Growth and Change Dynamics in Open Source Software Systems

Faculty of Information and Communication Technologies
Swinburne University of Technology
Melbourne, Australia

Submitted for the degree of Doctor of Philosophy

Rajesh Vasa

2010
Abstract

In this thesis we address the problem of identifying where, in successful software systems, maintenance effort tends to be devoted. By examining a larger data set of open source systems we show that maintenance effort is, in general, spent on addition of new classes. Interestingly, efforts to base new code on stable classes will make those classes less stable as they need to be modified to meet the needs of the new clients.

This thesis advances the state of the art in terms of our understanding of how evolving software systems grow and change. We propose an innovative method to better understand growth dynamics in evolving software systems. Rather than relying on the commonly used method of analysing aggregate system size growth over time, we analyze how the probability distribution of a range of software metrics change over time. Using this approach we find that the process of evolution typically drives the popular classes within a software system to gain additional clients over time and the increase in popularity makes these classes change-prone.

Furthermore, we show that once a set of classes have been released, they resist change and the modifications that they do undergo are in general, small adaptations rather than substantive rework. The methods we developed to analyze evolution can be used to detect releases with systemic and architectural changes as well as identify presence of machine generated code.

Finally, we also extend the body of knowledge with respect to validation of the Laws of Software Evolution as postulated by Lehman. We find consistent support for the applicability of the following laws of software evolution: first law Continuing Change, third law Self Regulation, fifth law Conservation of Familiarity, and the sixth law Continuing Growth. However, our analysis was unable to find evidence to support the other laws.
Acknowledgements

I would like to acknowledge with particular gratitude the assistance of my supervisors, Dr. Jean-Guy Schneider and Dr. Philip Branch. I am also indebted to a number of other people, in particular, Dr. Markus Lumpe, Prof. Oscar Nierstrasz, Clinton Woodward, Andrew Cain, Dr. Anthony Tang, Samiran Muhmud, Joshua Hayes and Ben Hall who collaborated with me on various research papers 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 current employer, Swinburne University of Technology for providing the resources and support to pursue a research higher degree.

Finally, I would like to thank my family for their loving forbearance during the long period it has taken me to conduct the research and write up this thesis.

Rajesh Vasa, 2010
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.

Rajesh Vasa, 2010
Publications Arising from this Thesis

The work described in this thesis has been published as described in the following list:


Although the thesis is written as a linear document, the actual research work involved substantial exploration, idea formation, modelling, experimenting and some backtracking as we hit dead-ends. The following text outlines how the publications relate to this thesis.

The early articles helped lay the foundation and scope the work presented in this thesis. Specifically, the QAOOSE’03 and ISESE’05 articles (papers 1 and 2) showed that software metrics typically exhibit highly skewed distributions that retain their shape over time and that architectural changes can be detected by analyzing these changing distributions. The article published at SC’2007 (paper 3) expanded on the ISESE’05 article (paper 2) and presented a mathematical model to describe the evolution process and also put forward the thresholds as well as a technique to detect substantial changes between releases. These papers helped establish and refine the input data selection method (Chapter 3), validate the approach that we take for extracting metrics (Chapter 4), and developed the modelling approach that we eventually used to detect substantial changes between releases (Chapter 5).

More recent work (in particular, ICSM’07 and ICSM’09 articles and the EVOL’07 article – papers 4, 5 and 7) contributed to the content presented in Chapters 5 and 6 of this thesis which address the primary research questions. The article in ASWEC’10 (paper 8) showed that the key analysis approach advocated in this thesis can also be used to understand how properties are used in Java software. The IEEE Software article in 2009 (paper 6) presented a method for reasoning about software architecture and the findings from this thesis influenced some of the arguments with respect to the long term stability of software architecture. The implications that we derived from all of the various papers are expanded upon in Chapter 7.
# Contents

1 Introduction ................................................. 1

1.1 Research Goals ............................................. 4

1.2 Research Approach ......................................... 5

1.3 Main Research Outcomes ................................. 6

1.4 Thesis Organisation ........................................ 8

2 Software Evolution ........................................ 10

2.1 Evolution .................................................. 10

2.2 Software Evolution ......................................... 13

2.3 The Laws of Software Evolution ......................... 18

2.4 Studies of Growth ......................................... 22

2.5 Studies of Change ......................................... 30

2.6 Research Questions ........................................ 39

3 Data Selection Methodology ............................... 41

3.1 Evolution History ......................................... 41

3.2 Open Source Software (OSS) ............................ 44

3.3 Open Source Project Repositories ....................... 46
3.4 Selection Criteria ................................................. 47
3.5 Selected Systems - An Overview ................................. 50
3.6 Focus of Study ...................................................... 53
  3.6.1 Categories of data sources .................................. 53
  3.6.2 Java Software Systems ....................................... 54
3.7 Summary ............................................................ 59

4 Measuring Evolving Software ................................. 60
  4.1 Measuring Software ............................................. 61
  4.2 Types of Metrics .................................................. 62
    4.2.1 Size Metrics .................................................. 62
    4.2.2 Complexity Metrics ......................................... 63
  4.3 Software Evolution History Model ............................ 66
  4.4 Measuring Time .................................................. 67
    4.4.1 Release Sequence Number (RSN) ......................... 67
    4.4.2 Calendar Time ............................................... 70
  4.5 Metric Extraction ............................................... 72
    4.5.1 Jar Extraction ............................................... 73
    4.5.2 Class Metric Extraction .................................... 74
    4.5.3 Merge Inner Classes ....................................... 82
    4.5.4 Class Dependency Graph Construction .................. 82
    4.5.5 Dependency Metric Extraction ............................ 84
    4.5.6 Inheritance Metric Extraction ............................ 88
5 Growth Dynamics

5.1 Nature of Software Metric Data
   5.1.1 Summarising with Descriptive Statistics
   5.1.2 Distribution Fitting to Understand Metric Data

5.2 Summarizing Software Metrics
   5.2.1 Gini Coefficient - An Overview
   5.2.2 Computing the Gini Coefficient
   5.2.3 Properties of Gini Coefficient
   5.2.4 Application of the Gini Coefficient - An Example

5.3 Analysis Approach
   5.3.1 Metrics Analyzed
   5.3.2 Metric Correlation
   5.3.3 Checking Shape of Metric Data Distribution
   5.3.4 Computing Gini Coefficient for Java Programs
   5.3.5 Identifying the Range of Gini Coefficients
   5.3.6 Analysing the Trend of Gini Coefficients

5.4 Observations
   5.4.1 Correlation between measures
   5.4.2 Metric Data Distributions are not Normal
   5.4.3 Evolution of Metric Distributions
   5.4.4 Bounded Nature of Gini Coefficients
5.4.5 Identifying Change using Gini Coefficient .......................... 115
5.4.6 Extreme Gini Coefficient Values ................................. 117
5.4.7 Trends in Gini Coefficients ........................................ 118
5.4.8 Summary of Observations ........................................... 119
5.5 Discussion ...................................................................... 121
  5.5.1 Correlation between Metrics ................................. 122
  5.5.2 Dynamics of Growth ............................................. 123
  5.5.3 Preferential Attachment ....................................... 124
  5.5.4 Stability in Software Evolution .............................. 126
  5.5.5 Significant Changes ............................................ 128
  5.5.6 God Classes ....................................................... 130
  5.5.7 Value of Gini Coefficient ..................................... 132
  5.5.8 Machine-generated Code ..................................... 134
5.6 Summary ....................................................................... 136

6 Change Dynamics .............................................................. 139
  6.1 Detecting and Measuring Change .............................. 141
    6.1.1 Approaches for Detecting Change ....................... 141
    6.1.2 Our Approach for Detecting Change in a Class ........ 143
    6.1.3 Identifying Modified, New and Deleted Classes .... 144
    6.1.4 Measuring Change .......................................... 145
    6.1.5 Measuring Popularity and Complexity of a Class ... 146
  6.2 Observations ............................................................... 146
6.2.1 Probability of Change ........................................ 147
6.2.2 Rate of Modification ...................................... 154
6.2.3 Distribution of the Amount of Change ............... 156
6.2.4 Modified Classes and Popularity ....................... 158
6.2.5 Modification Probability and In-Degree Count ...... 160
6.2.6 Popularity of New Classes .............................. 161
6.2.7 Structural Complexity of Modified Classes .......... 163
6.2.8 Summary of Observations ............................... 165

6.3 Discussion .................................................. 166
6.3.1 Probability of Change ................................... 166
6.3.2 Rate and Distribution of Change ....................... 168
6.3.3 Popularity of Modified Classes ......................... 169
6.3.4 Popularity of New Classes ............................ 171
6.3.5 Complexity of Modified Classes ...................... 171
6.3.6 Development Strategy and its Impact on Change ... 172

6.4 Related work ............................................... 175
6.5 Limitations ................................................ 180
6.6 Summary .................................................. 181

7 Implications .................................................. 182

7.1 Laws of Software Evolution ................................. 182
7.2 Software Development Practices .......................... 187
7.2.1 Project Management .................................... 187
List of Figures

2.1 The process of evolution. .............................................. 12

2.2 The different types of growth rates observed in evolving software systems. .............................................. 23

2.3 Illustration of the segmented growth in the Groovy language compiler. The overall growth rate appears to be super-linear, with two distinct sub-linear segments. ........ 27

3.1 Component diagram of a typical software system in our study. Only the Core System JAR components (highlighted in the image) are investigated and used for the metric extraction process. .............................................. 55

4.1 UML Class diagram of evolution history model. ............... 66

4.2 Time intervals (measured in days) between releases is erratic. .............................................. 69

4.3 Cumulative distribution showing the number of releases over the time interval between releases. .............................................. 70

4.4 Age is calculated in terms of the days elapsed since first release. .............................................. 71

4.5 The metric extraction process for each release of a software system .............................................. 72

4.6 The dependency graph that is constructed includes classes from the core, external libraries and the Java framework. The two sets, $N$ and $K$, used in our dependency graph processing are highlighted in the figure. .............................................. 83
List of Figures

4.7 Class diagram showing dependency information to illustrate how dependency metrics are computed. The metrics for the various classes shown in the table below the diagram 86

4.8 Class diagram to illustrate how inheritance metrics are computed. The metrics for the diagram shown in the table below the diagram. ............................... 88

5.1 Relative and Cumulative frequency distribution showing positively skewed metrics data for the Spring Framework 2.5.3. The right y-axis shows the cumulative percentage, while the left side shows the relative percentage. .......... 93

5.2 Change in Median value for 3 metrics in PMD. .............. 95

5.3 Lorenz curve for Out-Degree Count in Spring framework in release 2.5.3. .................................................. 100

5.4 Correlation coefficient distributions across all systems and releases. The top graph shows the box-plots for each of the 10 metrics under analysis. The bottom graph plots the distribution for the metrics. ............................. 108

5.5 Spring evolution profiles showing the upper and lower boundaries on the relative frequency distributions for Number of Branches, In-Degree Count, Number of Methods and Out-Degree Count. All metric values during the entire evolution of 5 years fall within the boundaries shown. The y-axis in all the charts shows the percentage of classes (similar to a histogram). .................................. 112

5.6 The distinct change in shape of the profile for Hibernate framework between the three major releases. Major releases were approximately 2 years apart. ............... 113

5.7 Box plot of Gini coefficients across all selected Java systems.115

5.8 IDC Gini evolution for Struts. ................................. 116

5.9 Evolution of selected Gini coefficients in Spring. The highlighted sections are discussed in Section 5.4.5. ................. 117

5.10 Box plot of Gini coefficients correlated with Age ............... 120
5.11 Correlation between Number of Branches and all other measures in ProGuard. Box plots show the range and distribution of the correlation coefficients over the release history. 122

5.12 Evolution of Type Construction Count for ProGuard. . . . . 129

5.13 NOB Gini profile for JabRef. . . . . . . . . . . . . . . . . . . . . 135

6.1 Change evolution in the Hibernate framework. This graph illustrates change property captured by Equation 6.2.9. . . 149

6.2 Change evolution in the Hibernate framework. This graph illustrates change property captured by Equation 6.2.10. . 150

6.3 Box plot of system maturity showing distribution of age (in days since birth) and if the change properties hold. Graph on the left covers Equation 6.2.9, while the right side graph covers Equation 6.2.10. . . 151

6.4 Probability of change reduces with system maturity. Graph on the left indicates the probability that Equation 6.2.9 holds, while the right side graph indicates probability for Equation 6.2.10. The probabilities were predicted from the Logistic regression models. Age indicates days since birth. . . . . . . . . . . . . . . . . . . . . . 152

6.5 Cumulative distribution of the modification frequency of classes that have undergone a change in their lifetime. The figure only shows some systems from our data set to improve readability. Systems that are considered outliers have been shown with dotted lines . . . . . . . . . . 155

6.6 Number of measures that change for modified classes. x-axis shows the number of measures that have been modified, while the y-axis shows the percentage of classes . . 156

6.7 Spring In-Degree Count evolution. Proportion of modified classes with high In-Degree Count is greater than that of new or all classes. . . . . . . . . . . . . . . . . . . . . . 159

6.8 Probability of modification increases as the In-Degree Count of a class increases. This graph is generated based on predicted values from the Logistic regression where In-Degree Count is the independent variable. . . . . . . . . . . . . . . . . . . . . . 161
6.9 Probability of modification increases as the *Number of Branches* of a class increases. The graph on the left shows the relationship between *Number of Branches* (independent variable) and the probability that a class will be modified. The graph of the right uses the Size adjusted branch count as the independent variable. As can be seen from the graphs, the probability increases independent of the size of the class.
List of Tables

2.1 The Laws of Software Evolution [175] .................................. 19

3.1 The different types of histories that typically provide input data for studies into software evolution. ....................... 42

3.2 The criteria that defines an Open Source Software System. 45

3.3 Systems investigated - Rel. shows the total number of distinct releases analyzed. Age is shown in Weeks since the first release. Size is a measure of the number of classes in the last version under analysis. .......................... 52

4.1 Structure of a compiled Java Class. Items that end with an * indicate a cardinality of zero or more [180]. ............... 75

4.2 Direct count metrics computed for both classes and interfaces. ................................................................. 79

4.3 Metrics computed by processing method bodies of each class. The mapping between these measures and the byte-code is presented in Appendix C. ................................. 80

4.4 Flags extracted for each class. ................................. 81

4.5 Dependency metrics computed for each class. ............. 85

4.6 Inheritance metrics computed for each class. .............. 87

5.1 Collected measures for distribution and change analysis using the Gini Coefficient ................................. 103
List of Tables

5.2 Spearman’s Rank Correlation Coefficient values for one version (0.3.0) of JasperReports. Strong correlation values are highlighted. .......................... 109

5.3 Gini value ranges in Spring Framework across 5 years of evolution .................................................. 114

5.4 Sample of observed significant changes to Gini coefficients in consecutive releases. ......................... 128

A.1 Meta data captured for each software system .......... 201

C.1 Metrics are computed by processing opcodes inside method bodies of each class. ......................... 203

E.1 Data files used in the study of Growth (Chapter 5) ... 206

F.1 Data files used in the study of Change (Chapter 6) ... 208
Chapter 1

Introduction

Software engineering literature provides us with a diverse set of techniques and methods on how one should build software. This includes methodologies [49, 158, 159], modelling notations [2, 32] as well as advice on how best to structure, compose and improve software systems [70, 81, 88, 237]. This knowledge base has also been enhanced by work investigating how humans tend to construct software [297] and by advances in understanding how we can better organise teams [50, 56]. We also have techniques available to measure properties of software and guidelines on what would be considered desirable characteristics of software development [77, 165, 227]. Despite a wealth of knowledge in how to construct software, relatively little deep knowledge is available on what software looks like and how its internal structure changes over time. This knowledge is critical as it can better inform, support and improve the quality of the guidance provided by much of the software engineering literature. Despite this need, a survey of empirical research in software engineering has found that less than two percent of empirical studies focused on maintenance and much less on how software evolves [147].

Research in the field of software evolution aims to bridge the gap in our understanding of how software changes by undertaking rigorous studies of how a software system has evolved. Over the past few decades, work in this field has identified generalizations that are summarized
in the laws of software evolution [174, 175] and has identified general facets of evolution [198], put forward techniques for visualising evolution and change [59, 65, 95, 163], collated and analyzed software metric data in order to understand the inherent nature of change [27], assembled methods for identifying change prone components [109, 281] as well as advise on expected statistical properties in evolving software systems [21, 270]. Although earlier work on how software evolves focused on large commercial systems [17, 146, 147, 175, 283], recent studies have investigated open source software systems [41, 100, 101, 192, 239, 305]. This work has been enriched by more recent studies into how object oriented software systems evolve [64, 71, 193, 269, 270].

An important contribution of research in the field of software evolution are the Laws of Software Evolution, as formulated and refined by Lehman and his colleagues [171, 172, 174, 175], which state that regardless of domain, size, or complexity, software systems evolve as they are continually adapted, they become more complex, and require more resources to preserve and simplify their structure. The laws also suggest that the process of evolution is driven by multi-level feedback, where the feedback mechanisms play a vital role in further evolution in both the evolution process as well as the software that is produced.

From a practical point of view, software development can be seen as a process of change. Developers work with, and build on top of existing libraries, as well as the code base from the previous version. Starting from an initial solution, most software systems evolve over a number of releases, each new release involving the following activities: (i) defect identification/repair, (ii) addition of new functionality, (iii) removal of some existing functionality, and (iv) optimisations / refactoring. When looking at this process from an evolutionary perspective, software developers tend to undertake all of the activities outlined above between two releases of a software system, possibly resulting in a substantial number of changes to the original system. The decisions that are made as a part of this process are constrained by their own knowledge, as well as the existing code base that they have to integrate the new enhancements into.
Chapter 1. Introduction

Given that change is inherent within an active and used software system, the key to a successful software evolution approach lies not only in anticipating new requirements and adapting a system accordingly [87], but also in understanding the nature and the dynamics of change, especially as this has an influence on the type of decisions the developers make. Changes over time lead to software that is progressively harder to maintain if no corrective action is taken [168]. Compounding this, these changes are often time consuming to reverse even with tool support. Tools such as version control systems can revert back to a previous state, but they cannot bring back the cognitive state in the developer’s mind. Developers can often identify and note local or smaller changes, but this task is much more challenging when changes tend to have global or systemic impact. Further, the longer-term evolutionary trends are often not easily visible due to a lack of easy to interpret summary measures that can be used to understand the patterns of change.

Software engineering literature recommends that every time a software system is changed, the type of change, the design rationale and impact should be appropriately documented [219, 257]. However, due to schedule and budget pressures, this task is often poorly resourced, with consequent inadequate design document quality [50, 97]. Another factor that contributes to this task being avoided is the lack of widespread formal education in software evolution, limited availability of appropriate tools, and few structured methods that can help developers understand evolutionary trends in their software products. To ensure that all changes are properly understood, adequately explained and fully documented, there is a need for easy to use methods that can identify these changes and highlight them, allowing developers to explain properly the changes.

Given this context, where we have an evolving product, there is a strong need for developers to understand properly the underlying growth dynamics as well as have appropriate knowledge of major changes to the design and/or architecture of a software system, beyond an appreciation of the current state. Research and studies into how software evolves is of great importance as it aids in building richer evolution
models that are much more descriptive and can be used to warn developers of significant variations in the development effort or highlight decisions that may be unusual within the historical context of a project.

1.1 Research Goals

This study aims to improve the current understanding of how software systems grow and change as they are maintained, specifically by providing models that can be used to interpret evolution of Open Source Software Systems developed using the Java programming language [108]. Two broad facets of evolution are addressed in this thesis (i) *Nature of growth* and (ii) *Nature of change*.

Our goal is driven by the motivation to understand where and how maintenance effort is focused, and to develop techniques for detecting substantial changes, identify abnormal patterns of evolution, and provide methods that can identify change-prone components. This knowledge can aid in improving the documentation of changes, and enhance the productivity of the development team by providing a deeper insight into the changes that they are making to a software system. The models can also provide information for managers and developers to objectively reflect on the project during an iteration retrospective. Additionally, the analysis techniques developed can be used to compare not just different releases of a single software system, but also the evolution of different software systems.

The primary focus of our research is towards building descriptive models of evolution in order to identify typical patterns of evolution rather than in establishing the underlying drivers of change (as in the type of maintenance activities that causes the change). Though the drivers are important, our intention is to provide guidance to developers on what can be considered normal and what would be considered abnormal. Furthermore, empirically derived models provide a baseline from which we can investigate our efforts in identifying the drivers of evolution.
1.2 Research Approach

Empirical research by its very nature relies heavily on quantitative information. Our research is based on an exploratory study of forty non-trivial and popular Java Open Source Software Systems and the results and interpretation are from an empirical software engineering perspective. The data set consists of over 1000 distinct releases encompassing an evolution history comprising approximately 55000 classes. We investigate Open Source Software Systems due to their non-restrictive licensing, ease of access, and their growing use in a wide range of projects.

Our approach involves collecting metric data by processing compiled binaries (Java class files) and analysing how these metrics change over time in order to understand both growth as well as change. Although we use the compiled builds as input for our analysis, we also make use of other artifacts such as revision logs, project documentation, and defect logs as well as the source code in order to interpret our findings and better understand any abnormal change events. For instance, if the size of the code base has doubled between two consecutive releases within a short time frame (as observable in the history), additional project documentation and messages on the discussion board often provide an insight into the rationale and motivations within the team that cannot be directly ascertained from an analysis of the binaries alone.

In order to understand the nature of growth, we construct relative and absolute frequency histograms of the various metrics and then observe how these histograms change over time using higher-order statistical techniques. This method of analysis allows us, for example, to identify if a certain set of classes is gaining complexity and volume at the expense of other classes in the software system. By analysing how developers choose to distribute functionality, we can also identify if there are common patterns across software systems and if evolutionary pressures have any impact on how developers organise software systems.

We examine the nature of change, by analyzing software at two levels of granularity: version level and class level. The change measures that we
compute at the level of a version allow us to identify classes that have been added, removed, modified and deleted between versions. Class level change measures allow us to detect the magnitude and frequency of change an individual class has undergone over its lifetime within the software system. We use the information collected to derive a set of common statistical properties, and identify if certain properties within a class cause them to be more change-prone.

1.3 Main Research Outcomes

In this thesis we address the problem of identifying, in successful software systems, where and how maintenance effort tends to be devoted. We show that maintenance effort is, in general, spent on addition of new classes with a preference to base new code on top of a small set of class that provide key services. Interestingly, these choices make the heavily used classes change-prone as they are modified to meet the needs of the new clients.

This thesis makes a number of significant contributions to the software evolution body of knowledge:

Firstly, we investigated the validity of Lehman’s Laws of software evolution related to growth and complexity within our data set, and found consistent support for the applicability of the following laws: First law Continuing Change, third law Self Regulation, fifth law Conservation of Familiarity, and the sixth law Continuing Growth. However, our analysis was not able to provide sufficient evidence to show support for the other laws.

Secondly, we investigated how software metric data distributions (as captured by a probability density function) change over time. We confirm that software metric data exhibits highly skewed distributions, and show that the use of first order statistical summary measures (such as mean and standard deviation) is ineffective when working with such data. We show that by using the Gini coefficient [91], a high-order statistical measure widely used in the field of economics, we can inter-
pret the software metrics distributions more effectively and can identify if evolutionary pressures are causing centralisation of complexity and functionality into a small set of classes.

We find that the metric distributions have a similar shape across a range of different system, and that the growth caused by evolution does not have a significant impact on the shape of these distributions. Further, these distributions are stable over long periods of time with only occasional and abrupt spikes indicating that significant changes that cause a substantial redistribution of size and complexity are rare. We also show an application of our metric data analysis technique in program comprehension, and in particular flagging the presence of machine generated code.

Thirdly, we find that the popularity of a class is not a function of its size or complexity, and that evolution typically drives these popular classes to gain additional users over time. Interestingly, we did not find a consistent and strong trend for measures of class size and complexity. That is, large and complex classes do not get bigger and more complex purely due to the process of evolution, rather, there are other contributing factors that determine which classes gain complexity and volume.

Finally, based on an analysis of how classes change, we show that, in general, code resists change and the common patterns can be summarized as follows: (a) most classes are never modified, (b) even those that are modified, are changed only a few times in their entire evolution history, (c) the probability that a class will undergo major change is very low, (d) complex classes tend to be modified more often, (e) the probability that a class will be deleted is very small, and (f) popular classes that are used heavily are more likely to be changed. We find that maintenance effort (post initial release) is in general spent on addition of new classes and interestingly, efforts to base new code on stable classes will make those classes less stable as they need to be modified to meet the needs of the new clients.
A key implication of our finding is that the Laws of Software Evolution also apply to some degree at a micro scale: “a class that is used will undergo continuing change or become progressively less useful.” Another implication of our findings is that designers need to consider with care both the internal structural complexity as well as the popularity of a class. Specifically, components that are designed for reuse, should also be designed to be flexible since they are likely to be change-prone.

1.4 Thesis Organisation

This thesis is organised into a set of chapters, followed by an Appendix. The raw metric data used in our study as well as the tools used are included in a DVD attached to the thesis.

Chapter 2 - Software Evolution provides an overview of prior research in the field of software evolution and motivates our own work.

Chapter 3 - Data Selection Methodology explains our input data selection criteria and the data corpus selected for our study. We discuss the various types of histories that can be used as an input for studying evolution of a software system and provide a rationale for the history that we select for analysis.

Chapter 4 - Measuring Evolving Software explains the metric extraction process and provides a discussion of the metrics we collect from the Java software systems and provide appropriate motivation for our choices.

Chapter 5 - Growth Dynamics deals with how size and complexity distributions change as systems evolve. We discuss an novel analysis technique that effectively summarises the distributions and discuss our findings.

Chapter 6 - Change Dynamics deals with how classes change. We present our technique for detecting change, identify typical patterns of change and provide additional interpretation to the results found in our growth analysis.


Chapter 7 - Implications outlines the implications arising from the findings described in Chapter 5 and Chapter 6.

Chapter 8 - Summary provides a summary of the thesis and presents future work possibilities. In this chapter we argue that the findings presented in the thesis can aid in building better evolution simulation models.

The Appendix collates the data tables and provides an overview of the files on the companion DVD for this thesis which has the raw metric data extracted from software systems under investigation.
Chapter 2

Software Evolution

How does software change over time? What constitutes normal change? Can we detect patterns of change that are abnormal and might be indicative of some fundamental issue in the way software is developed? These are the types of questions that research in the field of software evolution aims to answer, and our thesis makes a contribution towards this end. Over the last few decades research in this field has contributed qualitative laws [174] and insights into the nature and dynamics of this evolutionary process at various levels of granularity [41, 59, 65, 71, 85, 96, 100, 118, 127, 162, 171, 188, 194, 200, 284, 289, 289, 290, 292–294, 304]. In this chapter we present the background literature relevant for this thesis and provide motivation for our research goals.

2.1 Evolution

Evolution describes a process of change that has been observed over a surprisingly wide range of natural and man-made entities. It spans significant temporal and spacial scales from seconds to epochs and from microscopic organisms to the electricity grids that power continents. The term evolution was originally popularised within the context of biology and captures the “process of change in the properties of populations of organisms or groups of such populations, over the course of
generations” [84]. Biological evolution postulates that organisms have descended with modifications from common ancestors. As a theory it provides a strong means for interpretation and explanation of observed data. As such, this theory has been refined over a century and provides a set of mature and widely accepted processes such as natural selection and genetic drift [84]. The biological process of evolution applies to populations as opposed to an individual. However, over time the term evolution has been adopted and used in a broad range of fields to describe ongoing changes to systems as well as individual entities. Examples include the notion of stellar evolution, evolution of the World Wide Web as well as “evolution of software systems”.

Evolution, like other natural processes, requires resources and energy for it to continue. Within the context of biological and human systems (manufactured and social), evolution is an ongoing process that is directed, feedback driven, and aims to ensure that the population is well adapted to survive in the changing external environment [84]. The evolutionary process achieves this adaptation by selecting naturally occurring variations based on their fitness. The selection process is directed, while the variations that occur within the population are considered to be random. In its inherent nature this process is gradual, incremental and continuously relies on a fitness function that ensures the population’s continued survival [60].

A facet of evolution is the general tendency of entities undergoing evolution to gain a greater level of complexity over time [300]. But what is complexity? In general usage, this term characterises something with many parts that are organised or designed to work together. From this perspective, evolution drives the creation of new parts (it may also discard some parts) as well as driving how they are organised. This process adds volumetric complexity (more parts) and structural complexity (inter-connections between parts). The consequence of this increasing complexity, however, is the need for an increase in the amount of energy needed for the process to be able to sustain ongoing evolution.

Within the context of software the term evolution has been used since the 1960s to characterise growth dynamics. For example work by Halpern [112] has shown how programming systems have evolved and
Fry et. al. [82] studied how database management systems evolve. The term in relation to how a software system changes started to appear in work done by Couch [57]. Building on this foundation, Lehman [174], in his seminal work argued that E-type software (application software used in the real-world) due to their very use provide evolutionary pressures that drive change. This argument was supported by the observation that stakeholder requirements continually change, and in order to stay useful, a software system must be adapted to ensure ongoing satisfaction of the stakeholders. Unlike biological evolution which applies to a population of organisms, the term software evolution is used within the context of an individual software system. Similar to biological evolution, the process of evolution in software is directed and feedback-driven to ensure the software system is continuously adapted to satisfy the user’s requirements. However, a key distinction is that in software evolution, there is no random variation occurring within the software system (see Figure 2.1) and the term “evolution” in the context of software implies directed adaptation.
Chapter 2. Software Evolution

Although software evolution is typically used to imply a process of change to an individual software system, it is also used within the context of a product family [216], where the process involves a set of similar software systems, akin to the concept of population in biology. Though in both of these cases, there is a process of change, the object under study is quite different - a single product versus an entire product family. Further, the underlying drivers and mechanisms are also quite different. When a product family is considered, evolution is a process with some similarity to that in biological systems. For example, Nokia has a population of mobile phones and they mine functionality from a range of their models when creating new models [238]. In this scenario new phones can be seen to descend from an ancestor and market driven mechanisms of selection of functionality, cross-breeding of functionality from a number of models as well as intentional and random mutation where new ideas are tried out.

In the context of this thesis, we take an approach similar to that used by Lehman in this seminal work [174] and focus on the evolution of individual software systems as they are adapted over time to satisfy stakeholder requirements.

2.2 Software Evolution

Interestingly the term software evolution currently has no single widely accepted definition [26] and the term is used to refer to both the process of discrete, progressive, and incremental changes as well as the outcome of this process [171]. In the first perspective, the focus is on evolution as a verb (the process), and in the second perspective it is a noun (the outcome) [171].

Lehman et al. [174] describe software evolution as the dynamic behaviour of programming systems as they are maintained and enhanced over their life times. This description explicitly indicates evolution as the observable outcome of the “maintenance” activity that causes the changes, that is, the focus is on the outcome rather than the process.
Software maintenance which drives the software to change and evolve as originally proposed by Swanson [267] and later updated in ISO-14764 [126] involves the following mutually exclusive activities: (i) **Corrective** work which is undertaken to rectify identified errors, (ii) **Adaptive** work which is needed to ensure that the software can stay relevant and useful to changing needs, (iii) **Perfective** work that is done to ensure it meets new performance objectives as well as to ensure future growth, and (iv) **Preventive** work that ensures that actively corrects potential faults in the system, essentially as a risk mitigation activity. The maintenance activity, in general, is considered to take place after an initial release has been developed and delivered [257].

Though the four key activities of maintenance as identified by ISO-14764 [126] are a good starting point, Chapin [43] refines these into 12 orthogonal drivers that cause software evolution: evaluative, consultive, training, updative, reformative, adaptive, performance, preventive, groomative, enhancive, corrective, and reductive. Unlike the original ISO-14764 classification which was based on intentions, Chapin's typology is based on actual work undertaken as activities or processes, and detected as changes or lack of in: (i) the software (executable), (ii) the properties of the software (captured from code), and (iii) the customer-experienced functionality. In essence, Chapin et al. argue that in a given software system, these are the three sources that can change and evolve.

Within the context of this thesis, software evolution implies the *measurable changes between releases made to the software as it is maintained and enhanced over its life time*. Software, the unit of change, includes the executable as well as the source code. Our definition is a minor adaptation to the one proposed by Lehman [174], and reinforces the distinction between maintenance and evolution. It also explicitly focuses on the outcome of the maintenance activity and changes that can be measured from the software system using static analysis. That is, we focus only on the set of changes that can be detected without executing the software system, and without direct analysis of artifacts external to the software system, for example, product documentation. Our study focuses on the outcome from changes that are possible due
to the following drivers (as per Chapin’s typology [43]): groomative, preventive, performance, adaptive, enhancive, corrective and reductive.

Our definition of software evolution does not explicitly position it from the entire life-cycle (i.e. from concept till the time it is discontinued) of a product perspective as suggested by Rajlich [230], but it ensures that as long as there is a new release with measurable changes, then the software is considered to be evolving. However, if there are changes made via modifications to external configuration/data files, they will not be within the scope of our definition. Similarly, we also ignore changes made to the documentation, training material or other potential data sources like the development plans. Although these data sources add additional information, the most reliable source of changes to a software system is the actual executable (and source code) itself. Hence in our study of software evolution, we focus primarily on the actual artefact and use other sources to provide supporting rationale, or explanation for the changes.

Studies of Software Evolution

Studies into software evolution can be classified based on the primary entities and attributes that are used in the analysis [25]. One perspective is to collect a set of measurements from distinct releases of a software system and then analyze how these measures change over time in order to understand evolution – these are referred to as *release based studies*. The alternative perspective is to study evolution by analyzing the individual changes that are made to a software system throughout its life cycle – referred to as *change based studies*. These studies consider an individual change to be a specific change task, an action arising from a change request, or a set of modifications made to the components of a software system [25].

Release based studies are able to provide an insight into evolution from a *post-release* maintenance perspective. That is, we can observe the evolution of the releases of a software system that the stakeholders are likely to deploy and use. The assumption made by these studies
Chapter 2. Software Evolution

is that developers will create a release of a software system once it is
deemed to be relatively stable and defect-free [254]. By focusing on
how a sequence of releases of a software system evolve, the release-
based studies gain knowledge about the dynamics of change between
stable releases and more importantly have the potential to identify re-
leases with significant changes (compared to a previous release). The
assumption that developers formally release only a stable build allows
release based studies to identify patterns of evolution across multiple
software systems (since they compare what developers consider stable
releases across different systems) [254].

Change based studies, on the other hand, view evolution as the ag-
cgregate outcome of a number of individual changes over the entire life
{
} cycle [25]. That is, they primarily analyze information generated dur-
ing the development of a release. Due to the nature of information that
they focus on, change based studies tend to provide an insight into the
process of evolution that is comparatively more developer centric. Al-
though change based studies can also be used to determine changes
from the end-user perspective, additional information about releases
that have been deployed for customers to use has to be taken into con-
sideration during analysis.

Though software evolution can be studied from both a release based
as well as the change based perspective, most of the studies in the lit-
erature have been based on an analysis of individual changes [139].
A recent survey paper by Kagdi et al. [139] reports on the result of
an investigation into the various approaches used for mining software
repositories in the context of software evolution. Kadgi et al. show that
most studies of evolution tend to rely on individual changes as recorded
in the logs generated and maintained by configuration/defect manage-
ment systems (60 out of the 80 papers that they studied). Though a
specific reason for the preference towards studying these change logs is
not provided in the literature, it is potentially because the logs permit
an analysis of software systems independent of the programming lan-
guage, and the data is easily accessible directly from the tools typically
used by the development team (e.g. CVS logs).
Chapter 2. Software Evolution

A limitation of relying on individual changes is that the change log data needs to be carefully processed [86] in order to identify if the changes recorded are related to aspects of software system under study (for instance, source code), and also to ensure that the changes are significant (for example, minor edits in the code comments may need to be eliminated if the emphasis of a study is to understand how developers adapt the actual functional source code as they evolve the system). Another constraint that studies relying of change logs face is raised by Chen et al. [44] who found that developers in some open source projects did not properly record all of the changes. In their study, Chen et al. highlight that in two out of the three systems studied, over 60% of the changes were not recorded, and as a consequence, the information provided in the change logs cannot be considered to be representative of all the changes that take place within a software system. The significant drawback of change based studies is their heavy reliance on developers providing consistent and regular information about individual changes. There is currently no evidence that shows that developers record individual changes carefully. Furthermore, the definition of an individual change is likely to vary from developer to developer, as well as from project to project.

