowed from the field of economics. It would be valuable, though, to define how these indexes can be applied for software evaluation, how they differ to the considered inequality measures, and what benefits they might bring to software studies.
References


Appendix A

jCT Output File Examples

Package,ClassCount
org/hsqldb,118
org/hsqldb/cmdline,15
org/hsqldb/cmdline/sqltool,4
org/hsqldb/dbinfo,4
org/hsqldb/error,2
org/hsqldb/index,6
org/hsqldb/jdbc,34
org/hsqldb/jdbc/pool,15
org/hsqldb/lib,101
org/hsqldb/lib/java,1
org/hsqldb/lib/tar,14
org/hsqldb/navigator,9
org/hsqldb/persist,55
org/hsqldb/resources,1
org/hsqldb/result,6
org/hsqldb/rights,6
org/hsqldb/rowio,16
org/hsqldb/scriptio,6
org/hsqldb/server,26
org/hsqldb/store,11
org/hsqldb/types,31
org/hsqldb/util,48

Figure A.1: cpp Metric Output File for hsqldb-2.0.0.
APPENDIX A. JCT OUTPUT FILE EXAMPLES

org.hsqldb.jdbc.JDBCNClob:
Magic: CAFEBABE
Minor: 0
Major: 50
Constant pool:
CONSTANT_Methodref: org.hsqldb.jdbc.JDBCClob::<init> ()void
CONSTANT_Methodref: org.hsqldb.jdbc.JDBCClob::<init> (java.lang.String):void
CONSTANT_Class: org.hsqldb.jdbc.JDBCClob
CONSTANT_Class: org.hsqldb.jdbc.JDBCNCllob
CONSTANT_Class: java.sql.NClob
CONSTANT_Utf8: <init>
CONSTANT_Utf8: ()V
CONSTANT_Utf8: Code
CONSTANT_Utf8: (Ljava/lang/String;)V
CONSTANT_Utf8: Exceptions
CONSTANT_Class: java.sql.SQLException
CONSTANT_NameAndType: <init> ()void
CONSTANT_NameAndType: <init> (java.lang.String):void
CONSTANT_Utf8: org/hsqldb/jdbc/JDBCNCllob
CONSTANT_Utf8: org/hsqldb/jdbc/JDBCClob
CONSTANT_Utf8: java/sql/NClob
CONSTANT_Utf8: java/sql/SQLException
Access flags: public super
Super: org.hsqldb.jdbc.JDBCClob
Interfaces:
java.sql.NClob
Methods:
- protected <init> ()void
  Attributes:
  Code:
  MaxStack: 1
  MaxLocals: 1
  Byte code:
  0000: aload_0
  0001: invokespecial org.hsqldb.jdbc.JDBCClob::<init> ():void
  0004: return
- public <init> (java.lang.String):void
  Attributes:
  Code:
  MaxStack: 2
  MaxLocals: 2
  Byte code:
  0000: aload_0
  0001: aload_1
  0002: invokespecial org.hsqldb.jdbc.JDBCClob::<init> (java.lang.String):void
  0005: return
Exceptions:
java.sql.SQLException

Figure A.2: isummary Metric Output File for org.hsqldb.jdbc.JDBCNCllob class in hsqldb-2.0.0.
Figure A.3: gssummary Measure Output File hsqldb-2.0.0 (partial).
Figure A.4: nom Measure Output File for hsqldb-2.0.0(partial).
Appendix B

The Theil Index and the Gini Coefficient Behaviour Analysis

Figure B.1: The outdegree histogram for every release of ant system
Figure B.2: The *indegree* histogram for every release of *findbugs* system

(a) RSN 1  
(b) RSN 3  
(c) RSN 4  
(d) RSN 8  
(e) RSN 11  
(f) RSN 14  
(g) RSN 15  
(h) RSN 17  
(i) RSN 20

Figure B.3: The *cbo* histograms for every release of *activebpel* system

(a) RSN 1  
(b) RSN 6  
(c) RSN 10  
(d)  
(e)
Figure B.4: The rfc histogram for every release of jchempaint system
Figure B.5: The \textit{cbo} histogram for every release of \textit{tapestry} system
APPENDIX B. THE THEIL INDEX AND THE GINI COEFFICIENT BEHAVIOUR ANALYSIS

Figure B.6: The outdegree histogram for every release of pmd system

(a) RSN 1  (b) RSN 7  (c) RSN 8
(d) RSN 10 (e) RSN 11 (f) RSN 12
(g) RSN 13 (h) RSN 18 (i) RSN 19
(j) RSN 27 (k) RSN 33 (l) RSN 37
(m) RSN 42