Measuring Software Complexity Differences with the Analytical Hierarchy Process

by Noah Guilbault

B.S. in Information Technology, May 2006, University of Central Florida
M.S. in Digital Forensics, May 2012, University of Central Florida

A Praxis submitted to

The Faculty of
The School of Engineering and Applied Science
of The George Washington University
in partial satisfaction of the requirements
for the degree of Doctor of Engineering

January 19, 2018

Praxis directed by

Alan K. Gideon
Professorial Lecturer of Engineering Management and Systems Engineering

Gina Guillaume-Joseph
Professorial Lecturer of Engineering Management and Systems Engineering

The School of Engineering and Applied Science of The George Washington University certifies that Noah Guilbault has passed the Final Examination for the degree of Doctor of Engineering as of December 15, 2017. This is the final and approved form of the Praxis.

Measuring Software Complexity Differences with the Analytical Hierarchy Process

Noah Guilbault

Praxis Research Committee:

Alan K. Gideon, Professorial Lecturer of Engineering Management and Systems Engineering, Praxis Co-Director

Gina Guillaume-Joseph, Professorial Lecturer of Engineering Management and Systems Engineering, Praxis Co-Director

E. Lile Murphree, Professor Emeritus of Engineering Management and Systems Engineering, Committee Member

Amir Etemadi, Assistant Professor of Engineering and Applied Science, Committee Member

Richard Sparshatt. C2IE Systems Engineering Lead, Research Innovations, Committee Member

© Copyright 2018 by Noah Guilbault
All rights reserved

Dedication

This praxis is dedicated to Keri and Alex. You both inspire me to reach for greater heights. Thank you for all of the many years of understanding and support.

Acknowledgements

I would like to sincerely thank my praxis directors and committee members for sharing their time and expertise. I could not have completed this research without their help and guidance. I would like to thank Dr. Alan Gideon and Dr. Gina Guillaume-Joseph for serving as my praxis directors. I would also like to thank my committee members: Dr. E. Lile Murphree, Dr. Amir Etemadi, and Richard Sparshatt for their expertise. Finally, I would like to thank my beleaguered editor, Nancy A. Rosenbaum, for her hard work and expertise.

Abstract of Praxis

Measuring Software Complexity Differences with the Analytical Hierarchy Process

Software systems can become overly complex throughout the software development process, leading to increased error rates and failures. This praxis improves on the measurement of complexity, the first step in resolving this problem. Most software development technologies allow for incorporation of ever more interactive, error-prone, and interwoven designs that can introduce problems and increase the complexity of a software baseline. Because software typically holds latent defects, increasing the complexity of the software source code creates new adverse risks that undermine the usability, understandability, and reliability of software products. Despite following best practices during the software engineering process, the introduction of change to the software baseline can unnecessarily increase the complexity of software source code.

Software metrics are critical to fault prediction models that seek to improve the quality of software products by predicting the location of bugs in the source code. Measuring software complexity assists software developers by providing insights into projects by identifying areas of source code baseline that have potential issues with reliability, readability, understandability, and implementation. However, producing a set of software measurement data is not sufficient for making a reliable decision. Interpretation of the results of complexity measurements can be problematic because an accurate understanding of the numerical values that characterize specific software attributes such as coupling, reliability, and overriding can be difficult.

This praxis develops a technique for identifying the parts of a software baseline that experience the most complex changes between successive versions. This objective is

accomplished using a software complexity measurement technique that combines multiple aspects of the source code to quantify the variations in complexity of a software baseline between succeeding revisions. Building on prior successful efforts, this praxis develops a model for locating the portions of a software baseline that have undergone the most complex changes. The analytical hierarchy process (AHP) is used to combine the complexity metrics of number of metrics (NOM), number of children (NOC), coupling between objects (CBO), level of cohesion metrics (LCOM), weighted method complexity (WMC), and number of methods overridden (NMO) to determine the most complexly changed files for four open-sourced projects from the Apache Commons family of projects including IO, Net, Compression, and Discovery.

This praxis observed that the AHP process can be reliably applied to the set of six metrics used for the projects evaluated and produced AHP consistency ratings within the AHP process recommended ranges. Aggregate complexity values were successfully calculated for each source code file within the projects, and the most complexly modified files were identified. Several threats to the validity of the findings of this empirical study prompt the suggestion that the conclusions of this praxis should be treated as indications toward a pragmatic remedy to the problem of software complexity measurement between software versions rather than a final conclusive technique. To improve the reliability and the generalizability of this study, similar empirical studies should be conducted, using relatively large commercial systems developed in other programming languages such as C++, as well as selected from different domains.