In our study, we focus on how software evolves post-release both in terms of growth and changes between the releases that developers have made available to end-users. We focus on releases because an understanding of evolution from this perspective is of greater value to managers and developers as any post-release change, in general, has a greater impact on the end users [220]. Furthermore, existing release based studies have mainly investigated very few software systems (typically less then 20), including the seminal work by Lehman [174] which investigated only one large software system. The restriction on small data sets was potentially unavoidable in earlier work [85,148,284] due to the reliance on commercial software systems which have legal restrictions that make it challenging to investigate, and to replicate the experiments. The wide-spread and increasing availability of open source software systems over the past decade has allowed researchers to study distinct releases of a larger number of software systems in order to understand evolution. However, even these stud-
Chapter 2. Software Evolution

ies [59, 100, 101, 103, 127, 193, 204, 217, 239, 254, 277, 304, 310] focused on a few popular and large software systems (for example, the Linux operating system or the Eclipse IDE).

Interestingly, evolution studies that have consistently investigated many different software systems (in a single study) are change based studies. Change based studies tend to use the revision logs generated and maintained by the configuration management tools rather than collecting data from individual releases in order to analyze the dynamics within evolving software systems [25]. A few notable large change based studies are Koch et al. [152, 153] who studied 8621 software systems, Tabernero et al. [118] who investigated evolution in 3821 software systems and Capiluppi et al. [39] who analysed 406 projects.

Given the small number of systems that are typically investigated in release based evolution studies, there is a need for a comparatively larger longitudinal release based software evolution study to confirm findings of previous studies still hold, to increase the generalizability of the findings, and to improve the strength of the conclusions. Even though previous release based studies [59, 100, 101, 103, 127, 193, 204, 217, 239, 254, 277, 304, 310] have investigated a range of different software systems, a general limitation is that there has been no single study that has attempted to analyze a significant set of software systems. Our work fills this gap and involves a release based study of forty software systems comprising 1057 releases. The focus on a comparatively larger set of software systems adds to the existing body of knowledge since our results have additional statistical strength than studies that investigated only a few software systems. Our data set selection criteria and the method used to extract information is discussed in Chapter 3 and Chapter 4, respectively.

2.3 The Laws of Software Evolution

The laws of software evolution are a set of empirically derived generalisations that were originally proposed in a seminal work by Lehman and Belady [168]. Five laws were initially defined [168] and later ex-
## Chapter 2. Software Evolution

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Continuing Change</td>
<td>An E-type system must be continually adapted, else it becomes progressively less satisfactory in use</td>
</tr>
<tr>
<td>2</td>
<td>Increasing Complexity</td>
<td>As an E-type system is changed its complexity increases and becomes more difficult to evolve unless work is done to maintain or reduce the complexity</td>
</tr>
<tr>
<td>3</td>
<td>Self Regulation</td>
<td>Global E-type system evolution is feedback regulated</td>
</tr>
<tr>
<td>4</td>
<td>Conservation of Stability</td>
<td>The work rate of an organisation evolving an E-type software system tend to be constant over the operational lifetime of that system or phases of that lifetime</td>
</tr>
<tr>
<td>5</td>
<td>Conservation of Familiarity</td>
<td>In general, the incremental growth (growth rate trend) of E-type systems is constrained by the need to maintain familiarity</td>
</tr>
<tr>
<td>6</td>
<td>Continuing Growth</td>
<td>The functional capability of E-type systems must be continually enhanced to maintain user satisfaction over system lifetime</td>
</tr>
<tr>
<td>7</td>
<td>Declining Quality</td>
<td>Unless rigorously adapted and evolved to take into account changes in the operational environment, the quality of an E-type system will appear to be declining</td>
</tr>
<tr>
<td>8</td>
<td>Feedback System</td>
<td>E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems</td>
</tr>
</tbody>
</table>

Table 2.1: The Laws of Software Evolution [175]

Extended into eight laws (See Table 2.1) [175]. These laws are based on a number of observations of size and complexity growth in a large and long lived software system. Lehman and his colleagues in their initial work discovered [168] and refined [171, 175] the laws of evolution (which provide a broad description of what to expect), in part, from direct observations of system size growth (measured as number of modules) as well as by analysing the magnitude of changes to the modules. The initial set of Five laws were based on the study of evolution of one large mainframe software system. These five laws were later refined, extended and supported by a series of case studies by Lehman and his colleagues [171, 283, 284].
These empirical generalisations have been termed \textit{laws} because they capture and relate to mechanisms that are largely independent of technology and process detail. Essentially these laws are qualitative descriptors of behaviour similar to laws from social science research and are not as deterministic or specific as those identified in natural sciences [198].

The \textit{Laws of Software Evolution} (Table 2.1), state that regardless of domain, size, or complexity, real-world software systems evolve as they are continually adapted, grow in size, become more complex, and require additional resources to preserve and simplify their structure. In other words, the laws suggest that as software systems evolve they become increasingly harder to modify unless explicit steps are taken to improve maintainability [175].

The laws broadly describe \textit{general} characteristics of the natural incremental transformations evolving software systems experience over time and the way the laws have been described reflect the social context within which software systems are constructed [198]. Furthermore, these laws also suggest that at the global level the evolutionary behaviour is systemic, feedback driven and not under the direct control of an individual developer [171].

The laws capture the key drivers and characteristics of software evolution, are tightly interrelated, and capture both the change as well as the context within which this change takes place [168, 170, 175]. The first law (\textit{Continuing Change}) summarises the observation that software will undergo regular and ongoing changes during its life-time in order to stay useful to the users. These changes are driven by external pressures, causing growth in the software system (captured as \textit{Continuing Growth} by the sixth law) and in general, this increase in size also causes a corresponding increase in the complexity of the software structure (captured by the second law as \textit{Increasing Complexity}). Interestingly, the process of evolution is triggered when the user perceives a decrease in quality (captured as \textit{Declining Quality} in the seventh law). Additionally, the laws also state that the changes take place within an environment that forces stability and a rate of change that permits the
organisation to keep up with the changes (captured by the fourth and fifth laws of Conservation of Organisational Stability and Conservation of Familiarity respectively). The laws suggest that in order to maintain the stability and familiarity within certain boundaries, the evolutionary process is feedback regulated (third law – Self Regulation), and that the feedback takes place at multiple levels from a number of different perspectives (eighth law of Feedback System).

The Laws of Software Evolution are positioned as general laws [171, 175] even though there is support for the validity of only some of the laws [41, 59, 85, 171, 188, 192, 200]. However, there is increasing evidence [39, 100, 101, 103, 119, 120, 127, 153, 217, 239, 277, 306, 310] to suggest that these laws are not applicable in many open source software systems and hence have to be carefully interpreted (we elaborate on these studies in the next section). A recent survey paper by Ramil et al. [192] studied the literature and argues that there is consistent support for the first law (Continuing Change) as well as the sixth law (Continuing Growth), but no broad support exists for the other laws across different empirical studies of open source software systems.

From a practical perspective, the applicability of the laws is limited by their inability to provide direct quantitative measures or methods for interpreting the changes that take place as software evolves [198]. Whilst the laws of evolution continue to offer valuable insight into evolutionary behaviour (effect), they do not completely explain the underlying drivers or provide a behavioural model of change (the why) [169]. Despite many studies into software evolution, a widely accepted cause and effect relationship has not yet been identified, potentially due to the large number of inter-related variables involved and the intensely humanistic nature of software development that adds social aspects to the inherently complex technical aspects [188, 192, 198].

In spite of their limitations, the laws of evolution have provided a consistent reference point since their formulation for many studies of software evolution, and therefore we investigate the validity and applicability of these laws within our data set. Furthermore, our research approach analyzes the distribution of growth and change (discussed in
Chapter 2. Software Evolution

Chapter 5 and Chapter 6) rather than observe the overall growth trend in certain measures which is the technique employed by many earlier studies [39, 41, 59, 85, 100, 101, 103, 119, 120, 127, 153, 171, 175, 192, 200, 217, 239, 277, 306, 310] (discussed further in the next section – Section 2.4). As a consequence our study can offer a different insight into the laws, as well as the dynamics of change within open source software systems.

2.4 Studies of Growth

The Laws of Software Evolution, as well as many studies over the last few decades have consistently shown that evolving software systems tend to grow in size [188, 192]. But, what is the nature of this growth? In this section we summarise the current understanding of the nature of growth in evolving software systems. In particular, we focus heavily on studies of growth in open source software systems as they are more appropriate for the scope of this thesis.

In studies of software evolution, the observed growth dynamics are of interest as they can provide some insight into the underlying evolutionary process [283]. In particular, it is interesting to know if growth is smooth and consistent, or if a software system exhibits an erratic pattern in its growth. For instance, managers can use this knowledge to undertake a more detailed review of the project if development effort was consistent, but the resulting software size growth was erratic. Although the observed changes do not directly reveal the underlying cause, it can guide the review team by providing a better temporal perspective which can help them arrive at the likely drivers more efficiently. Additionally, studies into growth dynamics also establish what can be considered typical and hence provide a reference point for comparisons. Lehman and his colleagues in their initial work [168] discovered and refined [171, 175] the laws of evolution (which provide a broad description of the dynamics of software evolution), in part, from direct observations of size growth in long lived commercial software systems.
Growth rate

The laws of evolution state that software will grow as it is adapted to meet the changing user needs. However, what is the typical growth rate that we can expect to see in a software system? A consistent and interesting observation captured in early studies [85, 168, 175, 283] into software evolution was that the typical rate of growth is sub-linear (see Figure 2.2). That is, the rate of growth decreases over time. The laws of software evolution suggest that this is to be expected in evolving software since complexity increases (second law), and average effort is consistent (Fourth Law). The argument that is extended to support the sub-linear growth expectation is that in evolving software, the increasing complexity forces developers to allocate some of the development effort into managing complexity rather than towards adding new functionality [283] resulting in a sub-linear growth rate.

A model that captures this relationship between complexity and growth rate is Turski’s Inverse Square Model [283, 284]. Turski’s model (see Equation 2.4.1) is built around the assumption that the system size growth as measured in terms of number of source modules is inversely proportional to its complexity (measured as a square of the size to capture the number of intermodule interaction patterns) and has been shown to fit the data for a large long-lived software system [283].
Turski’s Inverse Square model [283] is formulated with system size $S$ at release $i$ ($S_i$) and constant effort $E$. Complexity of software is the square of the size at previous version ($S_{i-1}^2$).

$$S_i = \frac{E}{S_{i-1}^2} + S_{i-1}$$  \hspace{1cm} (2.4.1)

Beyond the work by Turski [283, 284], the sub-linear growth rate observation is also supported by a number of different case studies [41, 59, 85, 171, 175, 188, 192, 200, 217] that built models based on regression techniques. The increasing availability and acceptance of Open Source Software Systems has allowed researchers to undertake comparatively larger studies in order to understand growth as well as other aspects of evolution [39, 100, 103, 127, 217, 306]. Interestingly, it is these studies that have initially provided a range of conflicting results, some studies [17, 129, 153] found that growth typically tends to be sub-linear supporting the appropriateness of Lehman’s laws, but others [101, 119, 120, 127, 153, 239] have observed linear as well as super-linear growth rates suggesting that the growth expectations implied by Lehman’s laws of evolution are not universal.

Godfrey and his colleagues [101] were one of the first to question the validity of Lehman’s laws in the context of Open Source Software Systems. In their study they observed growth to be super-linear in certain sub-systems of Linux (specifically the driver sub-system in their study), suggesting that the increasing complexity and sub-linear growth rate expectation of Lehman’s laws do not universally hold. This observation of super-linearity was later confirmed by Succi et al. [264], González-Barahona et al. [105, 106] and more recently by Israeli et al. [127].

In contrast to these multiple findings on super-linear growth rates, Izurieta et al. [129] found no evidence of super linear growth rate in FreeBSD and the Linux kernels. Godfrey et al. [100] found that Fetchmail (e-mail retrieval and forwarding system), X-Windows (a Window manager) and the gcc compiler exhibit near linear growth while the Pine e-mail client had a sub-linear growth rate. Additional evidence from a study by Paulson et al. [217] suggests that the Linux kernel, Apache
Chapter 2. Software Evolution

Web server and gcc compiler all showed only linear growth. Robles et al. [239] analyzed 18 different open source software systems and found that sub-linear and linear growth rates to be the dominant trend with only two systems (Linux and KDE) fitting a super-linear growth trend. Mens et al. [192] in a study of the evolution of the Eclipse IDE observed super-linear growth in the number of plug-ins, while the core platform exhibited a linear growth rate. Koch [152], in an extensive change based study of over 4000 different software systems mined from Sourceforge (a popular open source software repository), found linear and sub-linear growth rates to be common, while only a few systems exhibited super-linear growth rate. Though, Koch et al. undertook a change based study by analysing the source code control logs, they reconstruct size measures in order to analyze the growth rates. More recently, Thomas et al. [277] investigated the rate of growth in Linux kernel and found a linear growth rate.

Researchers [101, 127, 153, 192, 200] that have observed the super-linear growth rate argue that the underlying structure and organisation of a system has an impact on the evolutionary growth potential and that modular architectures can support super-linear growth rates. They suggest that these modular architectures can support an increasing number of developers, allowing them to make contributions in parallel without a corresponding amplification of the communication overhead [153].

From an alternate perspective, in systems with a plug-in architectural style, evolutionary growth can be seen as adding volumetric complexity without a corresponding increase in the cognitive complexity [101]. This is the case because developers do not need to gain an understanding of all of the plug-ins in the system, rather they need to understand the core framework and the plug-in interface in order to add new functionality. For instance, in the case of Linux, the super-linear growth was attributed to a rapid growth in the number of device drivers [101], most of which tend to adhere to a standard and relatively stable functional interface, allowing multiple development teams to contribute without increasing the communication overhead and more importantly without adding defects directly into the rest of the system. Similarly, the expo-
nential growth in the number of plug-ins for the Eclipse platform [192] is similar to that of the driver sub-system in Linux and shows that certain architectural styles can allow the overall software systems to grow at super-linear rates, suggesting limitations to Lehman’s laws.

Studies that found super-linear growth rates [101, 119, 120, 127, 153, 239] show that it is possible to manage the increase in volumetric complexity and the consequent structural complexity. The implication of these studies is that certain architectural choices made early in the life cycle can have an impact on the growth rate, and a certain level of structural complexity can be sustained without a corresponding investment of development effort (in contrast to the expectations of the laws of software evolution).

A consistent method that is generally applied by earlier studies of growth has been to observe how certain system wide measures change over time (for example, Number of modules). These observations have then typically been interpreted within the context of Lehman’s laws of evolution in order to understand growth dynamics within evolving software systems. Though these previous studies have improved our understanding of how software evolves, there is limited knowledge with respect to how this growth is distributed among the various abstractions of a software system.

An early study that has provided some data about the distribution of growth is the one undertaken by Gall et al. [85] that suggests that different modules grow at different rates. This observation is also confirmed by Barry et al. [17]. Although these studies highlight that growth rates can differ across modules, they do not discuss in depth how the growth is distributed and what the impact of this distribution is on the overall evolution of the software that they study. More recently, Israeli et al. [127] investigated the Linux kernel and identified that average complexity is decreasing. The interesting aspect of the study by Israeli et al. was that they note that the reduction of the average complexity was a result of developers adding more functions with lower relative complexity. However, all of these studies have focused on individual systems and on a small set of metrics, and hence there is a gap in our
Segmented Growth

The common theme in studies of growth [101, 127, 152, 153, 192, 217, 239] is that they focus on the growth over the entire evolution history and as a consequence attach only a single growth rate to software systems. That is, they tend to classify the size growth of a system to be one of the following: sub-linear, linear, or super-linear. However, when a more fine-grained analysis was performed, software systems undergoing evolution have been shown to exhibit a segmented and uneven growth pattern. That is, the software system can grow at different rates at different time periods [6, 120, 123, 175, 256, 305], and also that some modules can grow much faster than others [17, 85]. This segmented growth pattern is illustrated in Figure 2.3. The data in the figure is from one of the software systems that we analyse in our study and highlights the need for analyzing growth from different perspectives.

Figure 2.3: Illustration of the segmented growth in the Groovy language compiler. The overall growth rate appears to be super-linear, with two distinct sub-linear segments.
Chapter 2. Software Evolution

The observation of segmented growth has been used to suggest that developers periodically restructure and reorganise the code base potentially causing a temporary reduction in size and complexity followed by a period of renewed growth [175]. An example of this segmented growth has been captured by Capiluppi et al. [38–41] in a sequence of studies. They presented evidence that shows that open source software systems tend to have segmented growth where each segment may have a different growth rate. For instance, Capiluppi et al. note that Gaim (an internet chat software that was investigated in their study) grows at a super-linear rate early in its life cycle, with a large gap in development followed by a linear growth rate.

The segmented growth pattern has also been confirmed by Smith et al. [256] and by Wu et al. [304,305]. Smith et al. [256] studied 25 open source systems developed in C/C++ and showed that growth rates are not consistent during the evolution of a software system and that they can change. More recently, Wu et al. [304,305] presented evidence of a punctuated growth in open source software system based on a study of 3 systems (including Linux). Wu et al. observed that developers work in periodic bursts of activity, where intensive effort goes into creating a major release followed by a less active period where minor defects are corrected. Additionally, work done by Hsi et al. [123] has also shown how the evolutionary drivers result in asymmetric and clumpy growth.

Summary

Studies of software evolution that have investigated the phenomenon of growth have shown that the rate of growth can be super-linear, linear or sub-linear. Furthermore, since this growth has been shown to be segmented, there are limitations in the value offered by an understanding of the overall growth rate. Additionally, the lack of consistency with respect to the observed growth rate in the studies of evolution [101,127,152,153,192,217,239] shows that there are limitations within the explanation of the dynamics as postulated by the laws of software evolution [175]. Specifically, there is evidence to suggest that the complexity that arises due to evolution does not necessarily create
Chapter 2. Software Evolution

a drag on the rate of growth [101]. Another aspect is that the studies identifying linear and super-linear growth rates show that the parameters considered in simple growth models (like Turski’s [283,284], or by models developed using regression techniques) are not sufficient when attempting to model and understand growth. That is, though there is some relationship between complexity and growth, there may be other aspects that influence the growth of a software system.

Current models and methods of understanding of growth in evolving software [118, 119, 127, 153, 168, 175, 200, 217, 239, 277, 284, 305, 310] have allowed for inferences to be drawn about certain attributes of the software system, for instance, regarding the architecture [100, 101, 127, 153, 192, 200, 290, 294], complexity and its impact on the effort [118, 168, 284]. However, an inherent limitation of these models is that they do not provide any direct insight into where growth takes place. In particular, we cannot assess the impact of evolution on the underlying distribution of size and complexity among the various classes. Such an analysis is needed in order to answer questions such as “do developers tend to evenly distribute complexity as systems get bigger?”, and “do large and complex classes get bigger over time?”. These are questions of more than passing interest since by understanding what typical and successful software evolution looks like, we can identify anomalous situations and take action earlier than might otherwise be possible. Information gained from an analysis of the distribution of growth will also show if there are consistent boundaries within which a software design structure exists.

In particular a key gap in our understanding of growth arises because previous studies have, in general, analyzed the changes to the aggregate measures (for example, the growth of total SLOC, or the total number of classes) rather than how these measures are distributed within the abstractions of a software system. This additional detail is necessary in order to gain an insight into the decisions that developers make. That is, when developers add, modify and extend existing classes, do they gradually centralise functionality and complexity, or does this process spread out the functionality and complexity across classes? This knowledge of the typical patterns of growth and its distributions can be
used to guide development teams to identify abnormal growth patterns. Further, we can also verify if developers follow what are considered good software engineering practice and avoid god classes [237] as software evolves. This form of analysis is also helpful in early identification of abnormal and unusual changes. This is necessary because research into software reliability shows us that structurally complex parts of the software tend to contain a higher rate of defects [20,205,281,317] and early identification of parts that are gaining in complexity will allow us to take corrective actions sooner.

In our study we aim to close these gap in our understanding and observe evolution from a different perspective. Rather than use global size growth models to infer support for laws and understand the nature of evolution, we study software evolution by observing how the distribution of size and complexity changes over time. More specifically, we undertake a longitudinal study that constructs probability density functions of different metrics collected from the software systems, and builds a descriptive model of evolution by observing how these metric distributions change over time.

2.5 Studies of Change

It is a widely accepted that software systems that are in active use need to be changed at some or many stages of its life in order to stay useful [175,188,192]. Given that change is the one inherent constant of an active and used software system, the key to a successful software evolution approach lies, however, not only in adapting a system to support the new requirements [87], but also in understanding the nature and the dynamics of change. Managing and reducing the costs and risks of changes to software that arise as part of the maintenance process are important goals for both research and the practice of software engineering [25,220].

The laws of software evolution, as well as a number of other studies, have consistently suggested that evolving software will grow and undergo change as it is adapted [175,188,192,293]. Since software
growth is dependent on change, a proper understanding of the nature and type of changes that a software system undergoes is important to understand the growth dynamics. For instance, the growth can be caused by developers adding new abstractions, removing existing code, or modifying some existing abstractions. Furthermore, studies of change can help address questions such as – Are there any common and recurring patterns of changes? Do developers tend to grow software by creating new abstractions?, or Do they prefer to modify and extend existing abstractions?, and Do certain attributes make a class more change-prone?. Knowledge gained from these studies of change help us understand growth better [102], improve development plans by providing information about the typical patterns and impact of change [281], and identify and inform developers of the changes that they made in a specific time interval to help them reflect on the development process better.

**Detecting Change**

The first step in understanding change is to identify the *unit of change*, specifically, the entity that is undergoing change [17]. In this thesis, we take a class to be the unit of change. We study classes since they are the primary organisational abstractions in object-oriented software systems [185, 186]. A class contains both data as well as methods and is the natural abstraction that developers use when designing, constructing and modifying an object-oriented software system. Although, change can be studied at a lower level of abstractions such as a method, within object-oriented software systems, a method is considered to be a part of a class and hence any change to methods are better understood within the context of a class. Similarly, focusing on a higher-level abstraction such as a package or a component provides a coarse-grained view of change than can be obtained by observing changes in classes.

The next step in understanding change requires the detection of change in a class. There are two approaches that are typically used to detect change in a class. The first, and more widely used method is to identify change by analysing transaction logs (*e.g.* CVS logs, issue logs) created
Chapter 2. Software Evolution

during the development of a specific release [25]. This is the analysis technique preferred by change based studies (as discussed in Section 2.2). Though this approach is widely used in the literature [138], there are significant limitations to this method of detecting change. The transaction logs that are used as input for detecting change do not directly record the nature of change. Specifically, they do not record if the modification altered the functional semantics of the program, or if the change was purely cosmetic [80]. For instance, widely used source code control systems such as CVS and SVN record changes by string comparison between two text files and tend to identify the lines added, removed and modified. However, these tools do not check if the change impacts the actual semantics of the program. That is, these tools treat the following change actions identically – comment addition, source code reformatting, and removing a method from a class. This inability of the current generation of popular tools of recording the type of change is a significant limitation if a study purely relies on data generated by these tools to detect change. The limitation arises because changes to the functional aspects of a class have a greater impact in maintenance as they have the potential to impact other classes [281], and as a consequence have a higher risk profile.

The second approach to detecting change analyses the actual class (or a program file) at two different points in time in order to determine if it has changed. This approach, referred to as origin analysis in the literature [102,282] is comparatively more complex, but provides a more accurate reflection of the changes since there is no reliance on reconstructing change information by analysing an external transaction log. Detecting change between two releases of a class has been an area of study for many years [242,282] and different methods to assist in the detection of the change have been developed [5,22,72,80,135,143,155,243,294]. In this thesis, we detect changes by analysing classes in two consecutive releases as it enables us to focus on changes to the functional aspects of a program. A more detailed discussion of these techniques, their strengths and limitations, as well as our own method for detecting the change is presented in Chapter 6.
**Dimensions of Change**

Change is defined\(^1\) as a process of transformation — *to make the form different from what it is*. Some of the earliest studies that attempted to investigate and understand the nature of change were undertaken by Lehman *et al.* [168, 175] as part of the work that developed the Laws of Software Evolution. Lehman *et al.* were able to establish that existing software was being adapted based on direct observation of size growth, and confirmation from the development team that the software was being adapted to meet changing requirements. Based on the work by Lehman *et al.*, Gall *et al.* [85] observed change by measuring the number of program files that were added, removed and modified in each release by analysing 20 releases of a single commercial software system. In this study, Gall *et al.* showed that some modules changed more than others and also that the rate of change, in general, decreased over time. More recent studies have focused on observing changes by measuring the number of functions that were modified, and also by observing how the complexity of functions changed over time [217, 291, 310]. A common finding in these studies of change is that they confirm the applicability of the First Law of Software Evolution – *Continuing Change*. A recent survey paper by Ramil *et al.* [192] that summarises empirical studies of evolution within the context of open source software evolution also suggests that there is consistent support for the first law.

Early work of Lehman *et al.*, as well as following studies into software evolution (as discussed in previous sections) have consistently found that software is adapted as it evolves. However, they have not focused on a more detailed analysis of the various dimensions of change. A method to understand change more effectively was proposed by Barry *et al.* [17] where they argue that volatility within software has three distinct dimensions from which it can be studied: *amplitude* (size of change), *periodicity* (frequency of change), and *dispersion* (consistency of change). Amplitude measures the size of modification and a number of different approaches can be applied to determine this amplitude. An example of a method used to determine the amplitude of change in a file (at two instances in time) can be measured as the sum of the lines

---

\(^1\)Oxford American Dictionary. 2005
added, removed and modified [85, 217]. Periodicity measures the regularity at which a system or an abstraction is modified. For instance, this measure is required to determine how often a file is modified over the evolution history. The expectation is that if a large part of the code base is modified frequently, then the software is highly volatile and may require corrective action. Finally, the measure of dispersion aims to identify if there is a consistent pattern to the change. This measure is motivated by the assumption that consistency allows managers to anticipate how much of the code base may change in the next version and hence can allocate resources appropriately. The dispersion measure can be applied to determine the consistency of the size of change, as well as the consistency in the frequency of change. In this thesis, we study change against these three dimensions. A discussion our approach to compute change against these three dimensions is presented in Chapter 6.

Studies of change that investigate these dimensions are able to provide us with a baseline on what to expect in evolving software. Specifically, we can identify periods of normal and abnormal change. Though an understanding of these dimensions of change is useful [17], we found comparatively few studies in the literature that have focused on a quantitative analysis of these dimensions of change. Furthermore, most studies investigated only a few software systems (typically under 10), impacting on the strength of their findings.

An early study that presents observations from an investigation of the frequency of change was undertaken by Kemerer et al. [147] who studied the profile of software maintenance in five business systems at the granularity of modules. They concluded that very few modules change frequently, and later extended this study by identifying that the modules that did change can be considered to be strategic [148] (in terms of business functionality offered). Both of these studies inferred change by analysing defect logs generated during the development of a commercial non-object oriented software system. This study was not able to establish the typical size of change, or the consistency of the change.
More recent work by Purushothaman et al. [228] has provided some insight into the size of change. Purushotham et al., based on an analysis of the source code revision control system logs conclude that most of the changes are very small based on the observation that 95% of the changes required a modification of less than 50 lines of code. A key weakness of this study is that it was based on a single large commercial software system, and hence lacks sufficient generalizability. Another study that investigated the size of change was undertaken by Mockus et al. [201] who observed that the size of change was very small for defect corrections. The study by Mockus et al. however does not provide any further insight into the dimensions of change beyond defect corrections.

Wu et al. [304,305] in a release-based study investigated the nature of change in 3 open-source software systems. They found that projects alternate between periods of localised small incremental changes and periods of deep architectural changes (which impacted a large number of modules). The studies by Wu et al. focused on a comparatively small data set, and investigated only architectural changes by analysing how incoming and outgoing dependencies for each object file change over time.

Anton et al. [3,4] investigated change in the set of functionality offered by the telephone network to end-users over a 50 year period. Anton et al. showed that different change drivers apply at different times and that functional evolution takes place in discrete bursts followed by gradual enhancement giving rise to a “punctuated equilibrium” pattern [14]. Interestingly, this is the pattern that Wu et al. also identified in their work even though the granularity and perspectives of these two studies are vastly different. Although the study by Anton et al. is interesting, their findings are helpful to review and understand long-term strategy rather than in directly helping developers understand the nature of changes within the software system at a local level over a relatively shorter time interval.

A different perspective to understanding dimensions of change in the context of object-oriented software systems has been driven by researchers
that focused on visualising software evolution [65, 95]. A simple, yet powerful method is the Evolution Matrix proposed by Lanza et al. [163] as a means to visualize the evolution, with the emphasis on revealing patterns of change. This method has the advantage of being able to provide a good overview for developers when they retrospectively audit the evolution of a project in order to gain a better understanding of the history and make improvements for the future. More recent work in evolution visualization [1, 164, 298] has aimed at highlighting changes to the design structure of a software system based on thresholds for metric values at various levels of abstraction (for instance, class or package level). Interestingly, research into visualization of evolution has focused on improving the quality and quantity of information that is communicated to the users. However, these visualizations can be improved if they have additional statistical information about the change patterns.

Change-prone Classes and Co-Change

In the context of object-oriented software systems, the aspect of change that has been investigated in considerable depth within the literature is identification of attributes that make a class change-prone, and detecting groups of classes that change together. The motivation for understanding change-prone classes is to improve the design of a software system, and minimise the need for change since modifications are considered to be risky and potentially defect inducing [205]. These studies into change-prone classes have identified some common aspects and have provided consistent evidence showing that structurally complex and large classes tend to undergo more changes [20, 28, 31, 63, 130, 177, 213, 214, 261, 266, 314, 318], and classes that changed recently are likely to undergo modifications in the near future [92, 93]. Studies that focused on change-prone classes do have some weaknesses. These studies have, in general, investigated only a few software systems and different studies have used slightly different methods for detecting change. However, given the recurring confirmation across multiple studies, the expectation that structurally complex classes will undergo more changes can be considered to be a strong possibility. Although there is considerable agreement that complexity and change are related,
a larger scale study can help improve the strength of this expectation and can provide a more robust model that captures the relationship between complexity and change.

Interestingly, these studies of change-prone classes have measured complexity within an individual class, typically using complexity measures proposed by Chidamber and Kemerer [46], or variations of these measures [116, 165]. For example, a commonly used measure of class complexity is the WMC (Weighted Method Count) with the McCabe Cyclomatic Complexity measure [190] used as the weight. Though the complexity of a class has been used to determine change-proneness, there is a gap in our understanding of the relationship between change-proneness and a class gaining new dependents. This aspect has not been properly investigated since the measures of a class complexity focus mainly on the set of classes that a particular class depends upon, rather than the set of classes that it provides services to. A recent change-based study by Geipel et al. [89], however, does present some evidence of a relationship between class dependencies and change propagation. Similarly, Sangal et al. [245] also show that a high concentration of dependencies acts as a propagator of change. Although, Giepel et al. and Sangal et al. suggest that a class with a high concentration of dependencies will propagate change, we do not fully understand the likelihood of change in a class that acts as a service provider as it gains new dependencies.

Another arc in the study of change has been the area of understanding co-change, where the expectation is that certain groups of classes or modules change together [85] because developers tend to group and construct related features together. This expectation is supported by research undertaken by Hassan and Holt who analyzed many Open Source projects and concluded that historical co-change is a better predictor of change propagation [113]. This observation is also supported by Zimmerman et al. [319, 320] which led to the development of tools [315] that can guide developers to consider a group of classes when they modify one class. These studies into co-change have been instrumental in developing tools that can help developers consider the impact of a change by exposing a wider ripple impact than is visible to
Chapter 2. Software Evolution

the compiler (via a static dependency analysis). However, these studies have relied on an analysis of revision logs and hence provide an understanding of changes during the construction of a release. There is still a gap in our understanding of post-release change, specifically in terms of statistical properties against the various dimensions of change.

Summary

Studies of change have a general theme. They have primarily used logs generated by the tools used in software development [138], and the typical focus has been mostly on establishing attributes within classes that make them change-prone [20, 28, 31, 63, 130, 177, 213, 214, 261, 266, 314, 318]. Also, these studies have arrived at their conclusions using relatively small number of software systems, and the emphasis has been on understanding fine-grained changes during the construction of a release rather than post-release.

There is currently a gap in our understanding of post-release changes in object-oriented software systems, specifically since existing studies capture only broad patterns of change based on an analysis of small data sets [4, 305]. Although previous studies suggest that change follows a punctuated equilibrium pattern with the size of change being small, they do not provide a statistical model of change. For instance, we currently do not have models that can help estimate the proportion of classes that are likely to be modified if the developers plan on adding a set of new classes. Additionally, developers also need to know where they can expect the changes within the context of their software system, and the magnitude of these modifications in order to take proactive steps and mitigate any potential risks arising from these changes. This gap can be addressed by creating descriptive statistical models of change as these models can assist in developing tools that highlight and monitor evolution prone parts of a system as well as support effort estimation activities. An additional gap in current studies is that, in general, they do not establish thresholds that can be used to flag potentially significant and systemic changes within an object-oriented software system.
In our study of change we aim to address these gaps by focusing our effort towards developing statistical models that can help establish normal and unusual patterns of change. We also use these models to understand better how evolving software systems grow, in particular, we can identify if growth is achieved by creating new abstractions, or if existing abstractions are modified and extended.

2.6 Research Questions

Evolution in software has been a field that has been investigated over the past few decades, with a heavier emphasis on object-oriented software systems over the past decade. These studies consistently establish that evolution causes growth as software is adapted to meet changing user requirements. In this chapter, we presented a summary of the key studies related to our focus areas – growth and change. A common aspect across much of the previous work is that the studies have focused on a few software systems, used different input data sources in their investigation, and the abstractions under study have not been consistent across these studies. Furthermore, the current understanding of growth has been established primarily by studies of aggregate system level size and complexity growth rather than by how this growth is distributed across the various parts of the software system. The focus of studies on change has been on identification of attributes that can make an entity change-prone, with a significant emphasis on changes during the construction of a release rather than post-release.

In order to address the gaps identified, we framed a set of research questions related to growth and change. As indicated earlier, our intention is to study growth in terms of how it is distributed across the various classes within an object-oriented software system. We investigate change as a means to better understand growth as well as to how and where the maintenance effort is focused by the developers as they modify, add and remove classes.
Chapter 2. Software Evolution

The questions related to growth that we address in this thesis are:

- What is the nature of distribution of size and complexity measures?
- How does the profile and shape of this distribution change as software systems evolve?
- Do large and complex classes become bigger and more complex as software systems evolve?

The questions related to change that we address in this thesis are:

- What is the likelihood that a class will change from a given version to the next?
- How is modification frequency distributed for classes that change?
- What is the distribution of the size of change? Are most modifications minor adjustments, or substantive modifications?
- Does complexity make a class change-prone?

Our study address the gaps in the current literature by investigating the evolution in forty non-trivial object-oriented Java software systems. The larger data set, as well as the consistent measures that we apply across 1057 releases under investigation in our study increases the strength of our conclusions. We analyse growth across a range of different measures and focus on how growth is distributed across the various classes. Furthermore, our study focuses on understanding post-release changes which have a relatively higher risk profile.

The next chapter (Chapter 3) presents the input data set, our selection criteria and the aspects that we investigate. Chapter 4 (Measuring Java Software) follows with a discussion of the metric extraction process and defines the metrics we collect from the Java software systems. The key findings of this thesis are presented in Chapter 5 (Growth Dynamics) and Chapter 6 (Change Dynamics). We discuss the implications arising from our findings in Chapter 7 and conclude in Chapter 8.
Chapter 3

Data Selection Methodology

Empirical research by its very nature relies heavily on quantitative information. In our study, we extract information from a number of different Open Source Software Systems. This chapter provides an overview of the various sources of information that can be used to study evolution, the Open Source Software Systems that we selected, and the criteria used to select the software systems.

3.1 Evolution History

Research into software evolution relies on historical information. When information is extracted from various data sources (for example, source code, project plans, change logs etc.) of a software project over time, we obtain the evolution history of a software system. Broadly classified, there are three types of evolution histories (see Table 3.1): (i) the release history, (ii) the revision history, and the (iii) the project history. The release history contains the software artifacts that are released at regular intervals in the project. The revision history is composed of the version control logs and issue/defect records. The project history is made up of the messages (e.g. email, chat logs), project documentation as well as process information.

The software artifacts from the release history (specifically binaries and source files) offer a direct evolutionary view into the size, structure and
### Chapter 3. Data Selection Methodology

<table>
<thead>
<tr>
<th>History</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release History</td>
<td>Source code, binaries, release notes, and release documentation</td>
</tr>
<tr>
<td>Revision History</td>
<td>Version control logs, issue/defect records, Modification history of documentation, Wiki logs</td>
</tr>
<tr>
<td>Project History</td>
<td>Messages (email, Instant message logs), Project documentation (plans, methodology, process)</td>
</tr>
</tbody>
</table>

**Table 3.1:** The different types of histories that typically provide input data for studies into software evolution.

The composition of the actual software system. The other categories of information tend to provide a supporting, indirect view and can help fill in the gaps in understanding the evolution of a software system. However, the information derived from the *revision history* and *project history* is reliable only if the development team was disciplined enough to record and archive it carefully. For instance, if most of the discussions in the project happen on personal e-mail or verbally, then that category of information is hard to obtain. Similarly, for version control logs and defect repository data to be useful, developers must properly and regularly use the appropriate tools and record information accurately.

Research work that focuses on analysing the *release history* studies the actual outcome of changes, in most cases the source code over time. The Laws of Software Evolution as defined by Lehman [167, 168, 174] were built pre-dominantly from the analysis of *release histories*. Researchers that have focused on this area have been able to identify typical patterns of evolution and change [27], construct statistical models of growth and change [21, 127, 132, 152, 193, 264, 269, 270, 283, 310], develop methods to identify change prone components [109, 149, 253, 281], and have proposed methods to visualise evolution [1, 65, 79, 95, 163, 164, 298]. An interesting contribution from studies of release history is that modular architectures style can allow for rate of growth beyond the sub-linear growth rate expected by the Laws of Software Evolution [100, 200]. This insight provides some level of empirical support for the recommendation from software engineering to build software as loosely coupled modules [218, 302].
Studies that focus on analysing the revision history provide a direct insight into maintenance activities as these studies focus on a log of the changes recorded in the version control system and defect tracking system [10, 17, 28, 68, 86, 109, 113, 140, 146, 149, 231, 232, 312, 316, 319, 321]. The revision history has been the primary choice used by change based studies (as discussed in Section 2.2).

Although the version control logs, change logs, and defect logs are inherently unreliable due to the human dimension, they still offer a valuable source of information when interpreted qualitatively as well as for providing a high-level indication of change patterns. For example, researchers that have analyzed version control logs [231, 232, 312, 316, 319, 321] developed techniques to identify co-evolution, that is, artifacts that tend to change together.

Unlike research studies based on release history and revision history, work that focuses on project history is comparatively minimal [25, 139], potentially because of the time consuming nature of the activity and the difficulty in obtaining necessary data. Despite these challenges, some researchers [133, 206, 246, 249] have studied aspects of this information in open source projects and have provided a valuable insight into the nature and social dynamics of these project. An interesting finding is that the community structure and culture within the project co-evolves with the software, and they influence the growth dynamics of each other [133, 206, 246, 249]. Some studies have also confirmed that developers in open source projects tend to participate in multiple projects, creating networks that influence multiple project evolutions as they tend to share code and solution approaches [134, 187, 248]. Based on an analysis of project histories, Mockus et al. [199], Scacchi [247] and German [90] have argued that there are ideas and practices that can be adopted from successful open source software development projects into traditional and commercial software development. For example, they show that shorter release cycles and use of defect repositories for tracking defects as well as requirements has allowed geographically dispersed team members to collaborate and work effectively on complex software projects (like the Apache Web Server and the Mozilla Browser) [199, 200].
In this thesis, we use the release history as our primary source of data. Our approach involves collecting metric data by processing compiled binaries (Java class files, JAR and WAR archives). We consider every release of the software system in order to build an evolution history. The analysis then uses the extracted metric data as the input. A comprehensive discussion of our technique, and the actual measures are presented in the next chapter (Chapter 4). Though, our focus is on the release history, we also make use of the revision and project history in order to gain a deeper insight and better understanding of any abnormal change events. For instance, if the size of the code base has doubled between two consecutive releases within a short time frame, additional project documentation and messages on the discussion board often provide an insight into the rationale and motivations within the team that cannot be directly ascertained from an analysis of the binaries or the source code alone. This approach of considering multiple sources of information in studies of evolution is also suggested to be effective by Robles et al. [240] as it will provide a more comprehensive picture of the underlying dynamics than can be obtained by purely relying on a single source of information.

### 3.2 Open Source Software (OSS)

In our study, we investigate evolution in Open Source Software (OSS). But, what is OSS? and why do we focus on this type of software?

Open Source Software (OSS) is, in general, software which is free, and distributed along with the source code at no cost with licensing models that conform to the Open Source Definition (OSD), as articulated by the Open Source Initiative ¹ (see Table 3.2). Typically the term “free” carries multiple meanings and in this context, it implies that the software is: (i) free of cost, (ii) free to alter, (iii) free to distribute, and (iv) free to use the software as one wishes. In contrast, commercial software is in most cases sold at a cost, with restrictions on how and where it can be used, and often without access to the source code. Though, some

---

¹Open Source Initiative [http://www.opensource.org](http://www.opensource.org)
Table 3.2: The criteria that defines an Open Source Software System.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free Redistribution</td>
<td>Redistribution of the program, in source code or other form, must be allowed without a fee.</td>
</tr>
<tr>
<td>Source Code</td>
<td>The source code for the program must be available at no charge, or a small fee to cover the cost of distribution and media. Intermediate forms such as the output of a preprocessor or translator are not allowed. Deliberately obfuscated source code is not allowed.</td>
</tr>
<tr>
<td>Distribution of Modifications</td>
<td>Distribution of modified software must be allowed without discrimination, and on the same terms as the original program.</td>
</tr>
<tr>
<td>License</td>
<td>The license must allow modifications, derived works, be technology neutral. It must not restrict other software, and must not depend on the program being part of a particular software distribution.</td>
</tr>
<tr>
<td>Integrity</td>
<td>The license may require derived and modified works to carry a different name or version number from the original software program.</td>
</tr>
<tr>
<td>No Discrimination</td>
<td>The license must not restrict the program to specific field of endeavour, and must not discriminate against any person or group of persons.</td>
</tr>
</tbody>
</table>

commercial systems are distributed free of cost, the licensing models often restrict alteration and how they can be used and distributed.

Projects that develop and distribute Open Source Software have over the past two decades championed a (radical) paradigm shift in legal aspects, social norms, knowledge dissemination and collaborative development [200]. One of the most compelling aspects of Open Source Software projects is that they are predominantly based on voluntary contributions from software developers without organisational support in a traditional sense [202]. The typical open source model pushes for operation and decision making that allows concurrent input of divergent agendas, competing priorities, and differs from the more closed, centralised models of development [83, 215, 234]. These open source projects have over time evolved tools and techniques by experimenting with a range of ideas on how best to organise and motivate software development efforts, even when developers are geographically dispersed and not provided any monetary compensation for their efforts. In these
projects, methods and tools that have not added sufficient value were rejected, while embracing approaches that have consistently provided additional value [215]. In a sense, this model of software development has provided an ongoing validation of collaboration techniques that tend to work, are light-weight and provide the maximum return on invested effort [160, 207, 215, 233].

Open Source Software projects due to their very nature often select licenses that do not place any restriction on the use of the software as well as the information and knowledge that is generated during development [176, 262]. The use of these open licenses has opened up a rich data set of information that can be analyzed to understand how developers tend to build such software, how they collaborate, share information and distribute the outcome of their efforts. Further, the lack of restrictions on analysis and reporting of the findings has motivated an interest in open source software for evolution research, including this work (see No Discrimination in Table 3.2). An advantage of focusing on Open Source Software projects is that the findings from research into these projects provides additional insight into the effectiveness and value of the development methods as well as helping identify typical and unusual evolution patterns. Given their increasing adoption in commercial projects [200, 202, 207, 262], an understanding of how these open source software systems evolve is also of value to stakeholders outside of the Open Source community.

### 3.3 Open Source Project Repositories

Quantitative analysis starts with an identification of the sources that can be used to provide the raw data. We selected projects and collected data from public open source project repositories. The past decade has seen the development, and free availability of repositories like Sourceforge ² that provides a comprehensive set of online tools that allow developers to host and manage Open Source Projects. These repositories typically provide tools for version control, discussion boards, messag-

²Sourceforge is currently the largest Open Source Project repository http://www.sourceforge.com.
ing, a web site to host and distribute various releases, a Wiki to create and share documents, as well as a defect/issue tracking tool.

The increasing use of these centralised repositories by the open source development community has created repositories with a substantial number of projects, with many repositories hosting well over 1000 active and popular projects (for instance, Sourceforge and Google code). Since these repositories act as portals to a number of projects, they maintain statistics on popularity of the various projects and provide a ranking based on the activity on a project. They typically rank projects by measuring the number of files modified/added, messages on the discussion board, and updates on the version control system log. We used this ranking data, specifically by focusing on the top 10 Java projects from each repository shown in the list below as a starting point to help identify candidate systems before applying a more tightly specified selection criteria (described in the next section).

We mined projects hosted on the following repositories:

2. OW2 Consortium - http://www.objectweb.org

### 3.4 Selection Criteria

In order to identify suitable systems for our study, we defined a number of selection criteria. The set of criteria used and the rationale for our selection is presented in this section.
Chapter 3. Data Selection Methodology

The selection criteria that each project must satisfy are as follows:

1. The system must be developed for the Java virtual machine. Source code and compiled binaries are available for each release.

2. The software is a single coherent system, that is, it is a distribution of a collection of related *components* packaged together.

3. At least 15 releases of the system are available. Only complete releases with a version identifier are considered. Branches and releases not derived from the main system tree are ignored. Minor and major versions are both considered (for instance, Version 2.0, 2.1 and 3.0 are all considered. In this case, the version with identifier 2.1 is often a release that provides minor enhancements and/or defect corrections).

4. The system has been in actively development and use for at least 36 months.

5. The system comprises of at least 100 types (i.e., classes and interfaces) in all releases under study.

6. Change logs do exist. This data provides the additional information to understand the rationale behind the changes.

Further to the criteria for individual projects, we set a target of collecting a minimum of 30 different software projects in order to ensure we have sufficient diversity in our data set allowing some flexibility in generalising our conclusions.

**Rationale of Selection Criteria**

Java is currently a popular language with wide spread use in both open source and commercial projects. This popularity and usage has resulted in a large amount of software developed using the Java programming language. Despite its popularity and use in a variety of domains, there are only a few studies that exclusively study release histories of Java software systems [21, 193, 208, 270, 319]. Further, these studies
restricted their scope to a few systems (typically less than 10 projects) and hence do not have sufficient statistical strength in order to generalise their findings. Although, studies exist into growth and change of both commercial and open source software systems [188], we do not have sufficient evidence to know if these findings would partially or wholly apply to Java software systems. Specifically, since most of the earlier studies have investigated software developed in C and C++.

Systems were required to have at least 36 months of development history and 15 releases to increase the likelihood of the existence of a significant development history. Further, as noted in recent work by Capiluppi [39], only 15% of open source projects have a development history greater than 2 years, with only 2% of the projects surviving more than 3 years. Our selection criteria in effect focuses on a small subset of the total available projects and we are studying only systems that can be considered successful projects. The bias was intentional as we wanted to learn from systems that have successfully evolved, rather than from software that failed to go beyond a few months. Systems that fail can do so for a number of possible reasons and in open-source projects the exact reason will be hard to identify precisely, predominantly because much of the work is voluntary and the typical business pressures such as a release schedule are not in operation. Furthermore, there is no widely accepted definition of failure [98,99].

The restriction of at least 100 types was motivated by the desire to avoid trivial software systems. Further, small software systems will not have sufficient variability limiting the extent to which we can generalise the conclusions. We restricted our input data to systems that provided a change log outlining the modifications made to the software system between releases. These logs were typically made available as part of the release notes or, captured by the defect tracking software used by the project. The change logs provide indicators that helped us explain anomalies and abrupt changes within the evolution history, for instance, these logs tend to mention when significant restructuring, or architectural changes took place. When this information was used in conjunction with the source code, we were able to understand better the nature and type of changes.
Chapter 3. Data Selection Methodology

The size and skill of development teams, though helpful, was a criteria that was removed after an initial pass at selecting systems mainly because it was not possible to obtain this information accurately. In some of our projects, the software used to host the source control repositories changed during the evolutionary history of a project and many projects choose to archive older contribution logs at regular intervals removing access to this data. These aspects limited our ability to determine the number of active and contributing developers to the project, specifically during the early part of the evolution. Another facet that could not be accurately determined was that the level of contribution from different developers. That is, we were not able to identify reliably if some developers contribute more code than others. Further, some project members contributed artwork, documentation, organised and conducted meetings while some focused on testing. These non-code contributions were often not visible as active contributors on the source code repository. Another interesting finding during our investigation was that developers that have not contributed any material code for many years are still shown as members in the project. These limitations, including an observation that suggests that a small sub-set of developers are responsible for a large amount of the changes and additions to the source code in open source software, has been noted by Capiluppi et al. [39].

The observation that few developers contribute most of the code by Capiluppi et al. [39] and the variance in the contribution levels over time indicates that we require a measure that can meaningfully identify the number of normalised developers working on a project at any given point in time. However, such a metric has not yet been developed and widely accepted as effective and hence we did not rely on the development team size as a variable for use in our study.

3.5 Selected Systems - An Overview

Using the selection criteria, we initially identified 100s of software systems that satisfy the criteria. However, we focused on a representative smaller subset in order to allow us to study each of the selected systems at a greater depth. Our final data set comprises of forty software sys-
tems, 1057 unique versions and approximately 55000 classes (in total over all the various systems and releases). Our data comprises three broad types of software systems: (a) Applications, (b) Frameworks, and (c) Libraries. In our selection, we aimed to select a similar number of systems for each of the types.

Applications are software systems that can be considered stand-alone and tend to perform a specific set of tasks. Examples from our data set include a Bit-torrent client, a role playing game, an image editor, and a text editor. These systems often tend to have a graphical user interface component. Frameworks and Libraries, on the other hand are systems that provide generic/reusable abstractions with a well defined API. There is one key distinction between between a framework and library. Frameworks tend to impose a much stricter flow of control in the program. However, when we classified the systems, we used the terminology that has been used by the development team. So, if a development team classifies their system as a framework, we use the same term. Examples of frameworks in our data set are the Spring framework (an inversion of control framework) and Hibernate Object Relational mapping framework. Some of the libraries that we investigated are popular XML processors like Apache Xerces and Saxon. A full list of all of the systems is provided in Table 3.3, and the meta data that we capture for each system is presented in Appendix A. Our data set contains 14 Applications, 12 Frameworks and 14 Libraries.

A set of projects in our data (for instance Hibernate, Spring Framework, Acegi and Saxon) though open source, are developed by engineers that get paid for their effort as these projects are managed by commercial (for-profit) enterprises. All of these systems originally started as traditional open-source projects and over time adopted business models that generate revenue while the source code is still made available under a range of open source licenses. We tagged a system as commercially backed based on information provided on the project web site which indicated the name of the sponsoring company and the developers that work for that entity.
### Table 3.3: Systems investigated - Rel. shows the total number of distinct releases analyzed. Age is shown in Weeks since the first release. Size is a measure of the number of classes in the last version under analysis.
Chapter 3. Data Selection Methodology

The type of software and its commercial sponsorship information are not properties that we directly use in the models constructed as part of the research described in this thesis. This additional meta-data was collected since one of the contributions of this thesis is the archive of releases which is useful for our own future work as well as for other researchers in this field.

3.6 Focus of Study

A typical software system contains a number of different items that can be used as input into a study of software evolution. For instance, we can study the evolution of binaries, the source code or documentation. In this section, we describe the focus of this study. Specifically, we explain the data set that is used as the primary input for the qualitative analysis.

3.6.1 Categories of data sources

Software projects offer a range of different kinds of information that can be analysed in order to understand how they have evolved. At a high-level, we can classify these sources of data into the following categories:

1. *Software artifacts* produced and distributed as a version, including binaries, source files, end-user and developer documentation (including release notes).

2. *Logs* generated by the tools used for version control.

3. *Messages* on mailing lists, discussion boards, instant message logs and e-mail that are generated as developers communicate with each other.

4. *Project documentation* that is generated during the development of a version and typically made available via a Wiki or a Content Management System (CMS). Examples of artifacts in this category are:
Process models, development methodology, management reports, resource allocation charts, project plans and coding standards.

5. *Records* added and updated on the Defect/Issue tracking system

In our study, we analyse the *Software artifacts* by building a release history and use data from other sources to understand and interpret the evolution of the software system.

### 3.6.2 Java Software Systems

The common practice in Java open source projects is to package the *Software artifacts* as a release consisting of a compiled binary bundle and a source bundle, both of which are distributed as compressed archives (typically zip archive files) with a number of different files within it.

A compiled Java software system comprises of a set of *class files*, which are generated by the compiler from the source code. Both Java classes as well as Java interfaces are represented as *class files* within the Java environment. In order to help with distribution and deployment of this set of class files, the Java development kit provides a utility to create a Java archive (JAR file) that holds all the class files as well as all related configuration and supporting data files (including images) in a single bundle. The JAR file format is used by most contemporary open source Java projects as the preferred method for distributing the Java software systems. In our data set all projects used JAR files as their preferred distribution method. The Java archive files also have the advantage of being able to integrate into the Java environment with minimal configuration.

We analyse the binary bundle in our study and it typically contains the following items:

- A set of Java archives (JAR or WAR files) that form the *core* software system
Chapter 3. Data Selection Methodology

Figure 3.1: Component diagram of a typical software system in our study. Only the Core System JAR components (highlighted in the image) are investigated and used for the metric extraction process.

- Third party Java libraries, often distributed as JAR files
- Third party Operating System specific libraries. Typically dynamically linked libraries or shared libraries.
- Configuration files
- Release documentation
- Data files (including database, media and other binary/text files)

In our study, we collect metrics from the core software system and ignore third-party libraries (see Figure 3.1).

Using Binaries

We extract the measures for each class by processing the compiled Java bytecode instructions generated by the compiler (details are explained in Chapter 4). This method allows us to avoid running a (sometimes
Chapter 3. Data Selection Methodology

quite complex) build process for each release under investigation since we only analyze code that has actually been compiled.

Our approach of using compiled binaries to extract metric data is more precise when compared to the methods used by other researchers that studied evolution in open-source software systems since the earlier work used source code directories as input for their data analysis [41, 100, 105, 120, 153, 217, 239, 256]. In order to process the large amount of raw data, many of the previous open source software evolution studies used data gathered from size measures, such as, raw file count, raw folder count and raw line count. These measures were computed with some minimal filtering using Unix text utilities that work with files based on their extension, for example, *.c and *.cpp to capture C and C++ source files respectively. These approaches have the advantage of providing a general trend quickly and are practical when attempting to process many thousands of projects. The file based processing method, however does not directly mine any structural dependency information. It also includes source code files that may no longer be part of the code base – essentially unused and unreachable code that has not been removed from the repositories.

This practice of leaving old code has been noted by researchers in the field of code clone detection who observed the tendency of developers to copy a block of code, modify it, and leave the old code still in the repository [5, 135, 155, 157]. Godfrey et al. [100] in their study of Linux kernel evolution noted that depending on the configuration setting in the build script (Makefile), it is possible that only 15% of the Linux source files are part of the final build. The use of only a small set of source for a release is common in software that can be built for multiple environments. For instance, Linux is an operating system designed to run on a large range of hardware platforms. When building the operating system for a specific hardware configuration, many modules are not needed and hence not included during the build process using settings provided in the Makefile. Hence, when using source code files as input into a study of evolution, ideally, the build scripts have to be parsed to determine a the set of files for a specific configuration and then the evolution of the system for this specific configuration has to be
analysed. Many previous studies [41, 100, 105, 120, 153, 217, 239, 256] that use release histories do not explicitly indicate if the build scripts have been pre-processed adequately to ensure that the correct set of source files is used as the input.

In our study, we use compiled releases (Java classes package inside JAR files) to construct our release history and so our input data has already gone through the build process, reducing the chance of encountering code that is no longer in active use. This approach allows us to focus on the set of classes that have been deemed fit for release by the development team.

**Third Party Libraries**

Development teams in general, use a number of third party Java libraries as well as the standard Java libraries (which are part of the Java runtime environment) in order to improve their productivity. In our study, we classify the set of classes created by the development team as the core system and focus explicitly on how this core system evolves (see Figure 3.1). This scope allows us to gain a direct perspective into the efforts of the development team. Our tighter focus has the advantage of ensuring that the size and complexity measures that we collect are not influenced by changes in the external libraries. Although, the developers of the core system potentially exert some evolutionary pressure on external libraries as consumers, they do not directly influence the internal structure, organisation and size of these external libraries.

Our intentional focus of ignoring third party libraries distinguishes our study from other previous large scale studies into open source software evolution where this choice was not explicitly made or stated in their work [39, 120, 129, 188, 217, 239, 306]. These third party libraries add volume to the the size measure and have the potential to distort the evolutionary patterns and may indicate a faster rate of growth than would be possible if only the contributions of the core team are considered.
Including the external libraries also has the potential to distort measures of complexity and may indicate that a project is far more complex than it really is. For example, if a project makes use of two complex libraries for visualization and signal processing the structural and algorithm complexity of these libraries will be considered to be part of the actual project under investigation and the core project will show far more complexity than what needs to be considered by the developers.

Although, including third party libraries provide another dimension into evolution, from the developers perspective the effort is expended on selection of the library and learning it rather than in construction of the library. Furthermore, it is possible that even though a large library is included, only a small fraction of the features are directly used and as a consequence reduce the strength of any inferences derived from the observed evolution. We therefore focus on the set of code that can be considered to be directly contributed by the developers and hence potentially maintained by the development team as it evolves.

All systems that we have analyzed made extensive use of additional Java-based third party libraries with a few systems making use of libraries written in C/C++. In our study, these third party libraries as well as all Java standard libraries are treated as external to the software system under investigation and for this reason we do not collect metric data for classes in these libraries (See Figure 3.1). For instance, if a software system makes extensive use of the Java Encryption API (provided as part of the standard Java framework), we do not extract metrics for classes in this external encryption library as the effort that has gone into developing these libraries does not directly impact on the software system under investigation.

We also noticed that many projects rely on the same set of libraries and frameworks. For example, the Apache Java libraries are extensively used for String, Math, Image, and XML processing. Though, there are a large number of options available, the repeated selection of the same set of libraries indicates that there is a strong preferential attachment model [16] at play in open source projects, where a few rich and popular projects tend to dominate the landscape.
Chapter 3. Data Selection Methodology

The approach we take for detecting third party libraries in order to remove them from our measures is explained in the next chapter (Chapter 4).

3.7 Summary

Research into software evolution relies on historical information. There are three types of histories that can be used to understand the evolution of a system: (a) Release history, (b) Revision history or, (c) Project history. Our research effort studies release histories of forty Java software systems. We investigate Open Source Software Systems due to their non-restrictive licensing. Further, unlike previous studies that worked with source code files we use compiled binaries and also actively ignore contributions from third-party libraries.

In the next chapter, we present our approach for collecting metrics from Java software, the description of the metrics collected, and how we model the information to enable our analysis.
Chapter 4

Measuring Evolving Software

The value of measurement is summarised fittingly by the often quoted statement — “You can’t control what you can’t measure” [62]. In the discipline of software engineering there has been wide agreement on the need to measure software processes and products in order to gain a deeper and more objective understanding of the current state of a software system [77]. This understanding is a pre-condition for establishing proper control over development, with software metrics providing the feedback required to undertake corrective actions and track the outcome of the actions [165]. By software metric, we mean a quantitative measure of the degree to which a software abstraction possesses a given attribute [124].

Previous research work in the field of software measurement has focused on defining a range of software metrics [46, 117, 165, 182] to measure different attributes within a software system. These studies are complemented by work undertaken to ensure that the metrics defined are mathematically valid and useful [18, 122, 151]. There have also been studies [31, 42, 52, 78, 205, 265] that have shown the applicability as well as the limitations of software metrics for measuring both size and complexity of a software system. Our research effort is based on this foundation and makes use of software metrics in order to understand the evolution of software systems.
In order to better understand the evolution of a software system, we extract a set of metrics from each release of a software system and observe how these metrics change over time. The core abstraction that we collect metrics from is a compiled Java class. This chapter describes the type of measures that we collect and provides a definition of all the software metrics that we extract. We also outline the approach used to extract the metrics from compiled Java classes and the model that is used to capture the evolution history in order to facilitate our analysis. The specific metrics that we use to address our research questions and the motivation for selecting the metrics is presented in Chapter 5 and Chapter 6 within the context of the analysis approach.

## 4.1 Measuring Software

Measurement is “the empirical, objective assignment of numbers, according to a rule derived from a model or theory, to attributes of objects or events with the intent of describing them” [142], and measurement theory classifies a measure into two categories: direct and indirect measures. Fenton [77] provides a general definition of these two types of measures – “Direct measurement of an attribute is a measure which does not depend on the measurement of any other attribute. Indirect measurement of an attribute is measurement which involves the measurement of one or more other attributes” [77]. A more precise distinction between direct measurement and indirect measurement is provided by Kaner et al. [142], and they describe that a direct metric is computed by a function that has a domain of only one variable while the indirect metric is computed by a function that has a domain with an n-tuple”. For example, Lines of Code (LOC) and the Number of Defects are direct metrics, while Defect Density (defects per line of code) is considered an indirect measure since it is derived by combining two measures – defects, and LOC [77].

In our study, we compute a set of metrics by counting various attributes of an entity under observation. We focus on two distinct entities for measurement: (i) an individual class, and (ii) a class dependency graph.
The classes that we measure are compiled Java classes and our metric extraction approach is discussed in Section 4.5.2. The class dependency graph captures the dependencies between the various classes in the software system. The graph is constructed by analysing all classes in the system and our method is discussed in further detail in Section 4.5.4 and Section 4.5.5. We consider the metrics that are described in this chapter as direct metrics since we compute the value by a direct count of either a class or a graph, rather than by combining different types of measures. That is, the domain used by the metric function contains only one variable.

4.2 Types of Metrics

Software systems exhibit two broad quantitative aspects that are captured by a range of software metrics: size and structural complexity [77]. These metrics can be used to provide a quantitative view of the software systems size and internal structure as well as infer the process being used to create the software system [117]. Over the past few decades, a number of different software metrics have been proposed (e.g., purely size-oriented measures like the number of lines of code (LOC) or function-oriented measures to analyze process aspects like costs and productivity) to aid in the comprehension of the size as well as the complexity of a software system [77, 165]. When these measures are collected and analyzed over time, we can distil a temporal dimension which is capable of revealing valuable information such as the rate of change [174, 175] and evolutionary jumps in the architecture and complexity of the software system under observation [101]. In this thesis, we collect both size and complexity metrics at the class level as this is the primary abstraction under study in our work.

4.2.1 Size Metrics

Size measures provide an indication of the volume of functionality provided by a software system. Size metrics are considered to be a broad indicator of effort required to build and maintain a software system,
since it takes usually more effort to create a larger-size system than a smaller one [78]. Examples of size metrics within the context of object-oriented systems are the *Number of Classes* and the *Number of Public Methods* within a class.

### 4.2.2 Complexity Metrics

Unlike size, complexity in software is an aspect that is harder to rigidly define and is an aspect that is often perceived subjectively making it harder to measure [116]. However, a number of researchers have put forward metrics that capture complexity in software. Before, we outline our approach to measuring complexity, we briefly explain why complexity is hard to measure and the various attributes that need to be considered when interpreting any measure of complexity.

**What is complexity?**

The Oxford dictionary defines complexity as "the state or quality of being intricate or complicated" [69]. From a general perspective, a system that is composed of many interacting parts whose behaviour or structure is difficult to understand is frequently described to be complex. Modern software systems are complex as they tend to have a large number of interacting parts, making it difficult to properly understand the overall behaviour, even when complete information about its components and their inter-relations is available.

Some of the key contributors to complexity are [88, 222]:

1. The *size* of the system: more parts require a need to organise them in order to properly comprehend,

2. The amount and depth of *knowledge* available (and used) to digest and understand the system,

3. The level of abstraction that is possible, without loosing too much information
4. The number of different \textit{abstractions} that one has to understand, essentially the variety of information. The size and variety add different aspects, but belong to the same dimension, and

5. The level of \textit{design and order}, where a better designed system lends itself to be understood easily. Specifically, a system that has detectable and well known patterns will tend to improve maintainability.

When complexity is considered from a cognitive perspective, developers perceive it due to the \textit{overload} caused by the number of abstractions they have to deal with as well as the interconnections between them, the \textit{inconsistency} in how the solution is organised, and the \textit{effort expended} on understanding the structure and organisation of a complex system [222]. All of these aspects are inherently subjective and depend on the education, experience and ability of the developers. The effort that developers put into developing a system increases their familiarity with the various abstractions within a software system. Hence, developers new to a system are likely to perceive a greater level of complexity than the developers that worked on a software system since inception. Similarly, depending on the education, capability, and experience of a developer, their perception of inconsistency and ability to deal with the range of abstractions is likely to be different. In sum, creating, understanding and modifying complex structures requires concerted effort. As a consequence, software systems with high complexity imply a great investment in resources in order to understand, and sustainably maintain and grow the software without errors [15, 20, 110, 213, 257, 313, 314, 317, 318].

\textbf{Measuring Complexity}

There are two broad types of complexity that can be measured: Volumetric and Structural [77]. Volumetric complexity is measured by counting the number and variety of abstractions, whereas the interconnections between these abstractions is used to derive structural complexity measures [147] which provide an insight into the structure and organisation of a software system [45, 46, 122, 250, 265, 277].
Chapter 4. Measuring Evolving Software

In the context of object-oriented software a range of measures of class complexity have been defined [117]. In general, there are two perspectives used when measuring the complexity of a class. In the first perspective, the complexity of a class can be computed by measuring the internal structure. A widely used metric to capture this internal structure is the Weighted Method Count (WMC) metric [46] where cyclomatic complexity [190] is used as the weight [177]. The WMC metric reflects the degree of control flow present within a class and has been shown to be an indicator of fault-proneness [19].

The second perspective computes complexity of a class by measuring its coupling with other classes in the software. Two commonly used metrics to capture the degree of coupling are the In-Degree Count and Out-Degree Count [21, 54, 55, 77, 117, 165, 209, 223, 287, 294, 299]. The In-Degree Count metric measures the number of classes a particular class provides services to (that is, a measure of its popularity), while the Out-Degree Count metric measures how many classes a particular class depends upon, respectively. These measures capture the level of coupling within a software system which serves as an indicator of the difficulty developers potentially face during maintenance and evolution [31, 52, 205]. For example, a class \( X \) with a high In-Degree Count (relative to other classes in the system) is considered complex, since any changes made to \( X \) have the potential to significantly impact other classes that depend on \( X \). Similarly, a class \( Y \) that has a very high Out-Degree Count is also considered complex, since \( Y \) makes use of a large number of different functional aspects of the system in order to satisfy its responsibilities. As a consequence, developers cannot alter \( Y \) in a meaningful way before they understand classes that \( Y \) uses.

In our study, we collect complexity metrics for each class from both perspectives. Specifically, we measure the internal structural complexity of a class as well as the coupling for a class (the specific metrics and their definitions are described in the sections that follow). Furthermore, we use the term “complexity” to imply structural complexity rather than volumetric complexity.
4.3 Software Evolution History Model

Any non-trivial object-oriented software system will generally contain many hundreds of classes, and in order to understand the evolution of these systems, there is a need to manage and process this potentially large pool of data. In our study, we model the entire evolution history of an individual project using three key elements: Release History, Version and Class Metric. Figure 4.1 shows the relationship between these three entities and the core data that they capture in our model.

Release History captures the entire evolution history of a single software system, and it holds the set of versions ordered by Release Sequence Number (RSN). The RSN is assigned incrementally for each version based on the release date and serves as a pseudo-time measure [58, 174]. Every Version consists of a set of Class Metric entities which directly map to an individual compiled Java class file and store the metric information extracted for each class. In Java, both interfaces and classes compile into class files and hence we model it directly as such, with a flag within the class metric entity that determines its actual type.
Our approach can be contrasted with the Hismo meta-model proposed by Girba et al. [93] which also models history as an ordered set of versions. However, in contrast to Hismo, we do not explicitly create abstractions for various type of histories, for example, the inheritance history or the package history. In our method, we achieve a similar outcome by constructing a set of reports (for example, size evolution report or an inheritance history report) by processing the information captured in our three main entities. Our technique allows us to construct dynamic reports as needed to answer various research questions, rather than building a larger static model.

4.4 Measuring Time

Studies into software evolution typically use two different measures of time: Release Sequence Number (RSN) and Calendar time. In this section, we present a discussion of these two types of measures and motivate our method for measuring time. In particular, we argue that calendar time is a more appropriate measure of time.

4.4.1 Release Sequence Number (RSN)

The measure of RSN is considered to be a pseudo-time measure [58] since it treats the time interval between two releases to be constant and it is independent of elapsed time. The RSN measure has the advantage of being able to directly reflect a specific version and hence corresponds to a well defined unit in the release history of a software system [283].

The key limitation to the use of RSN arises when attempting to compare aspects like growth rates in different software systems [17, 217, 277] since the time interval between releases in different software systems cannot be assumed to be the same constant value. Furthermore, since the time interval between releases does not correspond to a more intuitive measure of real elapsed time, models that use RSN have to be carefully interpreted.
Interestingly, Lehman et al. in his seminal studies of evolution used RSN as a measure of time [168, 175]. However, the limitations due to RSN being a pseudo-time measure have not been explicitly considered to be an issue, possibly because Lehman’s laws suggest that effort is on average constant, and that releases are made at regular intervals justifying the use of RSN’s as a proxy for time as well as effort (that is, RSN is considered to be an interval scale measurement [77]).

The shortcoming of RSN as a measure of time in evolution models is strongly highlighted by recent work of Thomas et al. [277] (published in 2009) who repeated an experiment conducted by Schach et al. [250] in 2002. Schach et al., used RSN as their measure of time and observed the Linux kernel size exhibited super-linear growth, and that common-coupling increased exponentially based on an analysis that used linear regression. Based on this super-linear growth observation, Schach et al. expected Linux kernel to experience serious maintenance problems and recommended restructuring of the kernel. Interestingly, in spite of the alarming conclusions of Schach et al., the Linux kernel continued to attract new developers and managed to grow in subsequent years. Thomas et al. were motivated to explain the contradictions between the expectations of Schach et al. and the actual observations and hence repeated the experiment using both RSN as well as calendar time in their regression models. Thomas et al. [277] observed that when calendar time was used as the primary measure of time, the size growth in the Linux kernel was linear and the growth in common coupling follows the same pattern. These observations provide additional support to highlight the limitation of RSN and the potential for improper conclusions to be derived if the assumptions about the time variable are not fully considered when interpreting the models.

Within the context of Open Source Software, we consider RSN as a measure of time that satisfies the *ordinal scale*, but not the *interval scale*. The Release Sequence Number is a valid ordering construct, but developers in Open Source projects do not always release software at near constant intervals and hence it cannot be on an interval scale, limiting the use of RSN in statistical models (e.g. linear regression). We illustrate the erratic time interval between releases in Figure 4.2 using
four software systems from our data set. If developers release software at regular intervals the scatter plots (cf. Figure 4.2) would show substantially less variability. Further, given the variability in the data, we are also unable to derive a generalizable, and sufficiently strong linear relationship between RSN and “Days between Consecutive Releases” which is necessary for RSN measure to be considered an interval scale measure [260]. Though, the intervals are erratic, interestingly in approximately 70% of the releases (across our entire data set) we noticed that the gap between consecutive releases is less than 90 days (see Figure 4.3). This observation indicates that there exists some pressure on the development team that compels them to release software at reasonable intervals, potentially to ensure ongoing community support.

Since we treat RSN as an ordinal scale measure, we apply only the set of mathematical operations that are valid for the ordinal scale. This restriction implies that we do not use RSN in any parametric regression.
Chapter 4. Measuring Evolving Software

![Cumulative distribution showing the number of releases over the time interval between releases.](image)

**Figure 4.3:** Cumulative distribution showing the number of releases over the time interval between releases.

equations, nor do we compute the mean and standard deviation on RSN since operations like addition and subtraction have no defined meaning. Though, RSN has been treated as an interval scale measure within some models of evolution in previous studies [173, 192, 284], we regard these models to be harder to use since any interpretation of these models needs to take into consideration the potentially unequal time interval between releases.

### 4.4.2 Calendar Time

The measure of calendar time is a more flexible measure than RSN because it directly maps to the more intuitive “elapsed time” with constant interval between units. Additionally, this measure of time is also recommended as a more appropriate and effective in studies of evolution by many researchers [17, 127, 188, 217, 277]. Although calendar
time is the preferred measure, it has a key limitation, in that it is not able to reflect the development effort. That is, we have to make the assumption that more days between releases implies more development effort. However, beyond the specific implication of the Fourth Law of Software Evolution (Conservation of Organisational Stability) which suggests that effort is on average invariant across the operational lifetime of a system [175] there has been no widely accepted relationship between calendar time and effort, specifically within the context of Open Source Software Systems.

In this thesis, we acknowledge this limitation (elapsed calendar time does not necessarily reflect development effort), and ensure that this constraint is appropriately considered when interpreting our observations. Furthermore, we use the days elapsed since first release (referred to as “Days”) as our measure of calendar time and use this measure as an indicator of the Age of a software system. In our study, the first release is always considered to be released on Day 1, with Day 0 corresponding to no release. The age of subsequent releases is computed by adding one to the days elapsed since the first release as illustrated in Figure 4.4. This adjustment is needed since we consider the initial version to be released on Day 1. We use the age measured in Days to represent the time parameter in the mathematical models that we constructed to address our research questions (discussed in greater detail in Chapter 5 and Chapter 6).
Our definition of Days places it on the ratio scale of measurement since we clearly define the zero value [77], permitting the use of Days in common mathematical operations and statistical techniques. Although, we avoid using the Release Sequence Number as a measure of time in the models that we construct, we use RSN as a measure of time when visually illustrating patterns of evolution in a single system, specifically to highlight key versions where changes take place. However, we do not use RSN when comparing different software systems, since the intervals between releases across systems need not be the same.

### 4.5 Metric Extraction

We extract the measures for each class in the Java program by processing the compiled class files. As discussed in the previous chapter (cf. Section 3.6.2), this approach allows us to avoid running a potentially complex build process for each release. The steps in the metric extraction process is presented visually in Figure 4.5 and elaborated in greater detail in the rest of this section.

**Figure 4.5:** The metric extraction process for each release of a software system
4.5.1 Jar Extraction

We begin our metric extraction process by first extracting the compiled class files from inside the set of Java Archives (JAR files) associated with an individual version. JAR files are a standard packaging and distribution method used by all Java projects under analysis. The set of JAR files provided as input into this extraction step was manually constructed and all known external libraries (also packaged as JAR files) were tagged manually for removal (cf. Section 3.6.2 for a discussion of the rationale for removing external libraries).

JAR files were tagged as potential external libraries based on the package names of classes inside the JAR file. We found that using the package names was an effective method to detect potential external libraries because Java developers tend to follow the recommended standard package naming convention and embed the name of the project, organisation or team in the package name [235]. For example, all classes developed by the Hibernate project have a package name that starts with org.hibernate. We used this common naming convention that is applied by developers to cluster package names manually (after a simple sort) and then identify potential third-party JAR files. Once potentially distinct set of packages was identified, we used a Google search to check if a distinct project with its own source code repository was available on the web that matched the package signature identified.

Using this external library identification technique on our data set, we were able to identify separate project web sites as well as source code repositories for many third party libraries within the software systems. Once a distinct open-source project was identified as the primary contributor of the external library, we created a regular expression to match the package names of known third party libraries and used this regular expression to identify and remove external library JAR files from all versions of the software system. An example of such a pattern was the one created to identify the use of Apache Commons library where the package names had a format org.apache.commons.*.
Chapter 4. Measuring Evolving Software

Once a pattern was established to identify specific libraries, we used the same pattern across all projects in our data set. The regular expression based external library identification lists created for each software system in our data set was also manually checked to ensure that it was not selecting classes that can be considered to be part of the core software system (cf. Section 3.6 for a description of core software system).

In the final stage of this step, we determined the release date for the version from the JAR files that have been determined to be part of the core software system. All Java archives contain a Manifest file that is created as part of the JAR construction process. We use the creation date timestamp of this Manifest file to determine the release date for the Version. Where a version contains multiple JAR files, we apply the maximum function and take the latest date to represent the release date for the entire version. This was needed since certain projects tend to constructed JAR files for their distribution over multiple days rather than build it all on a single date.

Once the release date for a version was established, we ordered all versions by release date and compute the Release Sequence Number (RSN) for each version. We started the RSN numbers at 1 for the oldest version and incremented it by 1 for each subsequent version.

4.5.2 Class Metric Extraction

After the classes files are extracted from the JAR file, we process each class file using ASM, a Java Bytecode manipulation framework [9], in order to extract information from the compiled Java class (Table 4.1 highlights the information that is available to be extracted from a compiled Java class). In this step we compute direct measures such as the Number of Fields for a class as well as extracting its additional information such as the fully qualified class name (i.e. class name includes the package name; an example of a fully qualified class name is java.lang.String, where java.lang is the package name).
Chapter 4. Measuring Evolving Software

| General          | Fully Qualified Class Name  
|                 | Super Class Name, Interfaces Implemented  
|                 | Modifiers  
|                 | Constant Pool: Numeric, String and Type Constants  
|                 | Source File Name (optional)  
|                 | Enclosing Class References  
|                 | Annotation*  
|                 | Attribute*  
| Inner Class*    | Name*  
| Field*          | Modifiers, Name, Type  
|                 | Annotation*  
|                 | Attribute*  
| Method*         | Modifiers, Name, Return Type, Parameter Types  
|                 | Annotation*  
|                 | Attribute*  
|                 | Compiled Code (Java bytecode instructions)  

Table 4.1: Structure of a compiled Java Class. Items that end with an * indicate a cardinality of zero or more [180].

Java compiler and class file structure

The Java compiler reads class and interface definitions, written in the Java programming language [108], and compiles them into class files [131] that can be executed by the Java Virtual Machine (JVM) [180]. A compiled Java class, in contrast to natively compiled programs (for example, a C/C++ application), retains all of the structural information from the program code and almost all of the symbols from the source code [184, 268]. More specifically, the compiled class contains all information about the fields (including their types), method bodies represented as a sequence of Java bytecode instructions, and general information about the class (see Table 4.1 for an overview). Although, a range of different Java compilers are available, the class files that are generated by these compilers must adhere to the JVM specification [180] and hence all of the data that we process to extract the metrics matches a common specification.

The Java bytecode instructions that are generated by the compiler consists of an opcode specifying the operation to be performed, followed
**A simple method to print “Hello World” to console**

```java
public void printHelloWorld()
{
    System.out.println("Hello World");
}
```

**Listing 4.1:** Same of bytecode generated for a simple Hello World method

by zero or more operands which contain the values to be operated upon [180]. There are a total of 256 possible opcodes instructions that can be generated by the compiler and all of these instructions are embedded in the method body (see Table 4.1). We process the bytecode instructions and determine the nature of the operation in order to compute appropriate metrics. The sample code in Listing 4.1 for a “Hello World” method shows the set of bytecode instructions that are generated as comments within the method body. In this sample listing, there are 4 bytecode instructions generated from a single line of source code (three bytecode instructions take a single operand, while one of the operations has zero operands). In our study, we process these bytecode instructions, as well as all of the other information embedded in the compiled class (as indicated in Table 4.1) to compute the various metrics. A fully annotated Java program is presented in Appendix D to further illustrate how our metric extraction approach counts the various metrics from the compiled class file.

**Differences between source code and compiled code**

Though, the compiled Java class is close to the source code, there are some differences:
A compiled class describes only one Java class, or one Java interface. This constraint extends to inner classes as well, and each inner class is compiled into a separate class file. When a source code file contains a main class with multiple inner classes, the Java specification requires the compiler to generate distinct compiled class files for each of the inner classes as well as one for the parent class. Furthermore, the compiled parent class will contain references to the inner classes. Similarly, the compiled inner classes also have a reference to either the enclosing parent class, or the enclosing parent method (if an inner class is declared within scope of the a method).

The type name of the compiled class is fully qualified. That is, the package name is included. However, in the source code the package name is stated as a separate statement.

All compiled Java classes are required to provide a constructor, and must inherit either directly or indirectly from the Java specification defined class java.lang.Object. However, within the source code it is valid for developers to write a class within a constructor, and they can choose not to explicitly inherit from another class. If the developers make these choices, the compiler will generate a default constructor, and will ensure that the class is a sub-type of java.lang.Object.

The names of all classes that a compiled class statically depends upon are resolved by the Java compiler, and the fully qualified type names are provided in the compiled class. This feature is enforced by the Java language specification and reduces some of the complexity of the metric extractor since no further processing is needed in order to extract the fully qualified type names. The need for further processing arises if we were to rely on the information provided in the source code for metric extraction since the developers do not generally use the fully qualified type name, nor do they typically import the exact set of classes that they depend upon in the source code. For example, developers may choose to import a set of classes within a package using a statement like “import java.util.*” in the source code, rather than stating...
the exact sub-set of classes that they use from this package. Furthermore, the type names within the source code typically contain just the class name, not the fully qualified name (for example, it is more common to use Math rather than java.lang.Math when the developers rely on mathematical library functions).

- All comments are removed from compiled class files.
- The compilation process typically erases local variable names and hence we lose these symbol names in the compiled class.
- A compiled class contains a constant pool which is an array containing all the numeric, string and type constants that are used in the class. These constants are defined only once in the constant pool and referenced (via an index) in all other sections of the class.

**Metrics Extraction and Post Processing**

We process each compiled Java class file and extract two types of metrics: direct count metrics and modifier flags. Table 4.2 shows the list of count metrics that we extract by processing field and method interface information for each class, Table 4.3 shows the list of count metrics that are computed by processing the bytecode instructions present in method bodies, and Table 4.4 shows the flags that we extract for each class. In this thesis, we treat all the count metrics as measures of size as they reflect size of a class from different perspectives. However, we consider the Number of Branches (NOB) measure as a complexity metric that captures the internal structure of a class. The NOB measure is equivalent to the widely used Weighted Method Count (WMC) metric [46] with Cyclomatic Complexity [190] used as the weight [46]. The WMC and (hence our formulation of the NOB) is accepted within the literature as a measure of structural complexity [116, 165].

Along with the metrics shown in Tables 4.2, 4.3, and 4.4 we also capture the fully qualified name of each class, its fully-qualified super class name as well as all method names (including full signature capturing the return type), field names (including the type) and the fully-qualified name of all other classes that a class depends upon.
## Chapter 4. Measuring Evolving Software

### Table 4.2: Direct count metrics computed for both classes and interfaces.

<table>
<thead>
<tr>
<th>Abbv.</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>Method Count</td>
<td>Number of methods</td>
</tr>
<tr>
<td>AMC</td>
<td>Abstract Method Count</td>
<td>Number of abstract methods</td>
</tr>
<tr>
<td>RMC</td>
<td>Protected Method Count</td>
<td>Number of protected methods</td>
</tr>
<tr>
<td>PMC</td>
<td>Public Method Count</td>
<td>Number of public methods</td>
</tr>
<tr>
<td>IMC</td>
<td>Private Method Count</td>
<td>Number of private methods</td>
</tr>
<tr>
<td>SMC</td>
<td>Static Method Count</td>
<td>Number of methods with a static modifier</td>
</tr>
<tr>
<td>FMC</td>
<td>Final Method Count</td>
<td>Number of methods with a final modifier</td>
</tr>
<tr>
<td>YMC</td>
<td>Synchronized Method Count</td>
<td>Number of methods with a synchronized modifier</td>
</tr>
<tr>
<td>NOF</td>
<td>Field Count</td>
<td>Number of fields defined</td>
</tr>
<tr>
<td>PFC</td>
<td>Public Field Count</td>
<td>Number of fields with a public modifier</td>
</tr>
<tr>
<td>IFC</td>
<td>Private Field Count</td>
<td>Number of fields with a private modifier</td>
</tr>
<tr>
<td>RFC</td>
<td>Protected Field Count</td>
<td>Number of fields with a protected modifier</td>
</tr>
<tr>
<td>FFC</td>
<td>Final Field Count</td>
<td>Number of fields with a final modifier</td>
</tr>
<tr>
<td>SFC</td>
<td>Static Field Count</td>
<td>Number of fields defined with a static modifier</td>
</tr>
<tr>
<td>ZFC</td>
<td>Initialized Field Count</td>
<td>Number of fields initialised when declared</td>
</tr>
<tr>
<td>UFC</td>
<td>Uninitialized Field Count</td>
<td>Number of fields uninitialized when declared</td>
</tr>
<tr>
<td>INC</td>
<td>Interface Count</td>
<td>Number of interfaces implemented.</td>
</tr>
<tr>
<td>EXC</td>
<td>Exception Count</td>
<td>Number of exceptions raised by the methods</td>
</tr>
<tr>
<td>CCC</td>
<td>Class Constructor Count</td>
<td>Number of constructors defined. This value will always be $\geq 1$ since the compiler always generates a default constructor, even if one was not provided in the source code.</td>
</tr>
<tr>
<td>Abbv.</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CBC</td>
<td>Try-Catch Block Count</td>
<td>Number of try-catch blocks</td>
</tr>
<tr>
<td>THC</td>
<td>Throw Count</td>
<td>Number of throw statements</td>
</tr>
<tr>
<td>ICC</td>
<td>Inner Class Count</td>
<td>Number of inner classes (counted recursively)</td>
</tr>
<tr>
<td>MCC</td>
<td>Method Call Count</td>
<td>Number of method calls</td>
</tr>
<tr>
<td>MCI</td>
<td>Internal Method Call Count</td>
<td>Number of internal method calls (that is, methods defined in the same class)</td>
</tr>
<tr>
<td>MCE</td>
<td>External Method Call Count</td>
<td>Number of times methods defined outside of the class are invoked</td>
</tr>
<tr>
<td>LVC</td>
<td>Local Variable Count</td>
<td>Number of local variables defined across all methods in the class</td>
</tr>
<tr>
<td>OOC</td>
<td>Instance Of Check Count</td>
<td>Number of times the instanceof operator is used</td>
</tr>
<tr>
<td>CAC</td>
<td>Check Cast Count</td>
<td>Number of times a cast is checked for</td>
</tr>
<tr>
<td>TCC</td>
<td>Type Construction Count</td>
<td>Number of times a new object is created</td>
</tr>
<tr>
<td>CLC</td>
<td>Constant Load Count</td>
<td>Number of constants loaded from a local variable</td>
</tr>
<tr>
<td>PLC</td>
<td>Primitive Load Count</td>
<td>Number of times a primitive is loaded from a local variable</td>
</tr>
<tr>
<td>PSC</td>
<td>Primitive Store Count</td>
<td>Number of times a primitive is stored into a local variable</td>
</tr>
<tr>
<td>ALC</td>
<td>Array Load Count</td>
<td>Number of arrays loaded from a local variable</td>
</tr>
<tr>
<td>ASC</td>
<td>Array Store Count</td>
<td>Number of arrays stored into a local variable</td>
</tr>
<tr>
<td>FLCC</td>
<td>Field Load Count</td>
<td>Number of times an object or primitive is loaded from a field</td>
</tr>
<tr>
<td>FSC</td>
<td>Field Store Count</td>
<td>Number of times an object or primitive is stored into a field</td>
</tr>
<tr>
<td>LIC</td>
<td>Load Count</td>
<td>Total number of load operations (is a sum of PLC, ALC, FLC, and CLC)</td>
</tr>
<tr>
<td>SIC</td>
<td>Store Count</td>
<td>Number of store operations (is a sum of PSC, ASC, and FSC)</td>
</tr>
<tr>
<td>IOC</td>
<td>Increment Operation Count</td>
<td>Number of times the increment operation is used</td>
</tr>
<tr>
<td>ZIC</td>
<td>Zero Operand Instr. Count</td>
<td>Number of bytecode instructions that have no operands</td>
</tr>
<tr>
<td>TTC</td>
<td>Instruction Count</td>
<td>Number of bytecode instructions</td>
</tr>
<tr>
<td>NOB</td>
<td>Branch Count</td>
<td>Number of branch instructions (counts all conditional branches including the cases inside a switch statement as well as for and while loops)</td>
</tr>
<tr>
<td>GTC</td>
<td>Goto Count</td>
<td>Number of times a goto instruction is used (this is generated when the source code contains loop constructs and is generally paired with a branch instruction)</td>
</tr>
</tbody>
</table>

**Table 4.3:** Metrics computed by processing method bodies of each class. The mapping between these measures and the bytecode is presented in Appendix C.
### Table 4.4: Flags extracted for each class.

<table>
<thead>
<tr>
<th>Abbv.</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAS</td>
<td>Is Abstract</td>
<td>A flag that is set to 1, if class is abstract</td>
</tr>
<tr>
<td>IEX</td>
<td>Is Exception</td>
<td>A flag that is set to 1 if class has java.lang.Throwable as an ancestor</td>
</tr>
<tr>
<td>INF</td>
<td>Is Interface</td>
<td>A flag that is set to 1 if class is an interface</td>
</tr>
<tr>
<td>IPI</td>
<td>Is Private</td>
<td>A flag that is set to 1 if class is private</td>
</tr>
<tr>
<td>IPR</td>
<td>Is Protected</td>
<td>A flag that is set to 1 if class is protected</td>
</tr>
<tr>
<td>IPU</td>
<td>Is Public</td>
<td>A flag that is set to 1 if class is public</td>
</tr>
<tr>
<td>IPK</td>
<td>Is Package</td>
<td>A flag that is set to 1 if the class has no visibility modifier and hence would revert to having package visibility</td>
</tr>
</tbody>
</table>

Once information from all the classes is extracted, we remove classes that are from external libraries that are not part of the core software system under study. Though, we ensure that the set of input JAR files used in the Jar Extraction step does not consist of any external libraries (see Section 4.5.1), this additional step was needed since some projects merged all external library code into their core JAR file in order to reduce the number of different files that needed to be distributed. We identify and removed the set of classes that are from external libraries using the same process that we applied during the Jar Extraction step (see Section 4.5.1).

The Java programming language provides developers the option for creating two different types of abstractions: a class, and an interface [108]. Within the context of our study, the key distinction between these two abstractions is that interfaces do not contain any method bodies and hence the metrics defined in Table 4.3 are not available for compiled Java interfaces which do not contain bytecode instructions in the method section of a class file. However, all of the other information (see Table 4.1) is available and therefore used to compute the metrics defined in Table 4.2 and Table 4.4. We are also able to extract dependency information from an interface, that is, other Java classes that an interface depends upon (discussed in Section 4.5.4). In the rest of this thesis, to improve readability, we use the term class to indicate a compiled Java class, and it may be either a Java interface or a Java
Chapter 4. Measuring Evolving Software

class. We use the terms *Java Interface* and *Java Class* where these abstractions are treated separately in our analysis.

### 4.5.3 Merge Inner Classes

In Java, the inner class abstraction is provided as a convenience mechanism to structure the functionality within a class [125]. For example, inner classes are commonly used to implement event handlers in graphical user interface applications. More generally, inner classes are used when an object needs to send another object a block of code that can call the first object’s methods or manipulate its instance variables.

Interestingly, the Java virtual machine specification [180] requires the compiler to emit a separate class file for each class including the inner classes defined within a class (of Java source code). However, semantically developers consider inner-classes to be a part of the parent class within the solution design. Especially since instances of an inner class cannot be instantiated without being bound to an enclosing class.

In the *Metric Extraction* step, inner classes are processed as separate entities since we use compiled class files as input. However, for the purposes of our study, we treat inner classes as part of the parent class and hence merge all metrics collected from inner classes into the parent class. The key benefit gained by merging is that it allows us to focus on the core abstractions within the solution rather than the specific internal representation of a class. However, the trade-off is that we are unable to directly observe the evolution dynamics of inner classes independent of their parent classes.

### 4.5.4 Class Dependency Graph Construction

In order to measure the structural complexity of a software system we construct a complete class dependency graph, $G^T$, and measure certain properties of this graph. When a class uses either data or functionality from another class, there is a dependency between these classes. In the context of Java software, a dependency is created if a class inher-
Figure 4.6: The dependency graph that is constructed includes classes from the core, external libraries and the Java framework. The two sets, $N$ and $K$, used in our dependency graph processing are highlighted in the figure.

its from a class, implements an interface, invokes a method on another class (including constructors), declares a field or local variable, uses an exception, or uses class types within a method declaration. The dependency graph contain nodes representing the classes in the system, and directed links which represent the dependencies between these classes.

During the construction of Java software systems it is common for developers to make use of third-party libraries, as well as functionality provided in the Java framework (which is distributed as part of the Java Runtime Environment). Therefore, a dependency can be created between classes inside the core software system as well as to classes external to the software system. The dependency graph $G_T$ that we construct contains nodes represented by classes from the core software system, external libraries as well as the Java framework.

We capture the relationship between two classes as a directed link (edge) in the dependency graph $G_T$ as this allows us to determine the set of incoming links (the in-degree) into a class as well as the set of outgo-
ing links from a class (the \textit{out-degree}). As discussed in Section 4.2.2, these two measures are widely used in the literature as measures of structural complexity [21, 54, 55, 77, 117, 165, 209, 223, 287, 294, 299] since they naturally capture the number of classes a given class $X$ depends upon and the number of classes that depend on $X$.

For the purpose of measuring the dependencies we define two sets $K$ and $N$. The first set, $K$ is a finite non-empty set of all classes in the software system and includes the classes from the core software system, as well as classes that provide services (to classes in the core) but are in part of third-party libraries or the Java framework. The second set, $N$ contains classes that are part of the core software system such that $N \subset K$. The distinction between these two sets is illustrated in Figure 4.6.

The type dependency graph $G^T$ is an ordered pair $< K, L >$, where $L$ is a finite, possibly empty, set of directed links between classes, such that, $L \subseteq N \times K$. Our restriction of focusing on the core software system (as discussed in Section 3.6.2) implies that we do not measure the dependencies between classes in the external libraries and hence we analyze only the set of links that are formed when classes within the set $N$ depend on classes in the set $K$. For example, the dependency between classes A and B in Figure 4.6 is not part of the graph that we construct. Additionally, classes A and B in Figure 4.6 are also not in the set $K$.

### 4.5.5 Dependency Metric Extraction

Once the dependency graph has been constructed, we can analyze each node $n \in N$ in the graph as well as the set of directed links $l \in L$ for each node within the graph $G^T$ and measure the \textit{In-Degree Count} $l_{\text{in}}(n)$, as well as the \textit{Out-Degree Count} $l_{\text{out}}(n)$ it. More precisely,

\[
l_{\text{in}}(n) = |\{(n_i, n) \land n_i \in N \land n \neq n_i\}|
\]  

\[\text{Eq. (4.5.1)}\]
<table>
<thead>
<tr>
<th>Abbv.</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODC</td>
<td>Out Degree Count</td>
<td>Number of classes that this class depends upon. Metric is defined by $l_{out}(n)$ and the values are within the interval $[0,</td>
</tr>
<tr>
<td>IDC</td>
<td>In Degree Count</td>
<td>Number of classes that depend on this class. Metric is defined by $l_{in}(n)$ and the values are within the interval $[0,</td>
</tr>
<tr>
<td>EDC</td>
<td>External Out Degree Count</td>
<td>Number of classes that this class depends upon, but belong in external libraries. Metric is defined by $l_{out}^e(n)$ and the values are within the interval $[0,</td>
</tr>
<tr>
<td>TDC</td>
<td>Internal Out Degree Count</td>
<td>Number of classes that depend on this class and are part of the core software system. Metric is defined by $l_{out}^i(n)$ and the values are within the interval $[0,</td>
</tr>
</tbody>
</table>

Table 4.5: Dependency metrics computed for each class.

\[
l_{out}(n) = | \{(n,n_j) \land n_j \in K \land n \neq n_j\} | \tag{4.5.2}
\]

The *In-Degree Count* is a measure of the “popularity” of node $n$ in the graph $G^T$ whereas the *Out-Degree Count* is node $n$’s “usage” of other types in the graph $G^T$ [223].

We further refine the notions of in-degree and out-degree in the context of our analysis by considering dependencies to classes in external libraries. These external dependencies give rise to a refinement of the measures in-degree and out-degree in which we also distinguish between *intra*- and *inter*-system links. A given link to or from a node $n$ may or may not cross the boundary of the containing core system, depending on some organizational, structural, and/or functional features. If an outbound link from node $n$ ends in a node $n_{int}$ that occurs within the boundary of the system under analysis, then we call this link an *internal* outbound link. On the other hand, if an outbound link ends in a node $n_{ext}$ that lies outside of the system’s boundary, then we call
Figure 4.7: Class diagram showing dependency information to illustrate how dependency metrics are computed. The metrics for the various classes shown in the table below the diagram.

<table>
<thead>
<tr>
<th>Class</th>
<th>IDC</th>
<th>ODC</th>
<th>EDC</th>
<th>TDC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassM</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>ClassN</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>ClassO</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ClassP</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>ClassQ</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

This link an external outbound link. An example of an outbound links is a dependency on \texttt{java.lang.Math}, since this class is defined in the Java framework. More precisely,

\begin{equation}
I^e_{out}(n) = \left| \{(n, n_{ext}) \land n_{ext} \in K \setminus N\} \right| \quad (4.5.3)
\end{equation}

\begin{equation}
I^i_{out}(n) = \left| \{(n, n_{int}) \land n_{int} \in N \land n \neq n_{int}\} \right| \quad (4.5.4)
\end{equation}
### Table 4.6: Inheritance metrics computed for each class.

<table>
<thead>
<tr>
<th>Abbv.</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCC</td>
<td>Super Class Count</td>
<td>Counted as 0 if super class is java.lang.Object, else 1.</td>
</tr>
<tr>
<td>NOC</td>
<td>Number of Children</td>
<td>Count of classes that directly inherit from this class. Metric value is in the interval ([0, N]).</td>
</tr>
<tr>
<td>NOD</td>
<td>Number of Descendants</td>
<td>Count of all classes that inherit from this class. Computed by walking down the inheritance tree. Metric value is in the interval ([0,</td>
</tr>
<tr>
<td>DIT</td>
<td>Depth in Inheritance Tree</td>
<td>If the class has no parent in core software system then the value is 1, otherwise it is 1+depth of inheritance of direct parent.</td>
</tr>
</tbody>
</table>

\[
l_{out}(n) = l^{c}_{out}(n) + l^{i}_{out}(n)
\]  \hspace{1cm} (4.5.5)

The dependency metrics collected for each class and the abbreviation used for the metrics are presented in Table 4.5. While determining the dependencies between classes we ignore all dependency links into java.lang.Object since all objects in Java inherit from this class. By ignoring this default link we are able to determine if there are classes that do not have any outgoing links to other objects, that is, Out-Degree Count can be zero for some classes. Furthermore, having a potential zero value for the dependency metrics simplifies the statistical analysis that we undertake in our study (discussed in further detail in Chapter 5 and Chapter 6).

We illustrate how our dependency metrics are computed by using an example class diagram (see Figure 4.7 showing both the figure and the metrics computed, and Table 4.5 presents the full form of the abbreviations used). In this thesis, we consider the dependency metrics that we extract to be a measure of structural complexity since they capture the degree of inter-connectedness between classes (as discussed earlier in Section 4.2.2).
Figure 4.8: Class diagram to illustrate how inheritance metrics are computed. The metrics for the diagram shown in the table below the diagram.

4.5.6 Inheritance Metric Extraction

The final step in our metric extraction process focuses on measuring inheritance. The set of inheritance measures that we compute are listed and explained in Table 4.6. We illustrate how our inheritance metrics are computed by using an example class diagram (see Figure 4.8 showing both the figure and the metrics computed). Since we do not process classes external to the core software system, the inheritance measures that we compute may not include the complete set of ancestors for any given class in a software system. For example, consider a class `ReportView` that extends the class `javax.swing.JFrame` which is part of the Java framework. We compute the inheritance metric Depth
of Inheritance Tree to have a value of 1. However, if the Java framework was fully analysed then this would change to 6 (for Java 1.6), since we now are processing the full extent of the inheritance chain by extracting additional information from the external Java framework. Though our metrics are constrained, the inheritance hierarchy within the external framework was designed and created by an external team and hence all changes to it are outside of the direct control of the development team creating the software under study. Hence, we do not measure the the inheritance hierarchies of external libraries and the core Java framework as part of our analysis. Furthermore, we do not consider interface implementation as inheritance and hence do not count them in our metrics.

4.6 Summary

The evolution of a software system can be studied in terms of how various properties as reflected by software metrics change over time. We build a release history model by analysing the compiled class files. Our release history model captures meta-data and 58 different metrics at a class level. We also build a class dependency graph for each release in the evolution history.

The data selection and metric extraction method that we use ensures that we study non-trivial software allowing us to extend our findings to other comparable software systems built in Java. We also analyse compiled binaries that have already gone through the build process improving the accuracy of our measures. Further, as discussed in the previous chapter, we focus on contributions from the core development team ignoring third party libraries ensuring that the metrics that we collect are a better reflection of the development effort.

The next chapter (Growth Dynamics) addresses the research questions related to growth. We describe how size and complexity is distributed as systems evolve and present a novel analysis technique to help understand growth dynamics.
Chapter 5

Growth Dynamics

Current models of software evolution [118, 119, 127, 153, 168, 175, 200, 217, 239, 277, 284, 305, 310] have allowed for inferences to be drawn about certain attributes of the software system, for instance, regarding the architecture [100, 101, 127, 153, 192, 200], complexity and its impact on the development effort [118, 168, 284]. However, an inherent limitation of these models is that they do not provide any direct insight into where growth takes place. In particular, we cannot assess the impact of evolution on the underlying distribution of size and complexity among the various classes. Such an analysis is needed in order to answer questions such as “do developers tend to evenly distribute complexity as systems get bigger?” and “do large and complex classes get bigger over time?” These are questions of more than passing interest since by understanding what typical and successful software evolution looks like, we can identify anomalous situations and take action earlier than might otherwise be possible. Information gained from an analysis of the distribution of growth will also show if there are consistent boundaries within which a software design structure exists.

The specific research questions that we address in this chapter are:

• What is the nature of distribution of software size and complexity measures?
• How does the profile and shape of this distribution change as software systems evolve?

• Is the rate and nature of change erratic?

• Do large and complex classes become bigger and more complex as software systems evolve?

The typical method to answer these questions is to compute traditional descriptive statistical measures such as arithmetic mean (referred to as “mean” in this thesis to improve readability), median and standard deviation on a set of size and complexity measures and then analyze their changes over time. However, it has been shown that software size and complexity metric distributions are non-gaussian and are highly skewed with long tails [21, 55, 270]. This asymmetric nature limits the effectiveness of traditional descriptive statistical measures such as mean and standard deviation as these values will be heavily influenced by the samples in the tail making it hard to derive meaningful inferences.

Recently advocated alternative method to analyze metric distributions [21, 55, 118, 223, 270, 299] involves fitting metric data to a known probability distribution. For instance, statistical techniques can be used to determine if the metric data fits a log-normal distribution [55]. Once a strong fit is found, we can gain some insight into the software system from the distribution parameters. Unfortunately, the approach of fitting data to a known distribution is more complex and the metric data may not fit any known and well understood probability distributions without a transformation of the data.

Software metrics, it turns out, are distributed like wealth in society — where a few individuals have a high concentration of wealth, while the majority are dispersed across a broad range from very poor to what are considered middle class. To take advantage of this nature, we analyze software metrics using the Gini coefficient, a bounded higher-order statistic [191] widely used in the field of socio-economics to study the distribution of wealth and how it changes over time. Specifically it is
used to answer questions like “are the rich getting richer?”. Our approach allows us not only to observe changes in software systems efficiently, but also to assess project risks and monitor the development process itself. We apply the Gini coefficient to 10 different metrics and show that many metrics not only display consistently high Gini values, but that these values are remarkably consistent as a project evolves over time. Further, this measure is bounded (between a value of 0 and 1) and when observed over time it can directly inform us if developers tend to centralise functionality and complexity over time or if they disperse it.

The rest of this chapter is structured as follows: Section 5.1 presents an overview of the nature of the software metric data and summarises the current approaches used to analyse this data and their limitations. Section 5.2 presents the Gini Coefficient that we use to understand software metric data and show how it overcomes the limitations of the statistical techniques applied in work by other researchers. Section 5.3 presents the analysis approach and shows how we apply the Gini Coefficient to address the research questions. Section 5.4 summarises the observations while Section 5.5 discusses our findings and offers an interpretation.

The raw data used is this study is available as data files on the DVD attached to this thesis. Appendix E describes the various data and statistical analysis log files related to this chapter.

5.1 Nature of Software Metric Data

A general characteristic of object oriented size and complexity metrics data is that they are heavily skewed with long-tails [21, 55, 118, 223, 270, 299]. It has been shown that small values are extremely common, while very large values can occur, they are quite rare. Typically software systems follow a simple pattern: a few abstraction contain much of the complexity and functionality, whereas the large majority tend to define simple data abstractions and utilities. This pattern is illustrated in Figure 5.1 with one metric, the Number of Methods in a class for
version 2.5.3 of the Spring Framework. As can be observed in the figure, approximately 20% of the classes have more than 10 methods suggesting that relatively few classes have a large number of methods in this system. This skewed metric distribution pattern repeats for the different metrics that we collect in our study across all the software systems (discussed further in Section 5.4.2).

### 5.1.1 Summarising with Descriptive Statistics

The sheer volume of metric data available from any object-oriented software systems can make it difficult to understand the nature of software systems and how they have evolved [75]. A common approach [77, 117, 165, 182] to reducing the complexity of the analysis is to apply some form of some simple statistical summarisation such as the
mean, median, or standard deviation. Unfortunately, these descriptive statistics provide little useful information about the distribution of the data, particularly if it is skewed, as is common with many software metrics [21,55,118,223,270,299]. Furthermore, the typical long-tailed metric distributions makes precise interpretation with standard descriptive statistical measures difficult.

Commonly used summary measures such as “arithmetic mean” and “variance” capture the central tendency in a given data set. However, where the distribution is strongly skewed, they become much less reliable in helping understand the shape and changes in the underlying distribution. Moreover, additional problems may arise due to changes in both the degree of concentration of individual values and and the population size. Specifically, since these summary measures are influenced by the population size which tends to increase in evolving software systems.

Descriptive statistics such as median and variance are also likely to be misleading, given the nature of the underlying distribution. Specifically, we found that the median measure does not change substantially over time reducing its effectiveness when applied to understanding software evolution. An example of this is illustrated in Figure 5.2, where the median of three different metrics is shown for PMD. As can be seen in the figure, the median value is very stable over a period of nearly 5 years of evolution. Though there is some change (to the median), in absolute terms the value does not convey sufficient information about the nature and dynamics of the evolution.

Additional statistics such as the skew, which measures the asymmetry of the data, and kurtosis, which measures the peakedness of the data, may be applied, but are ineffective for comparison between systems with different population sizes as these measures are unbounded and change depending on the size of the underlying population, making relative comparisons ineffective [221]. Given this situation, it is not surprising that metrics use in industry is not widespread [137]. This situation is also not helped by the current generation of software metric tools as many commercial and open source tools [47,51,196,203,

Comparison of different distributions may provide some insight, but require skill to interpret, particularly given the range of measures that can be used and the different population sizes that might be encountered.

### 5.1.2 Distribution Fitting to Understand Metric Data

A more recent method to understand software metric data distribution involves fitting the data to a known probability distribution [21, 55, 118, 223, 270, 299]. For example, statistical techniques can be used to determine if the Number of Methods in a system fits a log-normal distribution [55]. The motivation for fitting metrics to known distributions is driven by the notion that it can help explain the underlying processes that might be causing specific distributions [209, 287]. Furthermore, if
the fit is robust and consistent we can infer information from the distribution parameters as they summarise the data and can gain an insight into the evolution by observing changes to the distribution parameters over time.

Some of the early work on understanding object-oriented software metric data by fitting it to a distribution was conducted by Tamai et al. [269,270] who have observed that the size of methods and classes (measured using lines of code) within a hierarchy fit the negative-binomial distribution. Recently, researchers inspired by work in complex systems [209,287] (especially, real-world networks) have attempted to understand software metric distributions as power-laws. Baxter et al. [21] studied 17 metrics in a number of Java software systems and have shown that some metrics fit a log-normal distribution, while others fit a power-law distribution, and also that some metrics did not fit either of these distributions. Potanin et al. [223] investigated object graphs by analysing run-time data, and found that incoming and outgoing references fit a power law distribution. Wheeldon et al. [299] investigated the Java Development Kit and found 12 metrics fit power-law distribution. In a detailed case study of Visual Works Smalltalk, Java Development kit and Eclipse IDE, Concas et al. [54] observe that out-degree measures of the class graphs and Class Lines of Code fit a log-normal distribution, while method lines of code and in-degree measures of a class graph fit a Pareto distribution. Herraiz [118] investigated the distribution of SLOC (Source Lines of Code) in 12,010 packages available for the FreeBSD software system and found that SLOC fitted a double pareto distribution. The common element in all of these studies is that software metric distributions are non-gaussian and tended to be positively skewed with long tails. Unfortunately, these studies have not been able to identify a consistent probability distribution that can be expected for a certain metric.

Despite consistent results that find skewed distributions when a robust fit is found, the methods used to fit the distributions have certain inherent weaknesses and limitations. In order to fit many of these distributions, the raw data is often transformed since software metric data has a large number of zero values. For instance, it is common to
have a set of classes with no dependents or methods with no branching statements. These zero values need to be eliminated as log-normal, pareto and power-law distributions only work with data that has positive values. However, the impact of these transformations, if any, is not properly represented in the studies [21, 54, 223, 270, 299]. Furthermore, once data is transformed, this aspect has to be considered when deriving any inferences. Recently, a weakness of the approach with respect to fitting power-laws has been put forward by Goldstein et al. [104] as well as Clauset et al. [48]. They argue that the widely used approach of fitting power laws using a graphical linear fit of data transformed into a log-log scale is biased and inaccurate, especially since there is no quantitative measure of the goodness-of-fit that is used in this approach. This limitation would apply to the work by Wheeldon et al. [299] as they use a direct linear-fit of the log-log plot of the full raw histogram of the data. Potanin et al. [223] and Concas et al. [55] also use a linear fit of the logarithmically binned histogram which limits the power and conclusions in their studies [48].

Another limitation is that we cannot use these distributions for a meaningful comparison between software systems or different releases of the same software system. This is because the distributions are created by estimating the parameters from the underlying raw data rather than from empirically derived tables. Further, the value of fitting metric data to known distributions in order to infer the underlying generative process has not yet been properly established [210], especially since multiple non-correlated processes have been shown to generate the same distribution. Interestingly, this limitation is acknowledged by Concas et al. [54, 55], but they present their work of fitting metric data to a distribution as valuable since it provides a broad indication of a potential underlying process and more importantly can indicate presence of extreme values. A similar argument is also extended by Valverde et al. [287]. The common approach used by these studies based on the analysis of a metric data distribution is to infer the underlying generative process by investigating a single release. For instance, Concas et al. [55] argue that the presence of these skewed distributions in software denotes that the programming activity cannot be considered to be a process involving random addition of independent increments but
exhibits strong organic dependencies on what has been already developed. Though, fitting distributions has been shown to have merit for modeling networks [210] and to infer how these networks have been created, software evolution is better modelled by analysing the evolution history as we can reduce the number of assumptions one has to make. Rather than attempting to infer the generative process from a single release of a software system, we can gain more insight into the evolutionary pressures by analysing the changing metric distribution over time. In our work, we take this approach and study the metric distributions as they change over time in order to gain a better understanding of the underlying evolutionary processes.

Though there has been progress over the last decade in this field, there is still no widely-accepted distribution that captures consistently and reliably software metric data. But more importantly, we are not required to fit a given software metric to particular distributions in order to interpret it. What is needed is a set of measures that reliably and consistently summarize properties of the distribution allowing for effective inferences to be made about the evolution of a software system.

5.2 Summarizing Software Metrics

Given the skewed nature of metric data we are in need of methods that can effectively summarise this data and provide effective insight into the current state of a software system as well as detect worthwhile changes as the software evolves. In this section we introduce the Gini Coefficient, a measure that is effective when dealing with metric data and motivate its applicability for analysing evolving metric data distributions.

5.2.1 Gini Coefficient - An Overview

One of the key pieces of information we wish to obtain from software metrics is the allocation of functionality within the system. Understanding whether the system has a few classes that implement most of
Chapter 5. Growth Dynamics

the methods or whether methods are widely distributed gives us an insight into how the system has been constructed, and how to maintain it [66]. A technique to study allocation of some attribute within a population and how it changes over time has been studied comprehensively by economists who are interested in the distribution of wealth and how this changes [311] – we use the same approach in our analysis.

In 1912, the Italian statistician Corrado Gini proposed the Gini coefficient, a single numeric value between 0 and 1, to measure the inequality in the distribution of income or wealth in a given population (cp. [91, 229]). A low Gini coefficient indicates a relatively equal wealth distribution in a given population, with 0 denoting a perfectly equal wealth distribution (i.e., everybody has the same wealth). A high Gini coefficient, on the other hand, signifies a very uneven distribution of wealth, with a value of 1 signalling perfect inequality in which one individual possesses all of the wealth in a given population. The Gini Coefficient is a widely used social and economic indicator to ascertain an individual’s ability to meet financial obligations or to correlate and compare per-capita GDPs [286].

We can adopt this technique and consider software metrics data as income or wealth distributions. Each metric that we collect for a given property, say the number of methods defined by all classes in an object-oriented system, is summarized as a Gini coefficient, whose value informs us about the degree of concentration of functionality within a given system.

### 5.2.2 Computing the Gini Coefficient

Key to the analysis of the distribution of data and computation of the Gini Coefficient is the Lorenz curve [183], an example of which is shown in Figure 5.3.

A Lorenz curve plots on the y-axis the proportion of the distribution assumed by the bottom x% of the population. The Lorenz curve gives a measure of inequality within the population. A diagonal line represents
perfect equality. A line that is zero for all values of $x < 1$ and 1 for $x = 1$ is a curve of perfect inequality.

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

$$L(F(x)) = \frac{\int_{-\infty}^{x} t f(t) \, dt}{\int_{-\infty}^{\infty} t f(t) \, dt}$$  \hspace{1cm} (5.2.1)

The Lorenz curve can be used to measure the distribution of functionality within a system. Figure 5.3 is a Lorenz curve for the Fan-Out Count metric in the Spring framework release 2.5.3. Although the Lorenz curve does capture the nature of distribution, it can be summarized more effectively by means of the Gini coefficient. The Gini coefficient is defined as a ratio of the areas on the Lorenz curve diagram. If the area between the line of perfect equality and Lorenz curve is $A$, and the area under the Lorenz curve is $B$, then the Gini coefficient is $A / (A + B)$ [311].

More formally, if the Lorenz curve is $L(Y)$, then the Gini Coefficient is defined as:
Chapter 5. Growth Dynamics

\[ GiniCoefficient = 1 - 2 \int_0^1 L(Y) \, dY \quad (5.2.2) \]

For a population of metric data \( x_i, i = 1 \text{ to } n \), that are indexed in an increasing order such that \( x_i \leq x_{i+1} \), the Gini Coefficient \( G \) is computed as:

\[ G = \frac{1}{n} (n + 1 - 2 \left( \frac{\sum_{i=1}^{n} (n + 1 - i) x_i}{\sum_{i=1}^{n} x_i} \right)) \quad (5.2.3) \]

The Gini Coefficient is a higher order statistic as it is derived from the Lorenz curve, which itself is a summary measure computed over a cumulative probability distribution function.

### 5.2.3 Properties of Gini Coefficient

The Gini coefficient has a number of useful properties in that it is bounded between 0 and 1, makes no assumptions as to the distribution of the statistic under investigation, and can be compared between different sized populations. These properties make it an ideal statistic for comparing the distribution of metrics between software systems as well as multiple releases of an evolving software system. Moreover, the Gini coefficient provides a simple and intuitive means for qualitative analysis of observed software properties.

The Gini Coefficient summarises data independent of its distribution. For example, if the data has a gaussian distribution then the Gini Coefficient value will be around 0.10 (exact value will depend on the shape of the distribution). This is the case, since nearly half of the data have a very similar range of values and hence, there is minimal inequality in the data. However, when we compute the Gini Coefficient value for a highly skewed log-normal distribution where there is substantial inequality in the data, the value will be typically well over 0.5.
5.2.4 Application of the Gini Coefficient - An Example

We illustrate the applicability and simplicity of using the Gini Coefficient coefficient to measure the inequality within a software metric distribution with an example. We first present the Gini Coefficient of In-Degree Count for the Spring Framework and then contrast it with the standard descriptive statistical measure of arithmetic mean, median, standard deviation and skewness for In-Degree Count.

In the first version of the Spring Framework, the Gini Coefficient value for In-Degree Count is 0.62. Values of Gini Coefficient substantially greater than 0 are indicative of a skewed distribution, where a small set of classes are very popular (since In-Degree Count is the wealth in our case). Furthermore, in Spring Framework, the Gini Coefficient value gradually increases over time from 0.62 to 0.71 over a period of 4 years of evolution. The trend shows that over time, a small set of classes are gaining popularity. This type of trend analysis is used by economists to answer the question “are the rich getting richer?”.

In contrast to the Gini Coefficient, in the Spring Framework, the median value of In-Degree Count has remained unchanged at 1 for its entire evolution history, while the mean has increased slightly from 2.3 to 3.3. Neither of which provide us with sufficient information about the skew in the distribution and the fact that the a few classes have slowly gained in popularity. The standard deviation measure has also increased gradually from 3.74 to 11.5. The standard deviation measure provides some indication that the underlying data might have a few strong outliers, however it does not inform us of the shape of the distribution. The statistical measure of skewness can be used to gain additional information that can indicate the shape of the distribution. The measure of skewness for In-Degree Count increases from 4.32 to 18.10. The positive values for skewness do reveal that we have a long tail to the right of the distribution with some classes that have very high In-Degree Count. We can also infer from the increasing skewness that we have a potential increase in the upper bound, that is the highest In-Degree Count value is increasing. However, as discussed earlier, the measure of skewness is influenced by the population size making the interpretation difficult.
without further analysis. Although, the different descriptive statistical measures (arithmetic mean, standard deviation, skewness) can be combined to gain a picture of the underlying distribution, the Gini Coefficient provides a simpler and more direct method to measure the inequality in a distribution.

### 5.3 Analysis Approach

In this section we present our analysis method. We first introduce the set of metrics that were analysed using the Gini coefficient, followed by the various steps in our analysis. We describe the observations arising from our analysis in Section 5.4 and present a discussion of our findings in Section 5.5.

#### 5.3.1 Metrics Analyzed

In our study of metric distributions, we focused on 10 different measures that span a range of size and complexity measures. The selected measures and a brief description of these metrics is provided in Table 5.1.

In order to assess assigned responsibilities we use the two metrics *Load Instruction Count* and *Store Instruction Count*. Both metrics provide a

---

<table>
<thead>
<tr>
<th>Name</th>
<th>Rationale</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Instruction Count (LIC)</td>
<td>Responsibility</td>
<td>Number of read instructions</td>
</tr>
<tr>
<td>Store Instruction Count (SIC)</td>
<td>Responsibility</td>
<td>Number of write instructions</td>
</tr>
<tr>
<td>Number of Branches (NOB)</td>
<td>Complexity</td>
<td>Degree of algorithmic branching</td>
</tr>
<tr>
<td>In-Degree Count (IDC)</td>
<td>Popularity</td>
<td>Number of classes depending on this class</td>
</tr>
<tr>
<td>Out-Degree Count (ODC)</td>
<td>Delegation</td>
<td>Number of classes used</td>
</tr>
<tr>
<td>Number of Methods (NOM)</td>
<td>Decomposition</td>
<td>Breadth of functional decomposition</td>
</tr>
<tr>
<td>Public Method Count (PMC)</td>
<td>Interface Size</td>
<td>Exposure of responsibility</td>
</tr>
<tr>
<td>Number of Attributes (NOA)</td>
<td>Information Storage</td>
<td>Density of information stored in class</td>
</tr>
<tr>
<td>Fan-Out Count (FOC)</td>
<td>Delegation</td>
<td>Degree of dependence on others</td>
</tr>
<tr>
<td>Type Construction Count (TCC)</td>
<td>Composition</td>
<td>Number of object instantiations</td>
</tr>
</tbody>
</table>

Table 5.1: Collected measures for distribution and change analysis using the Gini Coefficient
measure for the frequency of state changes in data containers within a system. *Number of Branches*, on the other hand, records all branch instructions and is used to measure the structural complexity at class level. This measure is equivalent to Weighted Method Count (WMC) as proposed by Chidamber and Kemerer [46] if a weight of 1 is applied for all methods and the complexity measure used is cyclomatic complexity [190]. We use the measures of *Fan-Out Count* and *Type Construction Count* to obtain insight into the dynamics of the software systems. The former offers a means to document the degree of delegation, whereas the latter can be used to count the frequency of object instantiations.

The remaining metrics provide *structural* size and complexity measures. *In-Degree Count* and *Out-Degree Count* reveal the coupling of classes within a system. As discussed in Chapter 4, these measures are extracted from the type dependency graph that we construct for each analyzed system. The vertices in this graph are classes, whereas the edges are directed links between classes. We associate *popularity* (i.e., the number of incoming links) with *In-Degree Count* and usage or delegation (i.e., the number of outgoing links) with *Out-Degree Count*. *Number of Methods*, *Public Method Count*, and *Number of Attributes* define typical object-oriented size measures and provide insights into the extent of data and functionality encapsulation.

### 5.3.2 Metric Correlation

A natural consequence of selecting a range of metrics is to see if a smaller sub-set of these metrics would be sufficient. That is, do the selected measures all represent independent characterizing properties? We need to examine, therefore, all selected metrics more closely and check whether there exists a *relationship* between any of them. If we discover a consistent and strong relationship between two measures, we may be able to eliminate one metric when it does not provide additional insights.

In order to reduce the number of metrics needed, we use the technique of checking collinearity as recommended by Succi et al. [265] for sim-
plifying models constructed to understand software. Similar to Succi et al. [265], we compute the Spearman’s rank correlation coefficient $\rho$ and applied the \textit{t-test} to check if the reported coefficient is different from zero at a significance level of 0.05 for all 10 measures in all systems. The \textit{t-test} checks that the reported relationship between the Gini Coefficient and Age (days since birth) can be considered to be statistically significant, while the correlation coefficient reports the strength of the relationship between the Gini Coefficient and Age. The non-parametric Spearman’s correlation coefficient was selected over Pearson’s correlation coefficient since it does not make any assumptions about the distribution of the underlying data [279], specifically it does not assume that the data has a gaussian distribution.

5.3.3 Checking Shape of Metric Data Distribution

A consistent finding by other researchers [21, 55, 223, 270, 299] studying software metric distributions has been that this data is positively skewed with long-tails. Can we confirm this finding in our own data? Further, will this shape assumption hold if metric data was observed over time? We undertook this step in order to provide additional strength to the current expectation that metric data is highly skewed.

For a population with values $x_i$, $i = 1$ to $n$ with a mean of $\mu$ and a standard deviation of $\sigma$,

$$
MovementSkewness = \frac{1}{n} \sum_{i=1}^{n} \frac{(x_i - \mu)^3}{\sigma^3}
$$

(5.3.1)

In our analysis, we tested the metric data for each release over the entire evolution history to ensure that the data did not have a gaussian distribution by using the Shapiro-Wilk \textit{goodness of fit} tests for normality [279] at a significance level of 0.05. The expectation is that the test will show that the metric data is not normally distributed. Additionally, to confirm that the distribution can be considered skewed we computed the descriptive measure of \textit{movement skewness} (See Equa-
tion 5.3.1) [121]. The skewness measure was computed to determine if the data was positively skewed. A skewness value close to zero is an indicator of symmetry in the distribution, while values over 1.0 are used as an indicator of a moderate level of skewness in the underlying metric data, and values over 3.0 are observable in data with significant degree of skew [121]. The value of skewness is not bounded within a range and hence the degree of skewness can only be interpreted qualitatively.

5.3.4 Computing Gini Coefficient for Java Programs

We compute the Gini Coefficient for each of the selected metrics (see Table 5.1) using the formula in Equation 5.2.3. There were, however, some minor adjustments made to the calculation after taking into consideration certain Java language features. When we process code for metric extraction, we treat both Java classes and Java interfaces as abstractions from which we can collect metrics (see Chapter 4). However, Interfaces in Java are unable to include load or store actions, branching, method invocations, or type constructions, respectively. As a result, interfaces were excluded from these counts, but were included in the Out-Degree Count, In-Degree Count, Number of Methods, and Public Method Count measures. While interfaces in Java may include constant field declarations [108], it was decided to also exclude them from the Number of Attributes measure in order to focus more directly on field usage within individual classes.

5.3.5 Identifying the Range of Gini Coefficients

In our study, we use the Gini coefficient as a means to summarise the metric data distribution. But, do different systems have a similar range of Gini Coefficient values for any given metric? Though, the current state of the art has not been able to establish if a certain class of probability distribution functions fit metric data, a narrow boundary for the Gini Coefficient values across different systems in our data set will confirm certain consistency among how developers organise software solutions. For instance, if the Gini coefficient for Load Instruction
Chapter 5. Growth Dynamics

Count is in a very narrow boundary when measured across a range of software systems, it will be an indication that there are certain underlying distribution preferences that are not directly problem domain dependent.

Similarly, if the Gini Coefficient values of any system (for a selected metric) are within a very narrow boundary over the entire evolution history, then it is an indicator of organisation stability at the system level. For example, consider a software system which has increased in size by 300%, but the Gini Coefficient for Number of Branches is between 0.78 and 0.81 over an evolution of 4 years. This minimal fluctuation can be seen as an indicator of stability in how developers organise structurally complex classes. However, if Gini Coefficient values change substantially over the evolution period across different systems then it is an indication that evolutionary pressures do play a role in how developers organise the solutions.

We compute the range (difference between minimum and maximum) as well as the Inter-Quartile range of Gini Coefficient values for each metric and each system in order to identify any typical boundaries for a specific metric across all systems.

5.3.6 Analysing the Trend of Gini Coefficients

We analyse the trends in Gini Coefficient values over time to answer one of our research questions - do developers create more complex and large classes over time?. If we consistently observe that the value of the Gini Coefficients increase over time, this is a strong indicator that developers do tend to centralise functionality into a few complex and large abstractions as software evolves. However, if the Gini Coefficients decrease over time, this then suggests that there are pressures that compel development teams to reorganise the responsibilities more evenly. However, a third alternative is that developers have a certain set of habitual preference and that software evolution does not impact on the underlying distribution significantly – that is, the Gini Coefficients do not change substantially over time. If the Gini Coefficients consistently do not show
any substantial trend, it is an indication that there is a preference by developers towards a certain shape profile and the process of evolution does not have any impact of the underlying distribution.

In order to identify if there was a sufficiently strong trend, we compute the Spearman’s rank correlation coefficient $\rho$ [279] between Gini Coefficient values (for each metric) over Age (measured in days since birth) for each system in our data set. We applied the t-test to ensure that the reported Spearman correlation coefficient values were significantly different from zero.

Figure 5.4: Correlation coefficient distributions across all systems and releases. The top graph shows the box-plots for each of the 10 metrics under analysis. The bottom graph plots the distribution for the metrics.
Chapter 5. Growth Dynamics

<table>
<thead>
<tr>
<th>Metric</th>
<th>LIC</th>
<th>SIC</th>
<th>NOB</th>
<th>IDC</th>
<th>ODC</th>
<th>NOM</th>
<th>PMC</th>
<th>NOA</th>
<th>FOC</th>
<th>TCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIC</td>
<td>-</td>
<td>0.93</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIC</td>
<td>0.80</td>
<td>0.81</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOB</td>
<td>0.18</td>
<td>0.24</td>
<td>0.15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDC</td>
<td>0.86</td>
<td>0.82</td>
<td>0.77</td>
<td>0.08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ODC</td>
<td>0.71</td>
<td>0.66</td>
<td>0.44</td>
<td>0.26</td>
<td>0.63</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOM</td>
<td>0.21</td>
<td>0.20</td>
<td>0.09</td>
<td>0.27</td>
<td>0.18</td>
<td>0.75</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMC</td>
<td>0.70</td>
<td>0.79</td>
<td>0.48</td>
<td>0.23</td>
<td>0.50</td>
<td>0.53</td>
<td>0.16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOA</td>
<td>0.96</td>
<td>0.87</td>
<td>0.80</td>
<td>0.09</td>
<td>0.85</td>
<td>0.62</td>
<td>0.16</td>
<td>0.67</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOC</td>
<td>0.78</td>
<td>0.78</td>
<td>0.53</td>
<td>0.11</td>
<td>0.68</td>
<td>0.56</td>
<td>0.16</td>
<td>0.82</td>
<td>0.84</td>
<td></td>
</tr>
<tr>
<td>TCC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: Spearman’s Rank Correlation Coefficient values for one version (0.3.0) of JasperReports. Strong correlation values are highlighted.

5.4 Observations

In this section we present our observations within the context of the key research questions. The findings are summarised in Section 5.4.8 and discussed in Section 5.5.

5.4.1 Correlation between measures

As an initial step in our analysis to identify if a subset of the measures selected could be eliminated, we ran a correlation analysis between the 10 metrics under investigation (see Table 5.1). The range of the observed correlation coefficient values is summarized (graphically) in Figure 5.4 using both a box-plot as well as a histogram. The figure shows, for each metric, the distribution of the correlation coefficient against the other 9 metrics.

Our observations are summarised as follows:

- There exists a strong positive correlation (*i.e.*, > 0.8 [265]) between some different measures consistently across our entire data set. The high correlation coefficient values can be seen in skewed histograms chart in Figure 5.4 for most metrics. Across all system, except for *In-Degree Count* and *Public Method Count* the correlation coefficient values are in general very high.
The strength of the relationship varies across systems and versions. For example, the measures *Load Instruction Count* and *Fan-Out Count* are strongly correlated in JasperReports 0.3.0 (see Table 5.2), but this relationship between *Load Instruction Count* and *Fan-Out Count* is not as strong in other systems and releases.

Across all systems, the measure *In-Degree Count* (IDC) consistently shows the weakest correlation to other metrics suggesting that in general, the popularity of a class is not a monotonic function of the other metrics. This can be seen in Figure 5.4, where the IDC metric value box plot as well as the distribution plot are significantly different to that of all other metrics. Additionally, the outliers shown in the box plot (outside the whiskers) are caused by the IDC metric in all the other measures.

Except for *In-Degree Count*, in 75% of the releases all other measures show moderate to high positive correlation (i.e. $> 0.6$) between different measures.

*Load Instruction Count* and *Store Instruction Count* are in general strongly correlated (i.e., over 0.8). This signifies that data often requires a pair-wise read and write. However, there was one system, rssOwl where the correlation was consistently weak. In rssOwl the correlation coefficient value between *Load Instruction Count* and *Store Instruction Count* is below 0.5 during the entire evolution history, which is well below the typical expectation of a value well over 0.8. The low correlation value observed in rssOwl was caused by many classes loading a disproportionate amount of string constants in the user interface classes as well as in classes providing internationalization support. The typical strategy employed to load large number of strings is to load the data from external resource configuration files rather than by hard coding them in the source code.

Our observations suggest that there is a consistent correlation between the various internal size and internal structural complexity metrics of a class. However, the popularity of a class (as measured by IDC) is not a monotonic function of its size or internal complexity indicating that
large and complex classes need not directly service a large number of other classes.

### 5.4.2 Metric Data Distributions are not Normal

Software systems that we analysed contained many hundreds of classes. But how are they distributed? Are they highly skewed, as found by other researchers? When we analysed this data, we found that our observations confirm findings from other researchers [21, 55, 223, 270, 299], in that they do not fit a gaussian distributions. Further, we consistently found positive values for skewness clearly indicating that in all cases the distributions are skewed to contain a fat tail.

An example typical of the metric data in our data set is illustrated in Figure 5.1 and it shows the relative frequency distributions, for the metrics Number of Methods and Fan-Out Count for release 2.5.3 of the Spring framework (a popular Java/J2EE light-weight application container). In both cases the distributions, are significantly skewed. However, the shape of distribution is different. This is a pattern that is recurring and common, that is, though the distributions are non-guassian and positively skewed with fat tails, they are different for different systems and metrics. A complete list of all descriptive statistics and the result from our test for normality is presented in Appendix E.

### 5.4.3 Evolution of Metric Distributions

The upper and lower boundaries of the metric data distribution is bounded within a fairly narrow range. Figure 5.5 presents the boundaries of the histograms based on the minimum and maximum values of Number of Branches, In-Degree Count, Number of Methods and Out-Degree Count attained across all versions of the Spring Framework. The figures show that relative frequency distributions of these measures have a distinct profile that is bounded in a small range. The notable fact is that this remarkable stability is observable over an evolution period of 5 years.
Figure 5.5: Spring evolution profiles showing the upper and lower boundaries on the relative frequency distributions for Number of Branches, In-Degree Count, Number of Methods and Out-Degree Count. All metric values during the entire evolution of 5 years fall within the boundaries shown. The y-axis in all the charts shows the percentage of classes (similar to a histogram).

A similar phenomenon was observed across multiple projects for the metrics under study. The profile of the relative frequency distribution of all the metrics hold their broad shape across the evolutionary history of any given software system. For example, if 20% of the classes in a system have a In-Degree Count of 5 or greater in Version 1, the probability that this value will change by more than a few percent is very low over the evolutionary history of the product. This holds for all of the various values of the other distributions as well.

There are however, some exceptions to this rule that coincide with structural shifts from one major release to another. For instance, in Hibernate, one of the systems in our study, we noticed the profile of many distributions has shifted significantly, twice during its evolutionary history. Upon closer examination we found that the profile shifted to a new bounded range when the team moved from one major ver-
Figure 5.6: The distinct change in shape of the profile for Hibernate framework between the three major releases. Major releases were approximately 2 years apart.

sion to another with a different underlying structure. In the case of the Hibernate framework, different distribution shapes can be seen in Figure 5.6 between three major releases. This observation also corresponds with the release notes that indicate that the development team have undertaken substantial changes to the underlying structure and functionality of the software system. Though, substantial changes are evident, this is not the norm and in most cases the distributions can be considered to be stable.

5.4.4 Bounded Nature of Gini Coefficients

Given the observed bounded range visually, do the Gini Coefficients confirm a similar pattern? Do developers across domains tend to structure software similarly? Are there any bounds that they consistently do not cross? In order to understand the underlying distribution from a statistical perspective, we computed the Gini coefficients for the 10 metrics as outlined in Table 5.1.

The typical range for the Gini coefficient independent of the metric or system under consideration is between 0.47 and 0.75, with a mean value
### Table 5.3: Gini value ranges in Spring Framework across 5 years of evolution

<table>
<thead>
<tr>
<th>Metric</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Instruction Count</td>
<td>0.60</td>
<td>0.64</td>
<td>0.04</td>
</tr>
<tr>
<td>Store Instruction Count</td>
<td>0.63</td>
<td>0.68</td>
<td>0.05</td>
</tr>
<tr>
<td>Number of Branches</td>
<td>0.68</td>
<td>0.72</td>
<td>0.04</td>
</tr>
<tr>
<td>In-Degree Count</td>
<td>0.62</td>
<td>0.72</td>
<td>0.10</td>
</tr>
<tr>
<td>Out-Degree Count</td>
<td>0.45</td>
<td>0.49</td>
<td>0.04</td>
</tr>
<tr>
<td>Number of Methods</td>
<td>0.48</td>
<td>0.56</td>
<td>0.08</td>
</tr>
<tr>
<td>Public Method Count</td>
<td>0.47</td>
<td>0.56</td>
<td>0.09</td>
</tr>
<tr>
<td>Number of Attributes</td>
<td>0.57</td>
<td>0.67</td>
<td>0.10</td>
</tr>
<tr>
<td>Fan-Out Count</td>
<td>0.62</td>
<td>0.66</td>
<td>0.04</td>
</tr>
<tr>
<td>Type Construction Count</td>
<td>0.66</td>
<td>0.80</td>
<td>0.14</td>
</tr>
</tbody>
</table>

At approximately 0.65 (see Figure 5.7 for a box-plot). However, individual metrics across all systems fell within a narrower boundary. The inter-quartile-range (IQR) for Gini coefficients for any individual metric was within a very small range of 0.15.

When any single system is considered, the range (difference of maximum and minimum) for all metrics falls within an even smaller range across the entire evolutionary history. Table 5.3 shows the tight range of Gini values for the Spring Framework over 5 years of evolution. In 22 systems the Gini range across all metrics under investigation was under 0.10. In the remaining systems a couple of strong outliers pushed the range out to approximately 0.20. The metric that has the greatest range, is IDC, while the one that has the smallest range is ODC. Except for IDC all other metrics (across all systems) have a typical range well under 0.10.

Though, the relatively high value for Gini coefficient was not surprising (given that metric distributions are known to be skewed), the narrow range for the IRQ confirms the visual observation that metric distributions do not change substantially over time.

Another observation we made is that it is rare to see Gini coefficients greater than 0.8. For example, we noticed that in Struts, a Web application framework, the Gini coefficient for In-Degree Count initially moves beyond this threshold, only to fall back below it within a few releases (see Figure 5.8). This interesting behaviour suggests that, in
Chapter 5. Growth Dynamics

order for software systems to sustain evolution, responsibilities have to be distributed across abstractions in such a way that developers can maintain their cohesion. For each measure we can clearly identify lower and upper bounds that appear to define corresponding limits (see Table 5.3).

5.4.5 Identifying Change using Gini Coefficient

We discovered in our analysis that Gini coefficients normally change little between adjacent releases. This stability is illustrated visually in Figure 5.9 and shows four Gini Coefficient values for 51 consecutive releases of the Spring framework. Although stability is the norm, changes do happen and may result in significant fluctuations in Gini coefficients that warrant a deeper analysis. But, what is the nature of this change? Do they tend to be relatively stable or are they susceptible to substantial changes?

Figure 5.7: Box plot of Gini coefficients across all selected Java systems.
Our observations show that individual measures of Gini coefficients showed a remarkable stability between consecutive versions. The probability (across our entire data set) that Gini coefficient will change by more than 0.01 is only 10% and there is only a 1% probability that the Gini coefficient will change by more than 0.05.

In the analysis that follows we have chosen a difference of greater than 4% between adjacent releases as being significant. The motivation for this is the Pareto principle [30] (also known as the 80–20 rule). We found that Gini coefficients changed by more than 4% in less than 20% of the studied releases.

To illustrate the effectiveness of this threshold, consider again Figure 5.9 showing selected Gini coefficients from the Spring framework. We see a jump of 0.092 in Type Construction Count from the 6th (i.e., version 1.0m4) to the 7th (i.e., version 1.0r1) release. Upon further inspection we discovered that this change was caused by the removal of just one, yet very rich, class — ObjectArrayUtils, which also affected the Gini value for Number of Methods. This class provided 283 utility methods to map primitive arguments to an object array. These methods contained a total of 1,122 object instantiations. The next biggest class in terms of type constructions defined just 99 object instanti-
This concentration of type constructions in `ObjectArrayUtils` caused the high values for Type Construction Count of approx. 0.8 up to the 6th release. A similarly rich class was never added to the Spring framework again.

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

### 5.4.6 Extreme Gini Coefficient Values

An analysis of the distributions showed that very high Gini values (typically those over 0.85) were consistently observed in only two metrics:
NOB and TCC. We also noticed a few systems with high values of *Number of Methods* and *Number of Attributes*.

The systems where these high values were consistently found were: Checkstyle, Hibernate (version 3.0b1 and higher), PMD, Groovy, ProGuard, FreeMarker, JabRef (version 1.4.0 and higher), and JasperReports. In these systems we observed persistent occurrences of Gini values for *Number of Branches* greater than 0.83, with a value of 0.91 for CheckStyle (version 2.1.0). But, why should this be the case? We discovered, upon further inspection, that all systems contained machine-generated code, which yields an extremely uneven functionality distribution. Seven systems out of the eight used compiler-compilers to generate parsers which centralise the semantic rules into few classes that tend to be large and have a very high Weighted Method Count. The other system that we picked up with large Gini values was ProGuard with a TCC value over 0.9. This was caused by a few mapping classes that have been generated using a code generation template.

Two other systems also produced fairly high Gini values were Xerces2 and Xalan. In both of these systems, the Gini coefficient for NOB is between 0.75 and 0.81 over the entire evolution history. These high values resulted from hand-written parsers that produced functionality distribution profiles closer to systems that contained machine-generated code. These were the only instances in which we observed such high values for *Number of Branches* or *Type Construction Count* without the presence of machine-generated code.

### 5.4.7 Trends in Gini Coefficients

We analyzed the trend in Gini coefficients over the entire evolution history in order to determine if developers tend to centralise functionality and complexity by observing the relationship between the Gini coefficient and the Age.

The correlation coefficient $\rho$ which is an indication of the strength and direction of the relationship between two variables (Gini, Age) does not
show a consistent trend in the Gini Coefficients that can be generalised in 9 out of the 10 metrics, the exception was In-Degree Count. In all metrics except In-Degree Count, the range of \( \rho \) values was dispersed between 0.98 and \(-0.95\). This observation is shown in Figure 5.10 and the range for In-Degree Count is clearly different from the rest of the metrics under consideration.

Interestingly, we also observed in the other 9 metrics, the Gini coefficient values trended up for some metrics, while others moved down in the same system. There were also very strong positive and strong negative trends for certain metrics, though not consistently similar across all systems, for instance in PMD the \( \rho \) value for NOB was \(-0.98\) while in Kolmafia the value for NOB was \(+0.89\). In approximately 30% of the cases, there was no observable trend, with the \( \rho \) value in the range \(-0.35\) to \(+0.35\) which shows that there is at best a weak relationship. Figure 5.10 provides a box plot that indicates the range and IRQ of correlation coefficients for the 10 different metrics under study. As can be seen, the range of values is very broad and in 5 metrics (ODC, PMC, FOC, LIC and NOB) the median value was zero.

The In-Degree Count, a measure of popularity was the only one that consistently showed a positive trend. In 20 systems the \( \rho \) value was greater than 0.8 and in 11 systems the value was in the range 0.55 to 0.8. 3 systems showed a weak relationship (\( \rho \) values of 0.08, 0.22 and 0.33), while only 3 systems (Struts, Tapestry and Ant consistently) showed a strong negative trend (\( \rho < -0.6\)).

### 5.4.8 Summary of Observations

The aim of our study into the nature of growth in software was to understand if evolution causes growth to be evenly distributed among the various classes in the software system, or if a certain classes tend to gain more complexity and size. In order to answer this question, we studied how the metric data distribution changes over time.
Figure 5.10: Box plot of Gini coefficients correlated with Age

The results of our study show that:

1. Software size and complexity metric distributions are non-gaussian and are highly skewed with long tails. Developers seem to favor solutions with few large and complex abstractions.

2. The metric distributions of both size and complexity measures, for any given application change very little between two consecutive versions. Though the distribution is stable, there are occasional spikes indicating significant changes do occur. However, the general pattern is that once developers commit to a specific solution approach the organisational preferences are consistent as the software system evolves.

3. Systems that contain machine generated code (such as those created by compiler-compilers or generated wrapper classes) have a unique profile and are detectable by Gini Coefficient values over 0.85.
4. The popularity of a class is not a monotonic function of its size or complexity (as captured by our 10 metrics). In other words, large and complex classes are no more popular than small and simple classes.

5. The process of software evolution in general causes popular classes to gain additional dependents and hence become even more popular.

5.5 Discussion

In this section, we discuss how developers choose to maintain software systems by interpreting the observations presented in the previous section. Specifically, we argue that though the different metrics selected are strongly correlated, the inconsistency of the strength between different systems warrants the use of different metrics in order to understand the maintenance of a software system effectively. Furthermore, we discuss the finding of the consistent increase in the Gini Coefficient value of the In-Degree Count and argue that this observation provides the empirical support for the preferential attachment model that has previously been postulated as one of the likely models of growth. We also present an explanation for the observation of highly stable and strongly bounded range for the Gini Coefficients and show that the decisions that developers make with respect to how they organise a solution are highly consistent during the development life cycle. Additionally, we present our interpretation for why developers prefer to construct and maintain solutions with highly skewed distributions where much of the functionality is centralised into a few god-like classes. We end the section with a discussion on the value and applicability of the Gini Coefficient for identifying releases with significant changes as well as for detecting the presence of machine generated code.
5.5.1 Correlation between Metrics

The analysis we undertook to identify if a smaller sub-set of metrics will be sufficient to characterise a software has shown that none of our selected measures, except Load Instruction Count and Store Instruction Count, qualify for potential exclusion from the analysis of metric distributions. Further, even if two metrics are correlated at one point, our observations showed that the strength of this relationship changes for different systems and is not necessarily maintained at the same level while a specific system evolves. For example, consider the range for correlation coefficients between Number of Branches and all other measures in Proguard as shown in Figure 5.11. As can be seen, these $\rho$ values are sufficiently broad. As a consequence, all selected measures have to be considered significant as they can reveal different, sometimes surprising, dimensions of the software system, which may be lost by eliminating certain metrics. Furthermore, these correlation values can
change significantly as the software system evolves as can be seen with the Number of Fields (NOF) metric in Proguard (see Figure 5.11).

Moreover, we found repeatedly that the Gini coefficients for metrics that may be considered correlated, diverge independently of the underlying metric data relationship. Though Load Instruction Count and Store Instruction Count are strongly correlated (often well over 0.90), we need to analyze summary measures derived from them separately also. We observed in some instances that the Load Instruction Count Gini Coefficients changed independently of the Store Instruction Count Gini Coefficient values. For example, between the release 3.0 and 3.1 of CheckStyle the Load Instruction Count Gini Coefficient changes from 0.870 to 0.804 while Store Instruction Count Gini Coefficient only changes from 0.831 to 0.827.

Interestingly, our findings contradict the recommendation provided in earlier work by other researchers [40, 41, 110, 120] who observed a strong correlation between various size and complexity metrics and hence argued that a smaller sub-set of the measures is adequate. However, the earlier observations did not consider how the strength of the correlation changes over time. We however find that it is important to consider multiple dimensions since the strength of the correlation cannot be considered to be a general property and it is clearly influenced during the maintenance of a software system. Furthermore, an abnormal change in the correlation between various metrics can be used as a trigger for further investigation.

### 5.5.2 Dynamics of Growth

How is maintenance effort focused by developers? When developers create a new version, they build it on top of existing code by adding new classes, modifying and potentially removing some existing classes. If the tendency of developers is to grow the size and complexity of a system by extending existing classes, then over time this preference causes older classes to gain size and complexity. This preferential modification will be observable as an upward trend in the Gini Coefficient values.
as existing classes gain more size and complexity, that is, they become relatively wealthier.

In our data set, we found a generalizable and consistent trend only in the Gini Coefficient for In-Degree Count. The Gini Coefficients for all other metrics increase in some systems, while in others they decrease. Our finding shows that the change in distribution of class size and structural complexity is not a consistent and predictable evolutionary property. The variability in the Gini coefficients across systems suggests that there are different evolutionary pressures for different systems and developers respond as needed. However, the stability of the distribution profiles indicates that a system generally keeps its character over time — a finding that provides some indirect support for the Law of Conservation of Familiarity (cf. Section 2.3).

When developers add a set of new classes, these fit within the overall probability distribution profile as indicated by the minimal change in the Gini coefficient. Although our observations indicate that the long-term trend for most Gini Coefficient values cannot be reliably predicted, there is a high probability that the distribution of size and complexity between any two consecutive versions are very similar.

### 5.5.3 Preferential Attachment

We analyzed the trend in the Gini values to answer one of our research questions—“do large and complex classes get bigger over time?”. If large and complex classes gain additional volume in terms of code, then they will grow at a slightly faster rate than the rest of the code base causing the Gini Coefficient values to increase as software evolves. Our observations showed a consistent trend in the Gini Coefficient value of IDC, but none of the other metrics had a sufficiently generalizable trend. In some systems there was an increase, while in others there was a decrease in the Gini Coefficient values over time.

Our findings show that popular classes tend to gain additional dependents indicating that, in general, software systems are built incre-
mentally, with new functionality constructed on top of existing code. Though not surprising the real value of our analysis is that it provides empirical support for the Yule process [210] also known as preferential attachment growth model. In the simple form of this growth model, as classes are added, the probability that a class will depend on another existing class is proportional to the popularity of the existing class [55]. Simply put, popular classes will gain additional dependents causing their popularity to increase. The preferential attachment model has been used as one of the explanations for how certain web sites gain popularity on the world-wide web [16,209]. An interesting statistical property of systems that exhibit growth via a preferential attachment is that the data exhibits a highly skewed distribution and tends to fit a power-law distribution [210].

Many researchers that study software have noted that the In-Degree Count metric distributions follow a power-law [21,54,55,223,287,299], and have hypothesised that this distribution arises due to preferential attachment. That is, they have inferred the potential growth mechanism from the observed outcome. However, although preferential attachment can cause a power-law distribution, it is not the only model that can give rise to this category of skewed distributions [48,145]. This possibility implies that the hypothesis of preferential attachment model generating skewed In-Degree Count distributions was not fully validated empirically. Our observations show that, in general, there is an upward trend in the Gini coefficient for IDC providing empirical support for the preferential attachment growth model as the most likely explanation for the observed highly skewed distributions for In-Degree Count metric.

In our data, we observed three systems – Struts, Tapestry, and Ant where the Gini Coefficients for In-Degree Count decreased over time. Why are these systems different? In order to answer this specific question, we had to study the release notes as well as all available architecture and design documentation for these systems.

The developers of Struts and Tapestry performed major restructuring of the code base and in both cases, abandoned a substantial amount
of the code base during these restructuring phases (more detail is presented in the next chapter). The major rework has caused the Gini coefficient values for In-Degree Count to reduce substantially. In essence, in both of these systems the popular classes were not able to establish themselves to gain additional popularity.

The Ant Build system was unique in its behaviour as the Gini Coefficient value for In-Degree Count decreased gradually over time. Furthermore, there was no evidence of any substantial rework in this system. The reason that the popular classes slowly lost popularity was driven by the inherent architecture of Ant, specifically how new features were added. In Ant, new features were typically added into the core functionality via a plug-in architectural style. The plug-in contained a set of new classes that provided the needed functionality. Although the plug-ins rely on some of the existing core classes, the dependency binding into the core classes was achieved via external configuration files rather than statically in Java code. However, since our metric extractor is unable to process external configuration files to determine dependency information, we were not able to fully identify these dependencies. When the dependency data was checked manually for four consecutive versions (1.6.4, 1.6.5, 1.7.0 and 1.7.1), we observed a weak positive correlation trend.

Unlike In-Degree Count, in the other metrics there was no strong generalizable trend in the Gini Coefficient suggesting that growth models other than preferential attachment may be at work.

5.5.4 Stability in Software Evolution

We observed in our analysis that Gini coefficients normally change little between adjacent releases. However, changes do happen and may result in significant fluctuations in Gini coefficients that warrant a deeper analysis (see Figure 5.9 showing selected Gini profiles for all releases of the Spring framework). But why do we see such a remarkable stability of Gini coefficients?
Developers accumulate system competence over time. Proven techniques to solve a given problem prevail, where untested or weak practices have little chance of survival. If a team has historically built software in a certain way, then it will continue to prefer a certain approach over others. This typical human behavioural trait was originally observed by Milgram [197] who showed that given multiple choices, most people will avoid selecting the choice which would invalidate their previous behaviour. Furthermore, this regularity in behaviour is consistent even at the level of a group [276] (Chapter 14). Although the problem context and personalities within the group have been shown to influence how a group behaves, in general, group behaviour is consistent over time [276]. Moreover, in the context of software development, we can expect that most problems in a given domain are similar, hence the means taken to tackle them would be similar, too. Tversky and Kahneman coined the term decision frame [285] to refer to this principle in which decision-makers proactively organize their solutions within well-established and strong boundaries defined by cultural environment and personal preferences. Our study suggests that these boundaries manifest themselves also in the software systems.

When developers are making decisions, they weigh the benefits of using a large number of simple abstractions against the risk of using only a few, but complex ones in their solution design. Our findings indicate that developers favor the latter. In particular, we learned (see Figure 5.7) that the Gini coefficients of most metrics across all investigated systems assume bounded values. These values mark a significant inequality between the richest and the poorest. For example, in Spring framework version 2.5.3 approx. 10% of the classes possess 80% of the *Fan-Out Count* wealth (see Figure 5.3).

Another observation we made is that it is rare to see Gini coefficients greater than 0.8. For example, we noticed that in Struts, a Web application framework, the Gini coefficient for *In-Degree Count* initially moves beyond this threshold, only to fall back below it within a few releases (see Figure 5.8). This interesting behaviour reveals that, in order for software systems to sustain evolution, responsibilities have to be distributed across abstractions in such a way that developers can main-
<table>
<thead>
<tr>
<th>System</th>
<th>Measure</th>
<th>Ver.</th>
<th>Gini</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMD</td>
<td>TCC</td>
<td>1.01</td>
<td>0.81</td>
<td>User interface code refactored into multiple smaller classes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.02</td>
<td>0.73</td>
<td></td>
</tr>
<tr>
<td>Checkstyle</td>
<td>IDC</td>
<td>2.4</td>
<td>0.44</td>
<td>Plug-in based architecture introduced.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.0b1</td>
<td>0.80</td>
<td></td>
</tr>
<tr>
<td>Proguard</td>
<td>TCC</td>
<td>3.8</td>
<td>0.78</td>
<td>2,500 line obfuscation instruction mapping class introduced.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.01</td>
<td>0.90</td>
<td></td>
</tr>
<tr>
<td>JabRef</td>
<td>NOB</td>
<td>1.3</td>
<td>0.75</td>
<td>Machine generated parser introduced.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.4</td>
<td>0.91</td>
<td></td>
</tr>
<tr>
<td>WebWork</td>
<td>MIC</td>
<td>2.1.7</td>
<td>0.51</td>
<td>A large utility class and multiple instances of copy-and-paste code introduced.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.21</td>
<td>0.62</td>
<td></td>
</tr>
<tr>
<td>Xerces2</td>
<td>IDC</td>
<td>2.0a</td>
<td>0.59</td>
<td>Abstract syntax tree node referencing changed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.0b</td>
<td>0.78</td>
<td></td>
</tr>
<tr>
<td>JasperReports</td>
<td>PMC</td>
<td>1.0.3</td>
<td>0.58</td>
<td>Significant design approach change with introduction of a set of new base classes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.1.0</td>
<td>0.69</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4: Sample of observed significant changes to Gini coefficients in consecutive releases.

tain their cohesion, further supported by the lower and upper bounds that appear to define corresponding limits (cp. Table 5.3).

The existence of both lower and upper bounds for Gini Coefficients in software systems can be viewed as a result of a trade-off developers use throughout development. Decisions on how to allocate responsibilities to classes within a system are a product of the application domain, past experience, preferences, and cultural pressures within the development team. The Gini Coefficient is a potential mechanism for summarising the outcome of these choices.

5.5.5 Significant Changes

The stability of the Gini coefficient indicates that developers rarely make modifications to systems that result in a significant reallocation of functionality within the system. Managers can, therefore, use this knowledge to define project-specific triggers both to help detect substantive shifts in the code base and to ask directed questions about the reasons for their occurrences. For example, in our study we were able to detect a major change in the Type Construction Count of Proguard, a Java byte code obfuscator, between release 3.8 and 4.0 (cf. Table 5.4). A detailed analysis disclosed that the developers had added a set of large
auxiliary classes to centralise the obfuscation mapping of instructions — a change, so significant, as to warrant an appropriate notification to both internal and external team members. A sample of observed significant changes in Gini Coefficients is presented in Table 5.4.

Architects often face a situation in which they have to select a third-party component. They need to consider the desired functional requirements as well as project-specific and institutional principles governing local development practices and styles.

In other words, we need to consider risks arising from mismatches between existing team habits and newly adapted third-party preferences. Therefore, before a final decision is made, architects should inspect past or ongoing projects and compare the responsibility distribution profiles (captured in the respective Gini coefficients). This method focuses on fitness rather than prescriptive rules to proactively avert emerging development risks.
The true benefit of the Gini coefficient is its ability to capture precisely and summarize changes in both the degree of concentration and the population size. When analyzing metrics data, crucial aspects to consider are the width of distribution and its underlying dispersion [183]. Standard descriptive measures like median are blind for this dimension. A system that vividly demonstrates the problem with standard descriptive statistics is ProGuard (see Figure 5.12).

The median for Type Construction Count, for example, stays firm at 1 for 340 weeks of development, suggesting a fairly routine evolution of the code base over a period of 6.5 years. The Gini coefficient for Type Construction Count, on the other hand, moves from 0.776 to 0.897 indicating that the arrival of new classes results in a less equitable concentration of object instantiations. In case of ProGuard, the changes to the system occur at the upper end of the Type Construction Count measure. While the median remains for Type Construction Count the same, the changing Gini coefficient reflects correctly the occurrence of regular fluctuations as well as a sudden architectural change at RSN 19 (corresponding to Version id. 3.8, see Figure 5.12). Between RSN 19 and 20, the developers of this software system modified the strategy used for mapping the actual code to obfuscated code which can be considered as a substantial architectural change since it adjusts how this software system implements its core functionality.

5.5.6 God Classes

An interesting facet observed in all studied systems is the that a few God-like classes shoulder most of the work. This observation is supported by the typical size and complexity metric distribution pattern where the majority of the classes are very small with minimal structural complexity. Our findings suggest that developers are not afraid to construct, maintain, and evolve very complex abstractions. Contrary to common belief developers can actually manage complexity in real projects, but at what price? Classical software engineering shies away from tightly arranged and centralised abstractions. However, we find that it is well within the developers’s mental capacity to organize and
control the structure of software. Furthermore, our analysis suggests that there has to exist a certain degree of inequality, which defines the *equilibrium* within which software evolution occurs.

The empirically observed nature of software systems is at odds with the recommendations of how software should be structured. Specifically, some of the guidelines prescribed by Object-Oriented Design Heuristics [237] as well as rules that are considered to encourage good practice [136, 161, 178, 179, 189].

These design guidelines suggest that developers should construct classes with clearly defined responsibilities [303] which are neither too complex nor too big [237]. Though some attempts have been made to quantify high complexity and large classes [61, 244] the interpretation of the heuristics has been left as a contextual and hence subjective choice of the developers.

The design guidelines are fairly broad. However, if the recommendations are followed, the metric distributions should be less skewed than observed since complexity and size would be much more equitably distributed. Although our observations do not imply that developers actively ignore these recommendations, nor that the recommendations themselves are incorrect, we still need to reconcile how developers are constructing software with how they *ought to* construct software. The likely explanation for the centralization of complexity and key functionality into few abstractions is that developers tradeoff volumetric complexity with structural complexity. That is, developers prefer to deal with a few complex classes rather than a large number of simpler classes. This explanation is supported by research into human memory in the field of cognitive psychology [12, 13]. These studies into memory and recall have shown that in general humans working memory works well when the task requires only working with few (less than 10) chunks of information [75].

In the context of object-oriented software development, the chunks of information are classes and developers will, in general, reduce the number of classes that they have to hold in their working memory.
Specifically, this is likely achieved by ensuring that for a given feature, developers will implement the functionality by ensuring that the set of abstractions that they need to focus on is well within the limits of their working memory. If this decision making method is applied for each feature, over time, the asymmetric distribution observed in various metrics is reinforced.

### 5.5.7 Value of Gini Coefficient

The Gini Coefficient is a single bounded value, and hence offers an easy to use trigger for an investigation that would reveal potential causal mechanisms. One alternatives to using the Gini Coefficient will be to identify and observe the movement in outliers using a range of outlier detection techniques [24]. Another alternative is to use a combination of arithmetic mean, median, skewness and kurtosis to qualitatively deduce the nature of the distribution. Though these techniques may offer some insight, they do not digest the information taking into consideration the entire population size and also are limited since there is no easy baseline to compare against. Furthermore, the outlier detection method will highlight a set of classes rather than present information that permits direct comparisons between versions without additional analysis.

Gini Coefficient based analysis offers information on what can be considered a typical range and provides a guideline for investigation, including for detecting machine generated code and unusual patterns via a single numerical value. The information provided by Gini Coefficients is helpful for solution architects and developers, especially when they need to take over a software system and maintain it. For example, if a system has machine generated code, additional knowledge and skills might be needed in order to maintain and enhance blocks of code that are automatically generated.
Software Code Audit

Knowing that Gini coefficients have strong boundaries can be used to improve software comprehension. We measured the Gini Coefficient during two commercial code audits undertaken as part of a broader consulting exercise specifically to help understand the nature of code developed by a third party. This auditing exercise was not part of the initial scope of this research project. Both software systems were developed in Java and had an evolution history of a minimum of 4 versions.

When we audited the systems, we used two rules to help us trigger a broader investigation. The first rule was to look if over the evolution history the Gini coefficient for any of the metric was outside the upper boundary of the Inter-Quartile Range for that metric (see Figure 5.7) as this suggests a distribution that is atypical. The second rule that we applied was to investigate releases where any of the 10 metrics had a Gini coefficient value over 0.85. This value occurs only 2.5% of the time in our data (if we ignore metric, release and system). Essentially this is a rare occurrence and flags potential machine generated code (discussed in Section 5.5.8).

In the first system (tax calculation software for large corporations with 5 releases) that we audited applying the above rules, we noticed that the Gini coefficient for NOB, NOA and TCC was consistently over 0.95 (in all 5 versions available for investigation). Additional investigation showed that this system contained a large number of stub classes that were generated from code templates. Interestingly, the stub class generator was a script developed specifically for this purpose and even though these stub classes were used extensively their generation technique was not specifically documented.

The second system that we investigated (phishing site detection engine with 8 releases) had Gini coefficient values within the IQR. There were however, two substantial jumps identified as a relative change over 4% between consecutive releases in the In-Degree Count and Out-Degree Count Gini Coefficient values. These two releases involved major revisions, specifically a reorganisation of the architecture to improve
maintainability and changes in many classes to improve performance. Unlike the first system, in the second system, the changes were documented in the release notes.

Using the Gini Coefficient in both cases identified key aspects of the software systems quickly without a deep investigation of the source code and source repository logs. Our personal experience from the two audits was that the quick identification of the use of templates to generate code in the first system helped us identify a key design element of the solution at the beginning of our audit exercise allowing for more effective use of our time since we eliminated the need to investigate approximately 30% of the classes scattered across different packages.

5.5.8 Machine-generated Code

In our study we noticed that certain systems consistently exhibited Number of Branches Gini values above 0.85. An investigation into those unusually high values revealed the presence of machine-generated code, specifically parsers and expression processors. These parsers typically contain a few large classes that encapsulate very long methods with abnormally high algorithmic density caused by many thousands of branches that use of hundreds of variables. This is caused because code generators often employ greedy strategies to map a given specification to target code.

Systems that contain machine-generated code structures tend to have a unique distribution profile, where Gini Coefficients tend to be close to 1 (see Figure 5.13 showing the Number of Branches Gini Coefficient values for JabRef). JabRef is a tool that helps manage bibliography databases, specifically BibTex files. Interestingly, the developers initially used a hand-written parsers and in RSN 6 introduced a machine generated parser. The introduction of the machine generated parser for BibTex files caused the Number of Branches Gini Coefficient value to increase from 0.74 to approximately 0.91 (see Figure 5.13).
High Gini Coefficient values indicate that these systems have a small set of very large and complex classes (wealthy in terms of size and complexity). As noted earlier, human developers rarely write code in which Gini Coefficients for specific measures go past 0.80 as they will find it hard to write and maintain methods with very high algorithmic complexity [74, 263]. The ability of detecting and flagging machine generated code is valuable since it signals the possible need for additional expertise in order to maintain or enhance an existing code base and to meet strategic objectives.

Interestingly, we did notice a higher Number of Branches value (Gini Coefficient around 0.8) in two XML parsers Xerces and Xalan, both of which contained human written parsers. We determined that they were handwritten by studying repository access logs and comments left within the code. When machines generate parsers, they create a full set of classes rather than partially adjusting a few lines to fix defects, this is visible in the repository logs related to those files. Further, the code files written by machine use variable names that are alpha-numeric with the numeric component increasing in value for each new variable rather than in a human readable form (for example the variables generated by the machine generated parser take the form jj_123, jj_124, jj_125 etc.)
Surprisingly, in software with human written parsers, the developers choose to centralise the functionality of the parser into a few classes with large and complex methods rather than distributing the complexity among a large number of classes. Though these classes did not match the size or complexity of machine generated code, they influenced the overall distribution sufficiently to cause a higher Gini value.

Another dimension of machine generated code was observed when we investigated two systems PMD and Checkstyle that had very high Gini Coefficient values for Number of Attributes and Number of Methods. Interestingly these systems had a substantial number of classes with either zero fields or zero methods. This zero inflation increases the Gini Coefficient value. But, what type of design choices cause such an unusual shape? Interestingly, both PMD and Checkstyle serve a similar purpose. These tools help developers check that Java source code adheres to specified coding conventions and standards. This functionality is provided via a set of configurable rules around coding standards, formatting rules and source code metrics.

A deeper investigation showed that both Checkstyle and PMD represent the Java source code that they process by constructing an Abstract Syntax Tree (AST) and use the visitor pattern to walk the nodes in the tree. In both these systems, the rules to check source code were developed by extending an abstract rule class with the child class providing the rule implementation. The rules that developers could configure and add in these tools were written without making use of any fields and often encapsulated within a single method to satisfy the visitor pattern. This design approach created a large number of “poor” classes making those that actually possessed more functionality look unusually rich, pushing the Gini coefficient value closer to 1.

5.6 Summary

Evolving software tends to exhibit growth. In this chapter our focus was on how maintenance effort is distributed by analyzing class size and complexity metric distributions.
Chapter 5. Growth Dynamics

The data we collected for our study showed that size and complexity are unevenly distributed, confirming similar observations made by other researchers. However, these skewed distributions cause standard statistical summary measures such as mean and standard deviation to provide misleading information making their application for comparative analysis challenging. In order to overcome this limitation, we analyzed the data using Gini coefficient, a higher-order statistic widely used in economics to study the distribution of wealth.

The results from our study showed that metric distributions have a similar shape across a range of different systems. These distributions are stable over long periods of time with occasional and abrupt spikes indicating significant changes are rare. A general pattern is that once developers commit to a specific solution approach, the organisational preferences are stable as the software system evolves.

An unexpected finding of this study was the observation that systems containing machine generated code have distributions that are significantly more skewed. Based on this unique profile we developed a technique that can be used to automatically detect the presence of generated code — a technique that can greatly aid software comprehension as well as to assist in software reviews.

Our analysis shows that the popularity of a class is not a function of its size or complexity, and that evolution typically drives these popular classes to gain additional users over time. This finding provides empirical support for the Yule process, also known as the preferential attachment growth model. Interestingly, measures of size and structural complexity do not show generalizable pattern. That is, large and complex classes do not get bigger and more complex over time purely due to the process of evolution, rather there are other contributing factors that determine which classes gain complexity and volume. Identification of these factors is an area that we leave as future work.

Given that popular classes in general gain additional popularity in evolving software, a natural question that arises is — “does this growth in popularity place any pressure on popular classes to change?” Lehman’
Chapter 5. Growth Dynamics

Laws of Software Evolution [174] suggest that software systems, due to their very use, provide evolutionary pressures that drive change. Would classes due to their popularity also undergo modifications? Although, this was not an initial research question that we set out to answer in our study, we added it into our study of the nature of change and discuss our findings in the next chapter.
Chapter 6

Change Dynamics

It is a widely accepted fact that evolving software systems change and grow [166]. However, it is less well-understood how change is distributed over time, specifically in object oriented software systems. The patterns and techniques presented in the previous chapter permit developers to identify specific releases where significant change took place as well as to inform them of the longer term trend in the distribution profile. This knowledge assists developers in recording systemic and substantial changes to a release, as well as to provide useful information as input into a potential release retrospective. However, the analysis methods presented in the previous chapters can only be applied after a mature release of the code has been developed. But in order to manage the evolution of complex software systems effectively, it is important to identify change-prone classes as early as possible. Specifically, developers need to know where they can expect change, the likelihood of a change, and the magnitude of these modifications in order to take proactive steps and mitigate any potential risks arising from these changes.

Previous research into change-prone classes has identified some common aspects, with different studies suggesting that complex and large classes tend to undergo more changes [20,28,31,63,130,177,213,266,314,318], and classes that changed recently are likely to undergo modifications in the near future [93,94]. Though the guidance provided is
helpful, developers need more specific guidance in order for it to be applicable in practice. Furthermore, the information needs to be available at a level that can help in developing tools that highlight and monitor evolution prone parts of a system as well as support effort estimation activities.

The specific research questions that we address in this chapter are:

- What is the likelihood that a class will change from a given version to the next?
  - Does this probability change over time?
  - Is this likelihood project specific, or general?
- How is modification frequency distributed for classes that change?
- What is the distribution of the magnitude of change? Are most modifications minor adjustments, or substantive modifications?
- Does structural complexity make a class susceptible to change?
- Does popularity make a class more change-prone?

In this chapter, we address the above questions and present a set of recommendations that can help developers to proactively monitor and manage change. These recommendations are derived from a statistical analysis of change in approximately 55000 unique classes across all projects under investigation. The analysis methods that we applied took into consideration the highly skewed nature of the metric data distributions, something not always properly considered in previous work [18, 19, 27, 28, 38, 93, 150, 177, 241, 255, 272].

The raw data used in this study is available as data files on the DVD attached to this thesis. Appendix F describes the various data and statistical analysis log files related to this chapter.
6.1 Detecting and Measuring Change

Change is defined\footnote{Oxford American Dictionary, 2005} as a process of transformation — to make the form different from what it is. The first step in understanding change is to identify the unit of change, specifically, the entity that is undergoing change. This is important since change can be observed and understood at different levels of abstraction (for example, package/module, class or method in an object-oriented system). Since classes are the abstraction under study in our work, we focused our investigation at this level.

6.1.1 Approaches for Detecting Change

Detecting if an existing block of code has changed or can be considered identical to another block of code (i.e., a clone) has been an area of study for many years \cite{242,282}. The typical methods applied for detecting changes are: (a) String matching \cite{72,135,143,243}, (b) Abstract-Syntax tree matching \cite{22}, and (c) Metric-based fingerprint matching \cite{5,155}.

String matching involves splitting a block of code (typically a file) into lines, sub-strings or tokens and then comparing if these sub-blocks occur in another file or block of code. The strength of this approach is its simplicity and variations of this approach have been widely used by source code version control tools such as CVS \cite{225} and Subversion \cite{53} to identify if a file (under configuration management) has been modified. However, this approach has some limitations. String matching does not analyse change in context of the underlying programming language semantics which causes it to detect and flag changes even when the program is semantically identical. For instance, if a method was moved from the bottom of a class to the top of a class, it would be flagged as a changed class. Similarly, minor reformatting to improve readability will also be identified as a change.

The Abstract-Syntax tree (AST) based matching approach relies on the construction of a syntax tree by parsing a program file and then match-
ing it with another syntax tree. In contrast to the String matching approach, the Abstract Syntax Tree (AST) based matching is programming language aware and is able to identify the types of changes more precisely. Though this approach is comparatively more complex, it has increasingly been adopted by contemporary IDE’s like Eclipse [73] to perform an intelligent diff allowing developers to compare two versions of a class. The AST based matching approach has also been enhanced to better detect structural changes in object oriented programs by combining it with call graphs analysis [157], and program behaviour analysis [7].

The metric based finger printing approach involves computing a set of measures from the abstraction that is being monitored and comparing this set with a set collected after the change has been made. If there is a difference between these two sets, the abstraction is considered to have been changed. This approach, depending on the implementation can be aware of the underlying programming language semantics, especially if an AST is used to represent the program under analysis. A key advantage of the metric based approach is its ability in providing a consistent qualitative distance measure which can be used to determine the magnitude of a change. One such distance measure is the n-dimensional Euclidian distance measure where the two sets of measures are treated as the input vectors from which a distance measure is computed [34,67,155]. The resulting distance falls on the ordinal scale and only provides sufficient information to permit ordering of changes by magnitude. That is, we can determine that a specific class has had a greater level of change than another. However, these measures are unbounded and hence need to be normalised if they are to be used in a comparative analysis (discussed in additional detail in the next section).

Change detection algorithms have applications beyond an intelligent diff of the source code [242]. Variations of these algorithms have been applied to detect different types of refactoring [288], as well as to identify duplicate blocks of code (clones) in a software system [156,195] as these have been shown to reduce maintainability [236]. Clones are in general caused by developers copying and pasting blocks of code, which
is considered a poor programming practice [236]. These clones often make error correction difficult (since an error may be fixed in the one block of code, yet remain in all of the clones), and also inflate the size of the code making maintenance harder [236].

Although methods to detect changes have been developed, a comprehensive study applying these techniques to understand change and stability from a statistical perspective has not previously been undertaken.

### 6.1.2 Our Approach for Detecting Change in a Class

In our study we detect changes using a combination of metrics-based fingerprint matching and string matching. Our use of metrics to detect identical classes was inspired by work done by Kontogiannis [154, 155] who studied programming patterns using metrics to identify potentially similar blocks of code.

In order to determine change, we start by identifying the class in both versions under comparison. We establish the identity of a class by using the fully qualified class name (that is, the class name including the package name) and checking if that name exists in both versions under comparison. Once identity is established, to determine if a class has undergone modification, we check if any of the following information has changed:

1. The class modifiers (\emph{i.e.}, public, private, protected, interface, final or abstract)
2. The fully qualified name of its direct super class
3. The set of interfaces that this particular class implements (we use fully qualified names)
4. The name, type and modifiers of all fields
5. The name, signature and modifiers of the methods (the return type as well as parameter names and types are considered as part of the method signature)
6. The set of classes that this class depends on (defined as $l_{out}(n)$ in Chapter 4.

7. The set of different absolute count measures computed for the class (see Table 4.4, Table 4.2, and Table 4.3).

If all of the above information is unchanged for a class in two versions under comparison, then that class is considered an identical clone and tagged as *unchanged*. Although it is possible that some distinctively smaller subset of the measures (as well as meta data) will suffice to assess whether a class has changed without significant loss of precision, we adopt the more conservative approach for our analysis and use the full set.

### 6.1.3 Identifying Modified, New and Deleted Classes

The set of classes in a given version is $N_v$, and any given version $v$ of a software system has the following disjoint sets:

- The set of unchanged classes $U_v$ (compared to the previous release),
- The set of classes that have changed $C_v$ (compared to previous release), and
- The set of classes that are newly added in this release $A_v$.

Additionally, there also exists the following disjoint sets in a given version $v$:

- The set of classes that are deleted in the next release $D_v^f$,
- The set of unchanged classes $U_v^f$ in the next release, and
- The set of classes that are modified in the next release $C_v^f$.  

144
The set of changed classes $C_v$ and the set of unchanged classes $U_v$ are determined by comparing the classes in two releases using the change detection technique presented in Section 6.1.2.

The set of deleted classes $D_v^f$ in the next release is determined by comparing any given version with the future version (if available) and checking if the fully qualified class name exists in both versions. The set of newly added classes $A_v$ is computed by comparing any given version with the previous version (if available) and checking if the fully qualified class name exists in that version.

### 6.1.4 Measuring Change

Once the set of changed classes is established, three different dimensions of change can be computed: *amplitude* (size of change), *periodicity* (frequency of change), and *dispersion* (consistency of change). These dimensions were proposed by Barry *et al.* [17] based on a study into volatility in software systems where a phase sequence analysis of maintenance log data (*revision history*) was performed. We based our study on similar dimensions, however we use *release history* (see Section 3.1) as our input and hence can not directly apply the method proposed by Barry *et al.* [17].

In this study, we use the following definitions for the dimensions of change — the *size of change* between two classes is defined as the number of metrics that differ (i.e., if five different metrics differ in value between two classes, then the size of change is 5). The *modification count* is the number of times a class has been modified since its creation — this measure is used as the basis for determining the periodicity. Finally, in order to assess dispersion of change, we measure the proportion of classes in the final version that remain unchanged since creation as well as the proportion that are modified after being created.

We measure the size, consistency and frequency of change for each class in our release history (that is, every class in every version). The rationale for our choices is presented in the next section within the context of the observations.
6.1.5 Measuring Popularity and Complexity of a Class

Once change has been detected and the magnitude assessed, we consider two attributes in order to understand the factors that impact on the probability of change: the popularity of a class, and the complexity of a class. A popular class is one that is used by a large number of other classes. In any development environment, it seems wise to make new classes depend on stable, reliable parts of the system, rather than on those that are constantly changing. As a consequence, it seems logical that new code should depend on existing, proven parts of the system. If this is the case, popularity should make a class stable. The other aspect that is likely to have a close relationship with change is complexity of a class. A complex part, is more likely to have defects [20, 130, 266], and hence changed as part of a corrective modification. However, the counter argument is that a complex part should resist change because developers will avoid these classes to minimize the chance of introducing new defects.

In our study, we define a class to be popular if it has an In-Degree of 5 or more. The value 5 was selected since, on average, only 20% of the classes (applying the Pareto principle) in a system have this level of In-Degree. We use Number of Branches as an indicator of the complexity of a class.

The set of popular classes is $N^p_v$, where $N^p_v \subseteq N_v$. Any given version $v$ of the system has the following disjoint sets of popular classes:

- The set of unchanged popular classes $U^p_v$,
- The set of popular classes that have changed $C^p_v$, and
- The set of popular classes that are newly added in this version $A^p_v$.

6.2 Observations

In this section we present the analysis method as well as the observations within the context of the key research questions. We start with an
analysis of the probability of change in a version after which we investigate the frequency and magnitude of the change. We end this section by presenting our observations on modification probability in popular as well as complex classes.

### 6.2.1 Probability of Change

What is the likelihood that a class will change from a given version to the next? Does this probability change over time? Is it project-specific?

As a system evolves incrementally, software entities are added, removed, adapted and otherwise modified. To assess the likelihood of a class changing, we gather the following statistics by comparing any given version of the software system with the next version in the release history (if a version is available):

- \( u_v^f \): ratio of classes that are unchanged
- \( c_v^f \): ratio of classes that are changed
- \( d_v^f \): ratio of classes that are removed

These ratios are determined as follows:

\[
 u_v^f = \frac{|U_v^f|}{|N_v|} \quad (6.2.1)
\]

\[
 c_v^f = \frac{|C_v^f|}{|N_v|} \quad (6.2.2)
\]

\[
 d_v^f = \frac{|D_v^f|}{|N_v|} \quad (6.2.3)
\]

and,

\[
 u_v^f + c_v^f + d_v^f = 1 \quad (6.2.4)
\]
We also gather the following statistics by comparing a version with the previous version in the release history (if a previous version if available):

\[ u_v \] ratio of classes that are unchanged
\[ c_v \] ratio of classes that are changed
\[ a_v \] ratio of classes that are added

The percentages \( u_v \), \( c_v \), and \( a_v \) are determined as follows:

\[
u_v = \frac{|U_v|}{|N_v|} \tag{6.2.5}
\]

\[
c_v = \frac{|C_v|}{|N_v|} \tag{6.2.6}
\]

\[
a_v = \frac{|A_v|}{|N_v|} \tag{6.2.7}
\]

and,

\[
u_v + c_v + a_v = 1 \tag{6.2.8}
\]

In our input data set, we determined that for any given version \( v \), the following property (Equation 6.2.9) holds in 80% of the versions:

\[
u_v^f > c_v^f > a_v^f \tag{6.2.9}
\]

and the following property (Equation 6.2.10) holds for 75% of versions:

\[
u_v > c_v > a_v \tag{6.2.10}
\]

When we look ahead one version, on average across all systems that we studied, we observe that 75% of the classes are unchanged, 20% are modified and 5% are removed. When we look back one version to detect new classes, on average we note that 70% of the classes are unchanged, 22% are modified and around 8% are new classes. Figure 6.2 highlights
Figure 6.1: Change evolution in the Hibernate framework. This graph illustrates change property captured by Equation 6.2.9.

An interesting observation was that in 12 (30%) systems, at least once in their lifetime, we noticed the percentage of unchanged classes dropping to zero, indicating that every class in the system was modified between two releases. Upon closer investigation, we noticed that this was caused by the developers modifying their package names. In Java, the standard coding convention is to embed the organisation name within the package name (for example, org.apache.wicket.Page indicates that this is a project managed by Apache). When projects move between organisations (or) get adopted by commercial sponsors they tend to rename all classes which causes the unchanged class count to drop to zero. Further, we also had a few instances (25 versions), where developers renamed some of the packages, but left many with the older names. When they partially renamed, we noticed that the number of unchanged classes was mostly under 10%.
Figure 6.2: Change evolution in the Hibernate framework. This graph illustrates change property captured by Equation 6.2.10.

Impact of System Maturity

A graphical analysis of our observations (Figure 6.3) suggested that as software systems mature, the probability that the properties (Equations 6.2.9 and 6.2.10) hold increases. But, is this observation statistically significant? In order to verify the strength of our observation we used a non-parametric one-way analysis of variance using age as a proxy of system maturity. The null hypothesis was that system maturity will have no impact and both properties (Equations 6.2.9 and 6.2.10) are just as likely to break in the older systems as it is in the younger systems.

This hypothesis was tested using the Kruskal-Wallis test [279] that checks equality of population medians among groups. The population was divided into two groups based on if the property under investigation holds or not. If age does not have a significant impact then the population median should be similar for both groups.
Figure 6.3: Box plot of system maturity showing distribution of age (in days since birth) and if the change properties hold. Graph on the left covers Equation 6.2.9, while the right side graph covers Equation 6.2.10.

Kruskal-Wallis equality-of-populations test for the first property (Equation 6.2.9) resulted in a $\chi^2$ value of 29.37 (with 1 degree of freedom), and for the second property (Equation 6.2.10) resulted in a $\chi^2$ value of 24.21 (with 1 degree of freedom). In both cases, the $\chi^2$ value is well above the critical level of 3.84 (1 degrees of freedom) and hence the null hypothesis was rejected at a significance level of 0.05.

The Kruskal-Wallis test shows that system maturity has an impact, but it does not answer a related question – Does the probability that the stated properties hold increase with age? That is, as software matures does the probability of stability increase from the 80% indicated for the first property? In order to answer this specific question, we constructed a Logistic model [279] to predict the probability that the change properties (Equations 6.2.9 and 6.2.10) will hold.
Chapter 6. Change Dynamics

Figure 6.4: Probability of change reduces with system maturity. Graph on the left indicates the probability that Equation 6.2.9 holds, while the right side graph indicates probability for Equation 6.2.10. The probabilities were predicted from the Logistic regression models. Age indicates days since birth.

The logistic model was used here rather than the OLS regression model since it is specifically designed to handle response variable with dichotomous (that is, can be one of two possibilities) outcomes. In our model we used Age as the explanatory (independent) variable while the response (dependent) variable was if the change property holds (0 was used to represent where the property was violated, and 1 was used to represent where the property holds). The parameters of the model were estimated using the Maximum Likelihood Estimate (MLE) method and the probability that the property holds was calculated from the logistic models for each value of age.

The models that we constructed for both properties showed that Age was a statistically significant predictor of how strongly the change prop-
properties hold. For every one unit increase in Age (that is one day), the odds that the change property holds increases by a factor of 1.0008 for Change Property 6.2.9, and by a factor of 1.006 for Change Property 6.2.10. The predicted properties are presented graphically in Figure 6.4 and show that as software systems mature, the proportion of code that is modified reduces.

The key finding is that as software evolves, though classes resist change, there is a certain proportion that are still modified. Furthermore, our observations suggest that in general, the number of modified classes is greater than the number of new or deleted classes. The main application of this observation is in effort estimation, since if $x$ new classes are anticipated to be added in the next version our change properties show that at least $x$ classes will be changed. Though, our findings do not imply that the new classes cause the change, the consistency of this observation in our data set suggests that it can be applied as a heuristic during the planning stage.

The probabilities that the change properties will hold were derived by using the entire data set of releases, ignoring individual projects. However, when we apply the break down by project, we noticed that in 7 systems these properties (Equations 6.2.9 and 6.2.10) are violated significantly more than expected. For instance, Ant Build System violates the second property (Equation 6.2.10) approximately 45% of the time, well over the 25% probability that was identified for the second property. Interestingly, apart from Ant, in all other systems the probability that both properties hold is at least 60%. Though there was no common and consistent driver across all of these systems, the higher than normal change pattern can be explained by certain architectural choices (Ant has a plug-in architecture hence new classes can be added with minimal changes to existing code) and decisions made regarding development method (details are presented in section 6.3).
6.2.2 Rate of Modification

As noted in the previous section, a relatively small set of classes in any given version is subject to change. However, how often are these classes modified? What does the distribution of this modification look like? In order to understand this better, we investigated the number of times a class is modified. We ignore the total number of releases for a project intentionally as it will reveal if there is a common pattern between systems at an absolute level. For example, we can answer the question - what is the probability that developers will modify a class more than 3 times? In essence, we are making the assumption that after a certain point (say the first few releases) the maturity of a software system will have a minimal impact on the number of times a class is modified.

For the analysis, we measured the number of times a class has been modified since birth across the entire data set (approximately 55300 classes across all projects). In our data, on average 45% of the classes were never modified once created. However, when the data was analyzed by project, the range of classes that were never modified was between 22% and 63%. For instance, in Saxon only 22% of the classes were unchanged, while at the other end in Jung and ActiveMQ 63% of the classes were unchanged after birth. Only considering classes that have been modified at some point in their life cycle and counting the number of times that they have been changed, we can observe that, approximately 60% of the classes are modified less than 4 times (see Figure 6.5). The range is however fairly broad. In Jung, for example, 90% of modified classes are changed less than 3 times.

Furthermore, though most systems have a similar modification profile (see Figure 6.5), a few systems (Axis, iText, Saxon, Jung and ActiveBPEL) show up as clear outliers. Axis, iText and Saxon had relatively higher number of classes that underwent modification, we noticed that Jung and ActiveBPEL were at the other extreme with fewer classes that underwent changes. In 33 systems less than 10% of all modified classes were changed more than 8 times. This suggests that the probability that a class is modified multiple times is quite low and
that this probability reduces non-linearly, as is shown in Figure 6.5. Further, less than 1% of classes were modified more than 20 times (most systems in our data set have at least 20 versions) which shows that only a small proportion of the classes undergo repeated adjustments.

Although there is some variance between systems, most systems in our data set have a similar profile. The range also shrinks as the modification count increases. The data range for modification count of 2 is close to 15%, however the range falls to 7% once the modification count reaches 4 and is less than 3% by the time we consider modification count of over 10. This consistency across different software systems indicates that the number of releases in the history of a software system has a minimal impact on the number of times a class is modified.
Figure 6.6: Number of measures that change for modified classes. x-axis shows the number of measures that have been modified, while the y-axis shows the percentage of classes.

6.2.3 Distribution of the Amount of Change

As software evolves, we have observed that there is a proportion of code that changes, but the number of modifications is minimal. But what is the magnitude of this change? Is there a typical profile for most systems, such as the one we observe in the modification frequency?

In order to observe the amount of change, we focus on the classes in the final version that have been modified at least once in their life cycle. We then compute the set of measures that have changed between a class when it was first created and its state in the final version. We use the final version as the baseline to allow for a qualitative comparison between versions and also to check if there is a broad pattern that is independent of system maturity. This high-level approach allows us to see if there is a common profile across the various systems.

Measuring only the number of metrics that change provides a size change measure that is bounded (since the total number of metrics that can change is fixed, in our study at 54 metrics). The bounded nature of
Chapter 6. Change Dynamics

the measure is necessary as it permits a comparative analysis between different software systems in our data set. Other researchers have suggested using a Euclidean distance measure in order to determine the magnitude of change [34, 67, 155], however, we did not use this measure because the resulting value is unbounded and hence cannot be directly used for comparative analysis [252] across different software systems. Furthermore, an indirect measure that combines multiple metrics is harder to interpret since the rate of change between different metrics is dissimilar. Computing a more representative measure of change by effectively combining different metrics and comparing it with the developers cognitive perception of change is an area of investigation that we intend to tackle in our future work.

Our observations for a representative sample of systems under analysis are shown in Figure 6.6 (a sub-set is shown on the diagram to improve readability of the chart and the outlier systems have been retained to ensure the boundaries are presented).

We found that there is a broad range for the amount of change across the systems, but the overall shape of the distribution is similar. The cumulative distribution curve is fairly linear for 90% of the classes across multiple systems and only 10% of the modified classes have more than 23 measures modified. When we investigated these observations by project, 3 systems (Axis, iText and Saxon) had a higher level of volatility than others, while one system (Jung) showed comparatively low volatility since most classes underwent very small modifications.

The reader may notice (in Figure 6.6) that a few classes do not indicate a change in any of the measures at all. This is possible since the change detection technique that we apply also takes into consideration method signatures, field names and additional metadata. Hence, minor changes in names are detected as a change, but the underlying measures need not change. However, our analysis has revealed that this is only rarely the case and over all versions analyzed, at most 3% of modified classes have a distance of zero, indicating that our metric based approach is effective at detecting the majority of changes. Further, the zero distance change were caused by minor modification to interfaces,
abstract classes or enum declarations. The typical zero distance change in these classes involves modifications such as reordering of parameters in method signatures, renaming fields, parameters or methods, as well as changes to type names (return types or dependent types).

In all of the systems that we studied, we have not found an instance where all 54 metrics have changed. The maximum that we found was 34 changed metric values. However, the set of changed metrics was different for different classes and systems.

### 6.2.4 Modified Classes and Popularity

In the previous sections we addressed the the probability, frequency and distribution of change. But, what characterizes the classes that do change? In particular, does popularity increase the stability of a class? In order to answer this question, within every release across all systems in our data set we analysed the proportion of classes that are popular $p_v$ (Equation 6.2.11), against the proportion of all the popular classes that are modified $c_p^v$ (Equation 6.2.12). In the following two equations, $N^p_v$ is the set of popular classes, $C^p_v$ is the set of popular classes that are modified, and $N_v$ is the set of classes in version $v$.

\[
p_v = \frac{|N^p_v|}{|N_v|} \tag{6.2.11}
\]

\[
c_p^v = \frac{|C^p_v|}{|C_v|} \tag{6.2.12}
\]

In order to ensure sufficient statistical power (in terms of our ability to compare both samples), we eliminate releases where less than 30 classes have changed [271] (Chapter1). Approximately 20% of the releases in our data set have less than 30 modified classes and these have been excluded from our data analysis.
Figure 6.7: Spring In-Degree Count evolution. Proportion of modified classes with high In-Degree Count is greater than that of new or all classes.

We observed the following property (Equation 6.2.13) holds, for 90% of the releases in our data set:

\[ c_p > p_v \]  

(6.2.13)

Interestingly, even if we include all of the data, that is, we consider releases where less than 30 classes have been modified, the above property hold approximately 75% of the time.

Although, this property holds across all releases, are individual systems different? Does this property get violated more in some systems? In order to answer this question, we investigated each system. We observed that in 3 systems (Proguard, Tapestry and Acegi), this property holds only 50% of the time. These systems appear to be outliers (all other systems in our data set had at least an 80% probability). Interestingly, in 17 systems, the property holds 100% of the time. That is, in these systems, the majority of modified classes are popular.
In Figure 6.7 we see a chart for that illustrates our observation in the Spring Framework and shows the proportion of popular classes over the entire evolutionary history. We can see that the proportion of modified popular classes is consistently greater than the proportion of popular classes. Developers ideally would approach a class with high In-Degree Count with care due to the associated risk of impacting other classes that rely on the services provided. However, our observations show that typically, classes with a higher In-Degree Count tend to be modified more than those with lower In-Degree Count.

In the previous chapter we showed that In-Degree Count and size are not correlated (Spearman’s rank correlation coefficient across all systems was in the range $-0.2$ and $+0.2$). This weak correlation implies that popular classes are being modified, but not because they have more code.

### 6.2.5 Modification Probability and In-Degree Count

The analysis presented in the previous section showed that popular classes tend to be change prone. Further, for any given class the baseline probability that it will change in its lifetime is 50%. However, does this probability of modification increase with an increase in popularity? In order to answer this question, we constructed a Logistic regression model where In-Degree Count was the explanatory (independent) variable. The response (dependent) variable was if a class was modified at least once in its release history (0 was used to represent classes that were not modified, 1 indicated that a class was modified at least once in its evolution history). The logistic regression model was constructed by using the set of all classes across all versions of all software systems in our input data set.

The logistic models that we constructed showed that In-Degree Count was a statistically significant predictor of modification probability (Likelihood ratio $\chi^2 = 599.43$ with 1 degree of freedom, p-value was 0.0001). In our model, for every one unit increase in In-Degree Count, the odds that the class will be modified increases by a factor of 1.029 (95% confi-
Figure 6.8: Probability of modification increases as the In-Degree Count of a class increases. This graph is generated based on predicted values from the Logistic regression where In-Degree Count is the independent variable.

dence interval is $1.026 - 1.0319$). The predicted probability derived from the model is presented graphically in Figure 6.8 and shows the probability that a class will be modified increases as it gains additional dependents.

6.2.6 Popularity of New Classes

We have shown in the previous chapter that the distribution of size measures are stable as software evolves. As a consequence we know that overall system size growth is mainly due to the addition of new classes, rather than growth within existing classes. But, what characterizes new classes? How do new classes compare with existing and modified classes? Are classes born popular or do they acquire popular-
ity with evolution? In order to answer these questions, we analysed the proportion of classes that are popular \( p_v \) (Equation 6.2.11), against the proportion of all the popular classes that are new \( a^p_v \) (Equation 6.2.14).

\[
a^p_v = \frac{|A^p_v|}{|A_v|} \tag{6.2.14}
\]

In Equation 6.2.14, \( A^p_v \) is the set of new classes that are popular, and \( A_v \) is the set of new classes in version \( v \).

As in the previous section, where we analyzed modified classes, to ensure sufficient statistical power (in terms of our ability to compare samples), we eliminate versions where less than 30 classes have been added [271].

We observed the following property (Equation 6.2.15) holds, for 88% of the releases in our data set:

\[
p_v > a^p_v \tag{6.2.15}
\]

And, the following property (Equation 6.2.16) holds, for 85% of the releases in our data set:

\[
c^p_v > p_v > a^p_v \tag{6.2.16}
\]

The property captured by Equations 6.2.15 and 6.2.16 shows that, in general, the new classes do not start out popular, and that classes that tend to be modified are those that are popular. This property is illustrated for the Spring Framework in Figure 6.7. Although, developers create some new classes with a higher In-Degree, in general the proportion of these classes is substantially lower than the proportion of popular classes within the software system.

We observed that the In-Degree profile of the new classes is very different from the profile of existing code. New classes tend to start with a substantially lower In-Degree Count, and as they are modified over time move towards the overall trend. If we compare the proportion of
popular new classes with that of all classes, we can see (Figure 6.7) that this proportion is consistently lower than the norm. New classes, therefore, tend to start out with relatively lower popularity and some of these gain dependents over time. The observation that in general new classes are less popular than existing classes is also supported by our finding from the Gini coefficient analysis in the previous chapter. We stated that the In-Degree Count gini coefficient in general increases as systems mature which also shows that new classes do not have the same distribution profile in terms of popularity as the existing classes — if they did, the Gini Coefficient would not change.

6.2.7 Structural Complexity of Modified Classes

In the previous section we have shown that the popularity of a class makes it more change-prone. But, does the internal structural complexity of a class increase the likelihood that a class will be modified? In this section, we address this question by investigating the relationship between the Number of Branches and change.

When a class is modified, there is a certain probability that the branching statements are altered. To appreciate if the number of branching instructions has an impact on the probability of change, we constructed two logistic regression models. In the first model, Number of Branches was the explanatory (independent) variable. For the second model, we used the size normalized Number of Branches as the explanatory variable. The size normalized Number of Branches is a ratio calculated as the number of branches per bytecode instructions in a class. The size normalization was applied to identify if size causes a significant distortion in the probability of modification. That is, the class is being modified because it is bigger rather than because it has more structural complexity. In both models the response (dependent) variable was if a class was modified at least once in its release history (0 was used to represent classes that were not modified, 1 indicated class was modified at least once in its evolution history). The logistic regression models were constructed by using the set of all classes across all versions of all software systems in our input data set.
Both the logistic models that we constructed were statistically significant at a level of 0.05 (for the first model, the likelihood ratio $\chi^2$ was 3944.47 and for the second model the likelihood ratio $\chi^2$ was 2871.54). In the first model, for every one unit increase in Number of Branches, the probability that the class will be modified increased by a factor of 1.040 (95% confidence interval is 1.038 — 1.042). For the second model, the odds that a class will be modified increase by a factor of 8.46 (95% confidence interval is 7.76 — 9.23). The higher value in the second model is due to the size normalization. The predicted probabilities derived from the models are presented graphically in Figure 6.9. In both of these models, even a single branch statement tips the balance of a class being modified to over 50%.

**Figure 6.9:** Probability of modification increases as the Number of Branches of a class increases. The graph on the left shows the relationship between Number of Branches (independent variable) and the probability that a class will be modified. The graph of the right uses the Size adjusted branch count as the independent variable. As can be seen from the graphs, the probability increases independent of the size of the class.
We observed that classes with a higher Branch Count are modified more than those with lower Branch Count. However, as discussed in the previous chapter (Chapter 5) class size strongly correlates with higher Branch Count. But, Number of Branches does not strongly correlate with In-Degree Count, which suggests that classes with complex code need not be more popular than simpler classes and modifications spread out beyond popular classes.

6.2.8 Summary of Observations

The aim of our study into the nature of change in software was to inform developers where they can expect change, the likelihood of a change, and the magnitude of these modifications allowing them to take proactive steps.

The observations in our study show that:

- Relatively few classes are modified as a project evolves, class removal is a rare event, and the set of changed classes in general is larger than the set of new classes in any given version.

- Most classes will be modified at least once in their lifetime, but a small proportion of classes remain unchanged during their entire history. Of the classes that are modified, the probability that it will be modified multiple times is quite low.

- The magnitude of change has a highly skewed distribution. That is, most classes undergo minor changes.

- Popular and complex classes are more likely to change. Interestingly, popular classes do not resist change, on the contrary, popular classes are more prone to modifications.

- Every class has a raw modification probability of 50%. However, even a single branch statement or having a single dependent increases the raw modification probability over 50%. Every new branch increases the probability of modification by 1.04, while each new dependent increases the probability of modification by 1.03.
6.3 Discussion

In the previous section, we summarized our observations. In this section, we discuss how our findings relate to the maintenance activities within the development life-cycle and offer an interpretation for the results.

6.3.1 Probability of Change

We have observed that relatively little code is changed in a software system as it evolves. This is not only reflected by the small number of classes that change, but also the small amount of change within modified classes. Our data also reveals that code is even less likely to be removed than changed. This suggests that developers tend to resist making substantial changes to the existing code once it has been released. We can conclude that any code that is released is likely to resist change.

In all of the systems that we have studied, a good proportion of classes remained unchanged. This indicates that many abstractions tend to stay very stable after they are created. The broad range of values for different systems suggests that this stability depends on the domain and possibly the development approach as well as the architectural style that the team has adopted early the development life cycle. Furthermore, the occasional package rename suggests that even with the availability of tools that can help with refactoring, developers do not rename or move classes between packages often. Developers seem to prefer stability in class names, especially once a version of the software has been released.

Our analysis of the relationship between new classes and changed classes as highlighted by Equation 6.2.10 shows that the set of changed classes is in general greater than the set of new classes. This finding can be used during the planning phase for a new version, specifically in improving the effort estimation. For example, if the initial analysis and design for a new version suggests that there is need for 25 new classes,
the change properties identified suggest that there is an 80% probability that at least 25 classes will be modified. Our observations also identified that as software matures, these properties (Equations 6.2.9 and 6.2.10) are much more likely to hold. Our results suggest that maturity increases the resistance to change as developers are likely to be comfortable with the current set of abstractions.

The change properties themselves cannot be directly used to assess the quality of a software project. Rather, they serve as effective and useful feedback mechanism allowing managers and developers to understand the evolution of a software system better and review the code more effectively. We highlight the value of these properties with two examples which had unusually high violations of the properties.

The first example is the Ant build system, a project that exhibited the greatest amount of violations (in our data set) of the two change properties (Equations 6.2.9 and 6.2.10). Our analysis shows that most of these violations are primarily driven by the plug-in architecture style used in this software system. Ant is a build automation software similar to `make` [258]. This software was conceptualised and built around a plug-in based architecture where new build automation tasks can be developed and added with minimal changes to the existing code. Most of the enhancements to Ant came as new plug-ins that provided new build automation tasks. The core of the system is only modified if a large number of plug-ins are replicating similar functionality. This plug-in architecture allows developers to create and add new classes without requiring many changes in the existing code base. This modular architectural style and the domain of the application in effect meant that in most new releases, there was a greater proportion of new classes compared to modified classes (violating the change properties at a greater rate than expected).

The second example is Struts, another project that also had a large number of violations for the change properties (nearly 50% of the releases). An investigation into the release notes and source code of Struts shows that changes can be attributed to the developers re-writing large parts of the codebase twice. The first complete re-write used
a small portion of the code from the previous versions. The second re-write abandoned much of the code base and started on top of another open-source project. This level of fluctuation, and continuous change effectively meant that developers did not have sufficient time to increase the maturity of the project as it was consistently undergoing substantive changes.

Although a higher degree of violations of these change properties does not imply a poor quality software system, consistent and repeated violations can indicate an unusual phase within the project which may required additional attention or appropriate documentation to properly communicate the state of the project to all stakeholders. This type of summarised information is often masked in software project due to the large volume of changes that take place, and because of a lack of empirically derived heuristics.

### 6.3.2 Rate and Distribution of Change

Modifications are unavoidable as systems are maintained. However, very few of the classes tend to experience a high level of modification. Of the classes that have changed, most have been touched a few times – only a small proportion is modified several times. This modification profile is very similar in all systems under analysis, suggesting that most classes in a system tend to reach a stable state very quickly. Combined with our earlier observation that a good proportion of code is never touched between releases, this suggests that development teams tend to create a stable set of abstractions very quickly. Our findings provide further support to a similar conclusion reached by Kemerer *et al.* [148] based on an analysis of maintenance logs.

Although our approach does not reveal the actual amount of code that has changed, it provides us with a broad indicator. One of the systems that has resisted changes was Wicket, a web application development framework. Interestingly, the rate of modification in Wicket was close to many other systems. However, the observed distribution of these modifications suggests that most of the changes were minor fixes. At the
other end, we have Saxon which shows up as a clear outlier in terms of the rate of modification as well as the distribution of these modifications. As evident from the data, the developers of this system made frequent changes, many of which are substantial. Two other systems had a similar pattern: iText and Axis. But, why was this the case? We consider this question in further detail in Section 6.3.6.

### 6.3.3 Popularity of Modified Classes

In the previous sub-sections we argued that developers construct a stable set of abstractions early in the development life-cycle and that they undertake actions to reduce the size and frequency of these changes as they prefer stable classes. In this section, we consider our observation that classes that are popular tend to be change-prone. This observation is suggestive of Lehman and Belady's first Law of Software Evolution which states that systems that are used will undergo continuing change [168,175]. In our study we see that classes that are heavily used, i.e., that have high In-Degree Count, are more likely to undergo change.

This observed popularity of modified classes does not square well, however, with Martin's Stable Dependencies Principle [189] which states that: "The dependencies between components in a design should be in the direction of the stability of the components. A component should only depend upon component that are more stable than it is." On the surface, the principle appears sound: to improve the overall stability of our system, we should make new things depend on stable and mature components. Unfortunately, our new interpretation of Lehman's Law of Continuing Change suggests that the very fact of depending on a stable component will make it less stable.

This leads us to question Martin's popular Instability measure, which essentially considers an abstraction to be maximally stable when it has only incoming dependencies and no outgoing dependencies, and maximally unstable when it has only outgoing dependencies. Martin’s reasoning is based on “stable” meaning “not easily moved” [189]. How-
ever, this measure confuses stable with inflexible ("unwilling to change or compromise")\(^2\). A more usual definition of stable is "not likely to change or fail" (op. cit.). In this case what Martin calls stable we consider to be unstable, and vice versa.

It is commonly accepted that one should build new software on top of stable abstraction, that is, on mature components with low rates of change. This leads, however, to the paradox that by relying on stable components, we increase their popularity, and thus cause them to become less stable. In effect, this suggests that Lehman and Belady’s laws of evolution also apply to some degree at a micro-scale: *a class that is used will undergo continuing change or become progressively less useful*. For instance, consider the String class (java.lang.String) in the Java environment which would be expected to be maximally stable due to its extensive use. But it has consistently changed. The String class is a popular class that had 30 methods in the first version of Java released in late 1996, the class in the latest stable build (Java 1.6) has 65 methods and was modified to correct a number of defects and to provide new functionality such as Unicode support and regular expression parsing. Though, there is a tendency in popular classes to change, there is likely to be an upper boundary to the external pressures that drive a classes to change, as developers will want to minimise the potential ripple impact from each change.

Our analysis indicates that there is a very strong probability that popular classes are less stable. However, in our data we encountered three outliers — JabRef, Proguard and Acegi where popular classes were just as likely to be modified as non-popular classes. Despite an exploratory analysis into each of these systems, we were not able to identify a common driver that can explain our observation. This indicates that our finding of popular classes being less stable has a strong probability of occurring, but we cannot yet generalise it into a universal principle with well defined constraints.

\(^2\)Oxford American Dictionary, 2005
6.3.4 Popularity of New Classes

We have seen that the distributions of size and complexity remain stable, hence it cannot be that growth mainly occurs in existing classes, but rather in the creation of new classes. But where does this growth occur — do developers build on top of the existing code base, or do they tend to build features as components that are added into a software system?

Since new classes have lower than average In-Degree Count, it seems clear that growth is on top of existing classes. It is highly unusual for a new class to have high In-Degree Count, so there must be little growth below classes of the existing system. This finding is also supported by the Gini coefficient analysis that shows that, in general, the Gini value of In-Degree Count tends to increase as software matures, suggesting that as new classes are added they tend to make use of existing classes hence increasing their popularity. Further, since open-source projects are known to be developed in an incremental and iterative fashion, our observations are consistent with the notion that these systems are built bottom-up, rather than top-down.

6.3.5 Complexity of Modified Classes

We have observed that classes that are modified tend to have a higher Branch Count than the ones that remain unchanged. Why is this the case?

Research into defect-prone classes [31, 213, 318] suggests that complex abstractions will tend to have more defects. Our observation that complex classes attract a higher proportion of modification is consistent with the fact that complex classes tend to have more defects and, therefore, are likely to undergo more modifications as the defects are corrected. Our observations are incomplete, however, since we have not analysed the defect data that will allow us to state that changes to complex classes are principally concerned with correcting defects. Furthermore, it is reported that corrective changes account for only
21% of changes [26], so we cannot conclude that defects are the main reason for change.

A class that changes is likely to have higher In-Degree Count and higher Branch Count. Simply put, complex classes have a higher probability of change. Furthermore, this probability increases with the complexity of the class.

6.3.6 Development Strategy and its Impact on Change

In our data set, three systems stand out as clear outliers: Saxon XML processor, Axis web services library and iText PDF generation library. All of these systems exhibited above average change both in terms of the frequency as well as the magnitude. But, why should this be the case? Is there a common shared trait?

It turns out that all three systems have one common attribute — they were all built to satisfy well defined industry standards. Saxon and Axis implement W3C (World Wide Web Consortium) standards for XML and Web services, respectively. The iText library attempts to satisfy the Adobe PDF specification. Saxon and iText are currently supported by commercial (for-profit) organisations, while the Apache consortium handles the Axis project (Apache is a non-profit organisation that manages a large number of open source projects, including the popular Apache Web server).

An inspection of the architecture documentation, discussion boards as well as release notes revealed a consistent theme in all three systems. In these systems developers made a choice early in the life cycle to incrementally satisfy the specifications, rather than define a software system that completely implements the standards as elaborated in the specifications. In addition, the developers enhanced their software systems as the specifications themselves were expanded and amended. Interestingly, in these three systems developers did not create well defined functional components early in the life cycle. The functional components gained coherence and a clear responsibility over time. For instance,
in Saxon, the developers choose to implement both compile-time and run-time handlers for XSL-T instructions in the same component [144]. they also share memory mapped data structures among different components, increasing the inter-dependencies, which has been shown to increase certain types of defects [130]. Unfortunately, the exact rationale for these choices in Saxon has not been explicitly stated (our inference is that the choice to merge compile-time and run-time handlers in Saxon may have been motivated by a desire to improve the performance).

The architecture guide provided with Axis [11] states that most modules are not functionally separated and attempt to provide a diverse range of functionality, this tight coupling is noted by the developers as an known concern that needs to be addressed in later releases [11]. The developers of Axis, interestingly, do suggest a potential solution that can address the coupling issue in the architecture document [11] but these recommendations have not been implemented (at time of analysis). The decisions that the iText team made with respect to compiling are similar to choices made for Axis and Saxon as inferred from the change logs and release notes [128].

Another aspect common in iText and Saxon was the choice made by the developers to build their own collections data structures (such as a List, Map and Queue) that underwent many changes when they were initially introduced into the code base. Comments provided in the source code suggest that the motivation for creating their own collections data structure was to improve performance. It is possible that at the time of development the developers did not have access to reliable and mature high-performance collections classes such as those that exist freely now like GNU Trove [280] and the Google collections framework [107]. Though, reliable high-performance collections libraries now exist the developers have left their original data structures and algorithms intact.

In iText, Saxon and Axis developers made a number of incremental modifications. Unfortunately, without an underlying architecture and design that allowed them to incrementally meet the specifications, they
had to modify many classes. The developers, in effect, had to make changes in a number of classes in new versions as they attempted to adhere to the standards incrementally. Furthermore, the lack of coherent reusable components and libraries, also meant that they had to improve existing classes rather than upgrade external libraries and components.

To ensure completeness, we investigated two other systems in our data set that share a similarity with iText, Saxon and Axis in terms of their functional domain. The two systems were, Xerces-2 XML processor and Xalan XSL-T processor. Both these products also satisfy standardised XML technology specifications developed by W3C. Interestingly, these projects made very different choices at the start of their life cycle. The Xerces-2 project was a complete re-write of the Xerces-1 platform, however, for the new build, developers satisfied most of the core specification in an initial release rather than via incremental refinement. Further, based on the experience gained from earlier attempts, the Xerces-2 developers built a modular architecture that allowed them to make incremental adjustments [309] to changing specifications, but the bulk of the code remained stable. The Xalan project did not have a previous release and knowledge base to build on top of [307]. The development team, however, chose to create a highly modular architecture with well defined functional boundaries that permitted incremental refinement and the ability to replace libraries and components [308]. They also relied on a number of existing stable libraries to provide common functionality (unlike iText and Saxon where developers built many key data structures and algorithms). The change analysis data suggests that these choices have helped the team as they evolve and adapt the systems.

The development strategy and choices made by the developers has an impact on the change profile. Despite the type of approach taken, all five systems (iText, Saxon, Axis, Xerces-2 and Xalan) have continuously evolved, potentially satisfying user needs. However, our study suggests that when attempting to match formal standards, a modular architecture built after a careful analysis of the entire specification can help reduce the frequency and magnitude of changes. The strategy of using
a loosely coupled modular architecture has been considered ideal in the literature on software design [257], and our observations provide further empirical evidence to support the benefits of this strategy.

6.4 Related work

Change in software has been a subject of study by a number of other researchers and in this section we present key related work to place our own work in context.

Kemerer and Slaughter studied the profile of software maintenance in five business systems at the granularity of modules by analysing maintenance log data. The business systems were closed source and not developed using object oriented programming languages. Kemerer et al. conclude that very few modules change frequently, and those that do are considered to be strategic [148]. Interestingly, our findings are similar, even though our analysis is based on release history and Open Source Systems developed using an object oriented programming language. The commonality in finding suggests that developers approach software change carefully and tend to minimise them as much as possible, independent of the licensing and development method used.

Similar to our work, Capiluppi et al. have analyzed the release history of a number of Open Source Systems [35, 39–41] with an intention of understanding if developers will undertake anti-regressive work, i.e., make changes to reduce the complexity of a certain module and hence improve maintainability. Their studies mainly focused at a macro-level, in particular on relative changes in the code size and on complexity at a module level [41] as well as the influence of the number of developers on the release frequency [39]. Their conclusions were that in Open Source Projects, relatively little effort is invested in anti-regressive work. In our study we found that developers minimize the frequency and magnitude of change, and the abstractions that do change tend to be popular or complex. Interestingly, if developers spend little effort on anti-regressive work, then it implies that the typical modification is to create new features or correct defects.
A recent study into change by Girba et al. showed that classes that have changed in the past are also those most likely to change in the future. They also showed that these classes were the minority [92]. In related and earlier work, Girba et al. have tested the hypothesis that classes that have changed in the past are likely to change in the future [94]. The reliability of this measure of “yesterday’s weather” seems to vary according to the “climate” of a software project. Girba et al. have also studied the relationship between change and developers [95]. Additionally, Girba et al. in their studies tag a class as having changed if methods were added or removed. They also identify change at the method level by observing a change in the number of statement and the cyclomatic complexity. In our study, we used a larger set of properties to detect change and hence are able to identify more fine grained changes. Rather than modeling and understanding the nature of change, their goal was to understand which developers are most knowledgeable about different parts of an evolving system.

Another arc in the study of change has been the area of understanding co-change, where the assumption is made that certain groups of classes or modules change together [85] because related features are grouped together. This assumption was supported by research undertaken by Hassan and Holt who analyzed many Open Source projects and concluded that historical co-change is a better predictor of change propagation [113]. This observation is also supported by Zimmerman et al. [319,320] which led to the development of tools [315] that can guide developers to consider a group of classes when they modify one class. Our study however, aims at understanding the statistical properties of post-release change and inherent properties of a class that might lead them to change. The properties that we identified can improve project plans and help during iteration retrospectives, rather than being able to directly guide developers about potential co-changing classes. Furthermore, Gall et al., Hassan et al., and Zimmerman et al. relied on the revision history in their studies, and identify change during the development phase. We however, investigated the release history and hence are able to provide a post-release change perspective.
Lanza et al. introduced the Evolution Matrix [163] as a means to visualize the evolution of object-oriented software, with the emphasis on revealing patterns of change. Though they are unable to establish which specific parts of the system are likely to change, their visualization approach provides a good overview for developers when they retrospectively audit the evolution of a project in order to better understand the history and make improvements for the future. More recent work in evolution visualization [1, 164, 298] has aimed to highlight changes to the design structure of a software system based on thresholds for metric values at various levels of abstraction (for instance, class or package level). Our work is slightly different in that we provide statistical properties of change. Given the additional information in our study, the visualisation tools can incorporate the change properties that we have identified in order to highlight versions where unusual changes have taken place.

Koru et al. [109] proposed a tree-based classification technique to identify change prone classes. Based on an investigation of two Open Source software systems (KOffice and Mozilla browser) they found that large classes tend to change more than smaller classes, i.e. size is an effective predictor of change. They also propose a tree based classification model to identify change prone classes. The classification method relies on cyclomatic complexity (similar to Number of Branches in our study), Lines of code and average change count (per class) in order to determine which classes may change in the future. In our work, we show that the probability of change increases with an increase in Number of Branches. A strength in our study is the size of the underlying data set from which we base our conclusions. Interestingly, Koru et al. use the average change count in their predictive model, but the average values computed may need some additional adjustment to improve the effectiveness of their approach because we noticed a highly skewed distribution for magnitude and frequency of change. An additional difference between our study and the one undertaken by Koru et al. is in the method used to compute change count. We calculate this value by analysing the release history, however Koru et al. extract change count information by analysing revision history (i.e. parsing CVS log files).
A study by Bieman et al. [28] examined 39 versions of a single industrial object-oriented software system that evolved over a three year period. In this study they investigated the relationship between design patterns, other design attributes, and the number of changes. Bieman et al. found a strong relationship between the size of a class and the frequency of changes, that is larger classes were changed more frequently. This finding fits well with the logical expectation of change, in that larger classes should change more frequently since there is a larger amount of code and hence the probability of change increases with size. The more unexpected and interesting finding in the study was however the findings that: (a) classes that participate in design patterns are in fact more change-prone, and (b) classes that are reused the most through inheritance hierarchies are also change-prone.

More recent work by Aversano et al. [10] and Di Penta et al. [68] based on empirical investigation of evolution in 3 Java software systems (both studies used the same data set) arrived at similar conclusions suggesting that classes that participate in design patterns tend to be change-prone. The findings from the study by Bieman et al., Aversano et al. and Di Penta et al. complement our own observations. Our study, however, is more specific, in that we show that popular and complex classes tend to be change-prone rather than be restricted to purely classes that participate in design patterns or inheritance hierarchies. Additionally, in contrast to the studies by Aversano et al., Bieman et al. and Di Penta et al. that focused on a few software systems, we have undertaken a larger-scale longitudinal study.

Khomh et al. [149] undertook a change-based study investigating 2 Java software systems. They show that classes with code smells are more change prone than others. The term “code smell” is suggestive of poor quality in some aspects of the the source code, and these smells are conjectured to negatively impact on the software maintenance activities [81]. Khomh et al. identified changes by analyzing the revision history (source control log files), and then checking if the modified code exhibited any of the 29 different code smells that they considered to be representative of poor quality. They conclude that classes with code smells were more likely to be changed. In their study, one of the code
smells that they consider is related to excessive complexity within a class (computed as the sum of method cyclomatic complexities [190]). In our work, we use the Number of Branches (NOB) as the measure of structural complexity within a class. The NOB metric that we use can be considered to be equivalent to the the cyclomatic complexity metric used by Khomh et al. since both metrics are based on the same underlying measure (see the NOB metric definition in Chapter 4). The findings by Khomh et al. are similar to our own conclusions with respect to the structural complexity of a class suggesting that structurally complex class are change-prone during development, and post-release. We also, identified a consistent relationship between popularity and probability of change which was not considered within the work by Khomh et al..

In a recent study, Capiluppi et al. [36,37] investigated the value of stability at predicting reuse of a software module (identified as a set of files within a source code folder) by studying evolution in 4 long-lived open source software systems. Capiluppi et al. argue that highly stable modules are good candidates to be tuned into independent reusable modules. Capiluppi et al. measured stability by using Martin’s Instability Metric [189] which uses the coupling of a module (similar to the In Degree and Out Degree count in our work) as the basis for calculating the stability. However as discussed in Section 6.3.3, Martin’s instability metric does not measure actual stability of an abstraction over time and hence cannot be considered to be a direct measure of actual change. We also did not find any rigorous longitudinal studies that have shown the relationship between Martin’s instability metric and actual changes that take place within a module. In contrast, our work shows that the new dependants place additional pressure on a class increasing the probability of change to satisfy new clients. A key distinction from our own study and that by Capiluppi et al. is the level of abstraction used in the study. We focus on changes at the class level, while Capiluppi et al. study modules. Although, the level of abstraction used in our study is different, a module can be treated as a set of related classes. Therefore, if a set of classes within a module change, we can considered the module to have changed as well and hence our findings are likely to be similar at the module level as well. However,
a serious investigation of change at higher level of abstraction remains to be done and we intend to consider it in future work.

6.5 Limitations

In order to place our study in context, we highlight some of the known limitations of our approach in addition to our findings.

The change detection method used in this work may miss a small set of classes if the change is not measurable in the metrics under consideration. The accuracy can be improved by adding further metrics and additional aspects like a comparison of the call graph. The value of this additional complexity remains as a topic for consideration in our future work. Furthermore, our distance measure compares the initial version of a class to the final version. This method will miss edits where a class is modified and returned back to its original shape, as seen by our change detection technique. This limitation can be addressed by computing a distance measure incrementally. As a consequence, the analysis approach will need to be adjusted to take into consideration the way the metric information is being collected.

Our method of computing modification misses classes that have been renamed; they will be considered as a deletion (in the earlier version) and an addition (to the later version). However, of the few classes that are deleted (less than 4% in general), a subset might be class renames. We checked if a class has indeed been renamed manually in 300 randomly selected instances. Less than 5% of the classes that we checked can be considered refactored with a name change. Based on this observation, the benefit from using a complex algorithm to detect renamed classes is likely to have at best a minimal overall impact and our overall conclusions and recommendations will still hold.

We have also not investigated in depth why changing classes have higher than normal In-Degree Count. We speculate that the introduction of new clients creates the need for adaptations. Other possibilities are that (i) new clients introduce new requirements, but that suggests
new growth in existing classes, which we did not consistently find, or (ii) new clients exercise existing classes in new ways, thus uncovering previous unknown defects. Further work is needed to discover which, if any of these hypotheses is valid.

6.6 Summary

Change in software systems is unavoidable as they are adapted to meet the changing needs of the users. Our study shows that when we look at the latest version of a given system, around a third (or more) of the classes are unchanged in their lifetime. Of the modified classes, very few are changed multiple times, and the magnitude of change is small suggesting an inherent resistance to change. These findings show that maintenance effort, which is considered to be a substantial proportion of the development effort (post initial versions) is spent on adding new classes. In the absence of a substantial architectural shift or a rewrite of the system, much of the code base resists change. Furthermore, efforts to base new code on stable classes will inevitably make those classes less stable as they need to be modified to meet the needs of the new clients.

In the next chapter, we discuss the implications arising from our observations. Specifically, we consider how our findings relate to the Laws of Software Evolution, and how our work can help improve software development practices.
Chapter 7

Implications

In previous two chapters we addressed the research questions that motivated this thesis. In this chapter we discuss the implications arising from our observations. Specifically, in Section 7.1 we discuss how our findings provide support for some of the laws of software evolution, and offer recommendations to help improve software development practices in Section 7.2.

7.1 Laws of Software Evolution

Do our findings provide support for the Laws of Software Evolution? [175]. If so, which specific laws?

Lehman and Belady pioneered the study of evolution of large-scale software systems, and established the well-known “laws of software evolution” [168]. As discussed in Chapter 2, the laws were originally framed and refined based on a study of few large software systems. Additionally, the analysis methods in previous studies focused heavily on inferring support for the laws from models of aggregate size growth, and observation of change during the construction of a software system. In our study, we investigated evolution of the distribution of size and complexity metrics, and post-release change and hence we are able to present a discussion on the laws from a different frame of reference.
Specifically, we find support for First law Continuing Change, third law Self Regulation, fifth law Conservation of Familiarity, and the sixth law Continuing Growth. However, our analysis was not able to provide sufficient evidence to show support for the other laws.

**Change**

Our observations show that in all cases the metric distributions continuously changed. Providing support for the first law software evolution – Continuing Change. The only consistent facet in our data was that in most cases, the change as measured by the difference in Gini Coefficient values was small. This small level of change indicates that a certain level of stability is common and potentially desirable for evolution to proceed since development teams are most productive with stable abstractions or those that are changing at a rate that allow for developers to learn them. Rate of change must be in synchronization with the development teams ability to learn, adapt and effectively utilize these abstractions. If the rate of change is too slow or zero, then it would imply that the system may slowly deteriorate as it no longer meets changing/maturing user needs. Conversely, if the many abstractions (classes) change too quickly, the effort spent on learning the changes will be substantially greater than the effort spent on adding new features.

**Complexity**

The second law, Increasing Complexity states that “software is changed, its complexity increases and becomes more difficult to evolve unless work is done to maintain or reduce the complexity” [175]. We find that this law is hard to provide a definitive support to, mainly because complexity is a hard term to universally quantify (as discussed in Chapter 4). Our analysis indicates that system complexity is mainly due to growth in general (primarily caused by new classes), rather than a consistent increase of size and structural complexity of the individual classes. This conclusion provides an additional dimension to Lehman’s law of increasing complexity [168]. Software gains volumetric complex-
ity, that is, because there are a larger number of abstractions that developers have to deal with over time. However, when we consider the distribution of size and complexity metrics, we notice that the distribution of all measures fall within a tight boundary (see Chapter 5). In any given software system, though there is variation, the overall consistency in the shape suggests that developers either explicitly or implicitly take correction action in order to maintain the overall distribution of the size as well as complexity. The only material increase in complexity at the system level is not a consequence of complexity of a certain set of classes, but rather due to the size of the system as a whole and increase in the absolute number of complex classes.

Our observations related to complexity introduce uncertainty in interpreting the law of Increasing Complexity. The law as stated suggests that complexity increases unless effort is put into reducing it, that is, it implies that complexity can be managed or reduced. In our study, we find that (i) there is an absolute increase in the total number of classes, and (ii) developers either intentionally or otherwise manage to contain the distribution of complexity between the various classes. These findings lead us to argue that although the distribution of complexity can be managed, the volumetric complexity that arises because of the size of the system cannot be reduced without a corresponding reduction in functionality (assuming a competent development team).

**Feedback and Regulation**

The third law, Self Regulation states that evolution is regulated by feedback [166]. We do not collected direct quantitative data related to feedback. However, the bounded nature of Gini Coefficients across multiple software systems provides indirect support for this law. Feedback can be caused by different drivers. For instance, developers will get feedback from a set of classes that are too large or complex, especially if these classes are defect prone. Feedback can also be driven by team dynamics as well as the cultural bias (strong team habits) that influence how software is constructed. Hence, it is likely that there are potentially multiple sources of feedback at work in order to regulate the
overall structure and shape of the distribution. We found no evidence that developers actively measure the metric distribution and aim to fit within it, as there is no traces of this type of behaviour in the change logs, development process documents or release notes that we studied. The interesting aspect is that developers have no direct knowledge of the overall distribution, yet the set of choices they make ensure that the overall profile is highly consistent. This suggests that feedback operates at the local level rather than at the system level. That is, developers notice that a certain class is too complex and take steps to correct that. Similarly, refactoring efforts [81] in most cases are likely to be choices made with local information.

This behaviour of developers relying on localised feedback is indicative of software maintenance being a stochastic process that generates an emergent pattern when observed at the system level. We define emergence in this context as a process that drives a system to acquire a spatial, functional and temporal structure without specific directed interference from outside. By specific, we mean that the structure or functioning is not the result of directed and planned actions, rather the system achieves its structure via a series of autonomous actions that are taken at random intervals for a range of purposes. Interestingly, software is not unique in this emergent behaviour as this pattern has also been identified in human engineered, social and natural systems [111].

What are the set of causal factors that cause this observed phenomenon? We do not have a direct answer yet of the causal factors, but the consistency of the observations gives rise to the speculation that developer decisions are driven by some form of cognitive preference that makes developers choose solutions with certain typical profiles. It appears that there exists an acceptable range of complexity for programmers and this range is domain-independent since the Gini coefficients across all analyzed systems fall within a tightly-bounded interval and is independent of the Java language semantics as well as development-environment neutral. The Java programming language does not, in any way, limit developers to choose between design alternatives or forces them to construct software with the observed distribution profiles and hence the
language semantics is not the likely driver. Similarly, the development environment and tooling used are also not likely to be the primary contributing factor, since they too do not force developers to build software with highly skewed metric distributions.

Given the emergent nature of the overall statistical properties and the consistency across multiple software systems, there cannot be a finite and deterministic set of causes that drive developers to organise solutions within a constrained design space. Rather it is likely an interaction between many choices that gives rise to this typical organisational pattern.

Familiarity

The fifth law (*Conservation of Familiarity*) suggests that developers grow the software system at a rate that allows the development team to maintain familiarity with the code base. Earlier research in this field (as discussed in Chapter 2) found support for this law based on their observation of a global sub-linear growth rate in evolving software system [175, 188, 193, 239]. In our study, we support this law based on the observation that developers, in general, keep the set of changed classes much smaller than the set of unchanged classes (as discussed in Chapter 6). Further, the changes made are small and infrequent suggesting an inherent tendency towards maintaining familiarity, especially in existing code. We also find additional support for this law based on our observation that the metric distributions in general fluctuate within a very tight boundary, suggesting a strong preference for familiarity.

Growth

The sixth law (*Continuing Growth*) states that “evolving software must be continually enhanced to maintain user satisfaction”. Our observations (cf. Chapter 6) show that software systems are built incrementally bottom-up and that there is a small proportion of code that constitutes new classes in every release. Though, new classes need not always con-
tain new functionality (which is possible if new classes were created due to refactoring), the consistency with which new classes are added suggests that developers grow software systems by adding functionality in new classes.

7.2 Software Development Practices

We now discuss some of the implications of our findings and offer recommendations to help improve software development practices with a specific focus on how maintenance activities can be improved by software project managers, designers, testers and tool builders.

7.2.1 Project Management

Our results show that evolution proceeds by accumulation and combination of stable intermediate states, suggesting that the stability plays a role in ensuring successful evolution. In this context, stability implies the observed strong boundaries in the metric distributions. This knowledge can be directly used during project planning, specifically during effort estimation. Consider a situation where a preliminary high-level analysis of a new feature indicates the need for 50 new classes. We know that it is highly likely that these 50 classes will not substantially alter the overall shape and structure of the underlying distribution. That is, we can use the size and complexity probability distribution histograms to estimate the size and complexity distribution of these 50 classes which can be used as input into the effort estimation process. Further, our findings also provide additional support for the recommendation of collecting and using historical data to improve accuracy of estimation.

Although substantive changes in Gini Coefficient values are rare, these changes do happen. The real value for project managers is that this knowledge can be used as a trigger to ensure that the rationale and impact of changes are properly communicated. Furthermore, monitoring trends in the Gini Coefficient values can reveal an unknown
or poorly understood dynamic within the software system. Though, it would be natural to expect the project managers to be aware of all key changes, this may not always be possible, for instance in an outsourced project or where a large number of geographically dispersed developers are working on a software system. In such projects, monitoring the Gini Coefficient value can provide feedback to project managers. If the managers are able to explain the trend and changes in Gini Coefficient values, then they only have to ensure that these modifications are adequately captured in the documentation. However, if the managers find unexpected patterns, then it can be used to take corrective action (as the changes may be unplanned) or to clarify their own understanding of the software system’s growth dynamics.

Based on our findings, we do not offer a recommended ideal range of Gini Coefficient value since we did not directly investigate the impact of the Gini Coefficient coefficient value on productivity or quality attributes. However, managers can use the IRQ range as a broad guideline on what to expect and use this range as a reference to understand the structure of the software system better by triggering an investigation if many metric Gini Coefficient values are at the upper end of the expected range. Similarly, we do not offer a specific and constant threshold for change, but recommend a threshold be established based on an analysis of historical data. Further, this threshold should be revised at regular intervals to ensure that it reflects the current risk profile of the project. In our study we used 4% relative change between Gini Coefficient values measured from two consecutive releases. If no historical data is available, we recommend that managers use a relative change of 4% as a starting point and tune it based on project specific risks. If, however, historical data is available, then managers can measure the changes in Gini Coefficient values at previous substantial change points and use that as a reference. Our own observations suggest that Gini Coefficient values will change less as software systems mature and hence the threshold is likely to need periodic revision based on the projects needs.

The nature of our recommendations are qualitative and restricted to macroscopic changes. This is the price to be paid in order to find suf-
Chapter 7. Implications

sufficiently broad general principles and rules that can be applied across a range of different software systems. Identifying more fine grained recommendations is an aspect that we intend to tackle in future work.

Planning and Estimation

The change properties that we identified (Equations 6.2.9 and 6.2.10) show that, in general, the set of changed classes is larger than the set of added or removed classes. This finding can be used as a simple heuristic during the planning stage of a new release. For example, consider the situation where developers estimate the need for 20 new classes to provide a new feature. The second change property (Equation 6.2.10) implies that there is a high probability that at least 20 existing classes will be modified. Though some of the modifications are likely to be for defect correction and not directly to support the needs of the new classes, this guideline can still be used during effort estimation. Conversely, evolving software systems where the change properties are consistently being violated are indicative of an architectural style (e.g. plug-in based) that can sustain this abnormal change pattern. Hence, in software where the history does not suggest that these change properties hold strongly, developers can estimate the effort independent of the likely modifications to existing set of classes.

7.2.2 Software Metric Tools

A number of commercial and open source tools are currently available to measure software systems with many being provided as plugin's into popular Integrated Development Environments like Eclipse [73]. The typical approach adopted in these tools [47, 51, 196, 203, 224, 226, 251, 274, 295, 296] to summarise the large amount of metric data is to use statistics such as mean, standard deviation and median. The tools also typically allow the users to set thresholds for various metric values (for example, flag all methods that have a cyclomatic complexity greater than 15) based on the advice provided in software metrics literature [141, 165, 181].
The key argument in our study is that when dealing with skewed distributions, it is hard to derive sensible conclusions from descriptive statistical summary measures such as mean and median. Hence, we recommend that software metric tool developers should present the distribution histogram (relative and cumulative), the Lorenz curve and the Gini Coefficient values. When this information is shown along with the typical range for the Gini Coefficient values, the tool users can reason about the nature of their software more effectively. Furthermore, rather than remove the widely used summary measures (such as mean and standard deviation) the tools should display a warning about the limitations of these measures. This approach will serve to inform and educate the users allowing them to interpret the data better.

The second recommendation based on our study is related to the feature allowing users to set thresholds for metrics. The common approach that is currently provided allows users to identify abstractions that breach a threshold value. Tool vendors suggest that this feature is helpful in identifying complex classes or methods. However, our observations show that it is normal in software systems to have a small proportion of classes and methods that are significantly more complex than others. We also found that this typical skewed nature does not impact on the evolution as many systems in our data set were able to sustain change and growth over many years. Based on these findings, we recommend that thresholds should be based on relative proportions rather than absolute values. For example, the tools should allow users to set 10% of the methods can have a cyclomatic complexity greater than 7. This approach also has the advantage of educating the users about the nature of the data that is being measured. Where sufficient historical data is available, the range for acceptable proportional threshold values can be determined from previous successful projects.

### 7.2.3 Testing and Monitoring Changes

Software testing literature recommends that developers should target complex classes when testing object oriented software [29], as the complexity tends to increase defect density [205]. Our findings provide
Chapter 7. Implications

additional support to this recommendation. However, this recommendation should be extended to ensure testing covers both complex and popular classes. But, is there a specific value that can be used as the threshold? Though a specific and well defined value is appealing, different projects have different needs and hence a universally applicable threshold is not practical. However, in our study we were able to generate modification probability models (See Figure 6.8 and Figure 6.9). Depending on the risk profile of a specific project (at a specific instance of time), developers can use these models to determine an appropriate modification probability threshold.

7.2.4 Competent programmer hypothesis

The component programmer hypothesis states that programmers write code that is close to being correct [211]. This hypothesis is used as one of the fundamental assumptions to support mutation testing [212]. Interestingly, though this assumption has been widely used, we were not able to find studies that verify this assumption. Our findings provide some empirical support for the competent programmer hypothesis. Though change is unavoidable, we observe that most of the changes made to classes are minor and the number of times a class is modified in fairly small. The consistency of minor changes in our study indicates that developers tend to deliver code that adequately satisfies the requirements by the time of a formal release.

Although our observations are suggestive of the competent programmer hypothesis, a system that has an abnormal modification profile does not imply that the programmers are incompetent. Rather, it should be used as a trigger to investigate potential underlying causes. For example, the unusual change profile may be driven by a high degree of volatility in the software requirements, or changes in the external domain.
7.2.5 Creating Reusable Software Components

Agile methodologies [23, 49, 189, 301] recommend that the design of a software system should ideally be to satisfy the requirements at hand and not aim to satisfy potential future needs. Interestingly, our observations suggest that developers, in general, inherently follow this approach (at the class level). That is, a class is given a set of responsibilities that it needs to satisfy its current dependents. We also found that developers do not create a class that is very popular directly, rather classes slowly gain additional dependents over time (Equation 6.2.16).

The findings outlined in Chapter 6 are also supported by findings from the Gini coefficient analysis of In-Degree Count (as discussed in Chapter 5). In most systems, popular classes gain additional popularity over time suggesting that developers expand the responsibilities of a class as needed. Our observations indicate that developers do not create highly used classes directly, rather classes are made popular over time. In essence, developers in Open Source Projects appear to follow the principle of “YAGNI” (You ain’t gonna need it) from eXtreme Programming [23] implicitly. The YAGNI principle suggests that developers should add functionality only when it is needed rather than in anticipation of a potential requirement.

Our findings have implications during software design, specifically when a component is created with reuse in mind. Creating a reusable component requires the designers to slightly broaden its responsibilities in anticipation of various situations where it might be used. Furthermore, a reusable component inherently would be more popular. However, our observations suggest that a reusable component is also a change prone component and this facet should be used as part of the tradeoff analysis when designing software. The recommendation provided by agile methodologies can be better communicated if developers are fully aware of the implication of creating a reusable software component, specifically the probability that you make it more change prone.

A related application of our finding is to improve the process of selecting software frameworks and libraries. For example, consider the scenario...
where a software architect needs to select a library to process image files. An analysis of evolution history can be used to determine if the change profile suggests some stability. Though, this may not be the only factor in determining which library to use, this information can be used to trigger an additional investigation to ensure that the risks arising from a volatile software system can be adequately mitigated.

7.3 Summary

Change in software systems is unavoidable as they are maintained to meet the changing needs of the users. Based on the observations in our study of software evolution we found consistent support for the applicability and validity of the following laws of software evolution: First law Continuing Change, third law Self Regulation, fifth law Conservation of Familiarity, and the sixth law Continuing Growth. However, our analysis was not able to provide sufficient evidence to show support for the other laws.

There are a number of implications that arise from our findings. In particular, we discussed how managers can monitor the changes and trigger a deeper investigation to explain abnormal changes as well as use the properties and thresholds identified to reflect on the development process. We also recommend that managers use the change properties outlined in Chapter 6 during planning and estimation and present the implications for software design. Specifically, we argue that reusable components should be designed to be flexible since our findings suggest that these components are change-prone.
Chapter 8

Conclusions

The long-term effects of evolution on software systems have been studied for over three decades, however there has been little research into understanding how growth and change are distributed over parts of software systems. In our study, we analyzed software metrics collected from forty non-trivial Java Open Source Software Systems, comprising over one thousand distinct releases in order to better understand the nature of growth and change in evolving software systems.

8.1 Contributions

The key contributions in this research effort are:

• We found consistent support for the applicability of the following laws of software evolution [175]: First law *Continuing Change*, third law *Self Regulation*, fifth law *Conservation of Familiarity*, and the sixth law *Continuing Growth*. However, our analysis was not able to provide sufficient evidence to show support for the other laws.

• We investigated how software metric data distributions (as captured by a probability density function) change over time. We confirm that software metric data exhibits highly skewed distri-
butions, and show that the use of first order statistical summary measures (such as mean and standard deviation) are ineffective when working with such data.

• We presented a method to effectively summarise the skewed data distributions using the Gini coefficient [91]. Using this approach, we show that large and complex classes do not get bigger and more complex (internally) purely due to the process of evolution, rather there are other contributing factors that determine which classes gain complexity and volume. However, we found that, in general, popular classes gain additional popularity due to the process of evolution.

• We showed that metric distributions have a similar shape across a range of different system, and that the growth caused by evolution does not have a significant impact on the shape of these distributions. Further, these distributions are stable over long periods of time with only occasional and abrupt spikes indicating that significant changes that cause a substantial redistribution of size and complexity are rare. We also show an application of our metric data analysis technique in program comprehension, and in particular flagging the presence of machine generated code.

• We showed that in general, code resists change and the common patterns can be summarized as follows: (a) most classes are never modified, (b) even those that are modified, are changed a few times in their entire evolution history, (c) the probability that a class will undergo major change is very low (i.e. classes tend to undergo minor tweaks, rather than substantial re-writes), (d) complex classes tend to be modified more often, (e) the probability that a class will be deleted is very small, and (f) popular classes that are used heavily are more likely to be changed.

• We found that maintenance effort (post initial release) is, in general, spent on the addition of new classes. Interestingly, efforts to base new code on stable classes will make those classes less stable as they need to be modified to meet the needs of the new clients. A key implication of our finding is that the Laws of Software Evolution also apply to some degree at a micro scale: “a class that is
used will undergo continuing change or become progressively less useful.”

• We presented a set of techniques to identify substantial and systemic changes. These techniques can be used during the design phase to reason about impact of changes as well as at the end of an iteration to ensure that these changes have been adequately captured in the documentation and properly communicated within the development team.

• We contributed a corpus of software evolution release history to enable further research. Our data set comprises forty object oriented software systems developed in Java spanning over one thousand releases. We also provide to the research community a tool to extract software metric information and generate reports from the evolution data.

8.2 Future Work

In this section, we suggest some possibilities for future research in the area of software evolution.

Software Metric Distribution Analysis

A consistent theme in our study was the observation that developers prefer to organize their solutions around a similar, and consistent design structure since the metric distributions stay within a tightly-bounded value space. Furthermore, evolution does not substantially distort the shape of the distribution that is created in the initial releases of a software system. This gives rise to the speculation that developer decisions are driven by some form of cognitive preference that makes developers choose solutions with certain profiles, and that these preferences are arrived at early in the life cycle of a software system. As a consequence, the perceived practical design possibilities occur in a narrow and, hence, predictable design space. Another facet common to our observations is that few god-like classes shoulder most of the work.
Chapter 8. Conclusions

Our findings are inconsistent with some of the key literature on object oriented design which suggests that these systems should be designed to ensure that responsibilities are well balanced among the classes [302], and god classes should be avoided [237]. However, developers do not appear to adhere to these recommendations. But, why is this the case? One possibility is that developers are able to deal with complexity within an individual abstraction, but struggle when they have to hold a large number of abstractions in their short-term memory. Although, we do not have direct experimental evidence to support this possibility, research into human memory supports this position [12].

In our work we show that Java software developers tend to create and evolve software with these skewed distributions. However, we do not yet know the key drivers for this behaviour, and if the structure arises due to the limitations of human memory (which would make it a more broad property). This is an area of future work arising from our study, where we seek to establish why these skewed distributions arise. Additionally, we also seek to conduct experiments to verify if it is possible to construct, and evolve a software system with a more equitable metric distributions.

Visualizing Change and Evolution

Software visualisation, specifically the sub-domain which aims to present evolution visually is a developing field that has shown some promising results [59, 65, 95, 163, 298]. Though, there are many challenges in constructing appropriate and effective visualizations, our work offers a large amount of raw data that can be used as input by the visualisation tools. In future work, we plan to use the insights gained in our work in building change visualisation tools. Specifically, communicating substantive changes within a software system visually.
Beyond Java based Software

In our study, we focused on Java Open Source Software Systems. However, a natural question that arises is if software developed in other object oriented programming languages will exhibit similar patterns. In one of the peer-reviewed publications [289] arising from this thesis we investigated the metric data distribution in four programs developed in C# [114] targeting the .NET platform [275]. We found similar patterns to that in Java software systems. However a larger longitudinal study is needed to validate our findings. A more interesting tangent in terms of future work is to use the techniques that we put forward in this thesis on software systems developed in programming languages based on a different paradigm. For instance, functional programming languages and even languages such as Javascript where the cultural and development pressures are potentially very different.

Currently, very little is known in terms of the specific pressures that a programming language might place on the evolution of a software system, and this would be an avenue worth exploring as it can inform language designers as well as developers. Language designers can use the Gini coefficient in order to determine how developers use the various features within a language, and more importantly they can observe if certain features are gaining traction while others are not (i.e. are certain features gaining more wealth than others). This type of feedback can be used to tune the semantics of the language and potentially to also refine the libraries to ensure that they provide a more appropriate set of functionality to the end users.

Towards Computational Software Evolution

There is potential to construct software evolution simulation systems, similar to those used in economic and other dynamic systems. An emerging and relatively new field of research in the area of economics is agent based computational economics [8, 273]. Researcher in this field create a simulated economy within a computer system, where the
Chapter 8. Conclusions

humans and other participants that comprise the economic system are modelled as agents with a certain set of behavioural traits. The advantage of these models is that they permit experimenting without having to run time consuming, and costly real-world social experiments. Another benefit is that this model allows researchers to put forward hypothesis that can be tested within a simulation permitting the development of a stronger theory underpinning the observed dynamics. A similar research approach is also employed by scientists developing climate models where the dynamics are simulated rather than derived from static models [278]. These simulations are currently in use for predicting future climate scenarios [115], and the economics simulation models are being developed to help avoid another severe financial crisis [33, 76].

Although the long-term effects of evolution on software systems have been studied now for over three decades, there has been no widely accepted scientific theory that explains this phenomenon, and we are yet to formally identify the long term impact of decisions that developers routinely make [169]. Creating an agent-based model of the software development lifecycle where developers are represented and modelled as simulated agents may identify some of the key drivers and certainly may allow us to estimate the likely impact of a certain category of decisions. Such an undertaking will require close feedback between simulation, testing, data collection and will be an important step in the eventual development of the theory of software evolution.

This field of research also demands a multi-disciplinary collaboration ideally between software engineers, psychologists, and social scientists in order to properly identify the variable that need to be within the simulation. These evolution simulations can be tuned and improved by making use of the results from our study as we describe real-world evolution, and a good computational software evolution model should produce output that is similar to the patterns that we find in actual software systems.
Software engineering literature provides us with a diverse set of techniques and methods on how to build software. Over the past few decades we have seen the process of software construction developing from a collection of ad-hoc rules towards an engineering discipline based on sound, quantifiable principles. However, despite a wealth of knowledge in how to construct software, relatively little deep knowledge is available on what software looks like and how its internal structure changes over time. By providing a deeper insight into how successful software systems evolve, this thesis contributes towards closing this gap and aids in the progress of the discipline of software engineering.
Appendix A

Meta Data Collected for Software Systems

The meta data captured for each software system (see Table 3.3) in our data set is summarised in Table A.1. The release history as well as the meta data used in our study is released as the "Helix Software Evolution Data Set", and it available online at:


<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Full name of the software system</td>
</tr>
<tr>
<td>Short name</td>
<td>Short name (one word) of the software system. This name is used in the reports.</td>
</tr>
<tr>
<td>Type</td>
<td>This can one of the following values: Application, Library, or Framework.</td>
</tr>
<tr>
<td>Is GUI</td>
<td>This is a flag that is turned on if the system provides a graphical user interface for all, or some part of the functionality.</td>
</tr>
<tr>
<td>License</td>
<td>The open source license that this software system is distributed under.</td>
</tr>
<tr>
<td>Commercial</td>
<td>This is a flag that is turned on if the project is sponsored by a commercial organisation.</td>
</tr>
<tr>
<td>Project URL</td>
<td>This the URL of the project’s primary website.</td>
</tr>
<tr>
<td>Issue Log</td>
<td>This is a URL. It can be either a pointer to a local file, or a URL to the issue log.</td>
</tr>
</tbody>
</table>

Table A.1: Meta data captured for each software system.
Appendix B

Raw Metric Data

The raw metric data files for all of the metrics defined in Chapter 4 are located in the directory `data/rawmetrics` on the DVD. The metrics for each software system (cf. Table 3.3) are in a separate data file. The tool that we used to extract the metrics is available at:


The raw metric data files contain the following columns:

- Fully Qualified Class Name.
- Metric Name (We use the abbreviations as defined in Chapter 4).
- A list of metric values for each release of the software. Columns with empty values indicate that the class did not exist in that release.

Sample of the data that is available in the raw metric file for each system is shown in the box below. We list out the Number of Methods (NOM) metric for 6 releases of 2 classes in the Apache Ant system. As can be seen below, the second class did not exist in the first 4 releases and hence has empty values for the metric.

```
org/apache/ant/AntClassLoader,NOM,8,13,17,43,43,46
org/apache/ant/AntTypeDefinition,NOM,,,,20,20
```
Appendix C

Mapping between Metrics and Java Bytecode

Table C.1 shows the list of metrics where the Java Virtual Machine opcodes are used to compute the value.

<table>
<thead>
<tr>
<th>Abbv.</th>
<th>Name</th>
<th>Opcodes Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>THC</td>
<td>Throw Count</td>
<td>ATHROW</td>
</tr>
<tr>
<td>MCC</td>
<td>Method Call Count</td>
<td>INVOKEINTERFACE, INVOICESPECIAL, INVOKESTATIC, INVOICEVIRTUAL</td>
</tr>
<tr>
<td>OOC</td>
<td>Instance Of Check Count</td>
<td>INSTANCEOF</td>
</tr>
<tr>
<td>CAC</td>
<td>Check Cast Count</td>
<td>CHECKCAST</td>
</tr>
<tr>
<td>TCC</td>
<td>Type Construction Count</td>
<td>NEW, NEWARRAY, MULTINEWARRAY</td>
</tr>
<tr>
<td>CLC</td>
<td>Constant Load Count</td>
<td>LDC</td>
</tr>
<tr>
<td>PLC</td>
<td>Primitive Load Count</td>
<td>ILOAD, FLOAD, DLOAD</td>
</tr>
<tr>
<td>PSC</td>
<td>Primitive Store Count</td>
<td>ISTORE, FLOAD, DSTORE</td>
</tr>
<tr>
<td>ALC</td>
<td>Array Load Count</td>
<td>AALOAD, BALOAD, CALOAD, DALOAD, FALOAD, ILOAD, SALOAD, LLOAD</td>
</tr>
<tr>
<td>ASC</td>
<td>Array Store Count</td>
<td>AASTORE, BASTORE, CASTORE, DASTORE, FASTORE, IASTORE, SASTORE, LASTORE</td>
</tr>
<tr>
<td>FLC</td>
<td>Field Load Count</td>
<td>GETFIELD, GETSTATIC</td>
</tr>
<tr>
<td>FSC</td>
<td>Field Store Count</td>
<td>PUTFIELD, PUTSTATIC</td>
</tr>
<tr>
<td>IOC</td>
<td>Increment Operation Count</td>
<td>IINC</td>
</tr>
<tr>
<td>NOB</td>
<td>Branch Count</td>
<td>GOTO, GOTO_W, ATHROW</td>
</tr>
<tr>
<td>GTC</td>
<td>Goto Count</td>
<td>GOTO, GOTO_W</td>
</tr>
</tbody>
</table>

Table C.1: Metrics are computed by processing opcodes inside method bodies of each class.
Appendix D

Metric Extraction Illustration

The program in Listing D.1 illustrates how our extraction approach counts the various metrics from the code. This block of Java code is a syntactically and semantically valid Java program that highlights some of the more complex metrics (not directly intuitive like a simple count such as *Number of Methods*) that we extract from a class. The values computed for the metrics for the class are listed at the top of the program in the header comment (see Table 4.3 for full form of the abbreviations used). The comments surrounding each line of code provide an explanation of the metrics as computed for that line.

The program listing is on the next page.
/** Total counts for a subset of metrics are included in the header
 * LIC (Load Count) = 15, SIC (Store Count) = 2, NOM (Number of Methods) = 2
 * NOB (Number of Branches) = 6, TCC (Type Construction Count) = 1
 * MCC (Method Call Count) = 3, THC (Throw Count) = 1, EXC (Exception Count) = 1
 * ODC (Out Degree Count) = 4 (String, Exception, PrintStream, System.out)
 * NOF (Number of Fields) = 3, IOC (Increment Operation Count) = 3
 * LVC (Local Variable Count) = 3
 */

public class MetricCountExample
{
    private boolean bField; // un-initialised field

    // 2 LOADS: this, constant integer
    private int index = 5; // 1 STORE

    // 2 LOADS: this, constant string
    private String str = "Hello World!"; // 1 STORE

    // Default Constructor
    // 1 LOAD (this) , 1 METHOD CALL (constructor of super class java.lang.Object)

    public void methodX(String msg) throws Exception
    {
        int a = 5; // 1 LOAD constant of 5, 1 STORE into local variable a
        methodY(); // 1 INTERNAL method call, 1 LOAD: this object

        // BRANCH count = 2, one for the if and another for &&
        // 4 LOADS: a, this, index, and bField
        if ((a > index) && (bField == false)) return;

        // BRANCH count = 2, default is not counted, 1 per each case
        switch(index) // 2 LOADS: this , index
        {
            case 0: a++; //_INCREMENT operation
            case 1: a--; //_INCREMENT operation with -1
            default: a--; //_INCREMENT operation with -1
        }

        // BRANCH COUNT = 1, TYPE Construction Count = 1 (new Exception)
        // THROW Count = 1, 2 LOADS: args array, constant value 3
        if (msg == null) throw new Exception("Something odd");
    }

    public void methodY()
    {
        // 2 LOAD constants, 1 STORE, 1 INCREMENT OPERATION, 1 BRANCH (for loop condition )
        for (int j=0; j<3; j++)
        {
            // 3 LOADS: this, System.out and str
            System.out.println(str); // 1 EXTERNAL method call
        } // 1 GOTO count (to continue loop)
    }
}

Listing D.1: Metric Count Example Code Listing
Appendix E

Growth Dynamics Data Files

The data referred to in the study of the distribution of the growth (Chapter 5) is described in Table E.1, and located in the directory data/growth on the DVD. The data is also available online at: http://www.ict.swin.edu.au/personal/rvasa/thesis/data

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc-stat.txt</td>
<td>Contains the descriptive statistics for the different metrics analyzed. Data is in CSV format.</td>
</tr>
<tr>
<td>norm-test.txt</td>
<td>Contains the results of the test for normality for the different metrics analyzed.</td>
</tr>
<tr>
<td>gini.txt</td>
<td>Contains the Gini coefficient values (CSV format) for the different metrics analyzed.</td>
</tr>
<tr>
<td>correl.txt</td>
<td>The correlation coefficient values determined between the various metrics as described in Section 5.4.1.</td>
</tr>
<tr>
<td>gini-trend.log</td>
<td>Log of the statistical analysis undertaken in order to establish the trend in Gini coefficients. The analysis is outlined in Section 5.3.6, and the observations are described in Section 5.4.7.</td>
</tr>
</tbody>
</table>

Table E.1: Data files used in the study of Growth (Chapter 5)

The raw metric data is provided as a comma separated values (CSV) file, and the first line of the CSV file contains the header. A detailed output of the statistical analysis undertaken is provided as log files generated directly from Stata (statistical analysis software) [259].
Appendix F

Change Dynamics Data Files

The data referred to in the study of change (Chapter 6) is described in Table F.1, and located in the directory data/change on the DVD. The data is also available online at:

The raw metric data is provided as a comma separated values (CSV) file, and the first line of the file contains the header. A detailed output of the statistical analysis undertaken is provided as log files generated directly from Stata (a statistical analysis software) [259].

The data table (Table F.1) is on the next page.
<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>change-prop1.txt</td>
<td>Contains the raw data (CSV format) used to established the change property captured by Equation 6.2.9.</td>
</tr>
<tr>
<td>change-prop2.txt</td>
<td>Contains the raw data (CSV format) used to established the change property captured by Equation 6.2.10.</td>
</tr>
<tr>
<td>kwallis.log</td>
<td>Log file of the Kruskal-Wallis test described in Section 6.2.1.</td>
</tr>
<tr>
<td>logit-change.log</td>
<td>Log file of the Logistic analysis described in Section 6.2.1.</td>
</tr>
<tr>
<td>mod-classes.txt</td>
<td>Modification data (CSV format) used to determine the properties related to size and frequency of change. This data supports the observations outlined in Section 6.2.2, Section 6.2.3, Section 6.2.4, Section 6.2.5, and Section 6.2.6.</td>
</tr>
<tr>
<td>mc-schema.txt</td>
<td>Schema that describes the data in modclasses.txt file.</td>
</tr>
<tr>
<td>logit-idc.log</td>
<td>Log file that contains the complete results of the Logistic analysis described in Section 6.2.5. This file was generated by STATA.</td>
</tr>
<tr>
<td>logit-nob.log</td>
<td>Log file that contain the complete results of the Logistic analysis described in Section 6.2.7. This file was generated by STATA.</td>
</tr>
</tbody>
</table>

**Table F.1:** Data files used in the study of Change (Chapter 6)
References


References


References


References


ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE ’03), Darmstadt, Germany, July 2003.