Keywords: software metrics, complexity measurement, incremental changes, regression testing, AHP.

Table of Contents

Dedication iv

Abstract of Praxis v

Table of Contents viii

List of Figures x

List of Tables xi

Chapter 1 – Introduction 1

1.1 Regression Testing 5

1.2 Purpose of this Research 7

1.3 Research Questions and Hypotheses 9

Chapter 2 – Literature Review 11

2.1 Defining Complexity 13

2.2 The Analytic Hierarchy Process 20

2.3 Complexity Models 24

2.4 Complexity in Incremental Software Change 32

2.5 Using Analytical Hierarchy Process with a Programmatic Model 36

2.5.1 Number of Methods Metric (NOM) 39

2.5.2 Number of Children Metric (NOC) 40

2.5.3 Coupling Between Objects Metric (CBO) 40

2.5.4 Lack of Cohesion of Methods Metric (LCOM) 41

2.5.5 Weighted Methods per Class Metric (WMC) 42

2.5.6 Number of Methods Overridden Metric (NMO) 43

2.6 Principles of Measurement & Metrics 44

2.7 Critical Gaps in the Literature 49

2.8 Conclusion 51

Chapter 3 – Methodology 53
3.1 Research Questions and Hypotheses 55
3.2 Sources of Data 55
3.3 Calculating Version Differences 59
3.4 The Analytical Hierarchy Process Model 61
3.4.1 AHP Phase 1: Define Evaluation Quality Attributes/Factors 62
3.4.2 AHP Phase 2: Assign Factor Metrics 63
3.4.3 AHP Phase 3: Determine Factor Weighting 64
3.4.4 AHP Phase 4: Identify Alternatives 69
3.4.5 AHP Phase 5: Determine Alternative Weights 70
3.4.6 AHP Phase 6: Calculate Final Values 73
3.5 Efficiency of the Method 74
Chapter 4 – Results 76
4.1 Representative Individual Results 76
4.2 Aggregate Project Results 83
4.3 Project Calculated Complexity Differential Values 86
4.4 Results Discussion 89
4.5 Threats to Validity 96
Chapter 5 — Conclusion 99
Chapter 6 – Future Research 102
References 105
Appendix – Complete Results 126

List of Figures

Figure 3-1: The standard AHP compared to Differential Complexity AHP. 62

Figure 3-2: AHP differential complexity decomposition. 63

Figure 3-3: AHP differential complexity metric coupling.....64

Figure 3-4: AHP differential complexity alternatives identification. 69

Figure 4-1: Apache Net project NOM metric V4 values per range band. 78

Figure 4-2: Differential complexity scores for selected Apache Commons projects. 91

List of Tables

Table 2-1: Software Complexity Factors and Corresponding Metrics 39

Table 3-1: Data Sources, Revisions, and Average Project Size 57

Table 3-2: Project 1-4 Version Heuristics 57

Table 3-3: Saaty's Rating Scale for the Analytical Hierarchy Process 65

Table 3-4: Metric Pair-Wise Reciprocal Matrix and Eigenvector Weight Values 66

Table 3-5: Saaty's Consistency Index..... 68

Table 4-1: Calculated Metric Data for org.apache.commons.net.ftp.FTPClient 77

Table 4-2: Subsection of Reciprocal Matrix for NOM Metric 79

Table 4-3: Subset of NOM *n*th Root, Eigenvector, and Judgment Values 79

Table 4-4: Sample Class Eigenvector Weight values for Identified Metrics..... 81

Table 4-5: Calculated CR, CI, and λmax Values for Selected Apache Commons Net Project 81

Table 4-6: Metric Pair-wise Reciprocal Matrix, 6th Root and Eigenvector Weight Values 82

Table 4-7: Calculated Software Difference Based on Complexity Values for Sample Classes..... 83

Table 4-8: Project 1-4 Version Heuristics 83

Table 4-9: NOM Metric Outliers Count, Range Band, and Consistency Values for Project 1-4 85

Table 4-10: NOC Metric Outliners Count, Range Band, and Consistency Values for Project 1-4 85

Table 4-11: CBO Metric Outliers Count, Range Band, and Consistency Values for Project 1-4 85

Table 4-12: LCOM Metric Outliers Count, Range Band, and Consistency Values for Project 1-4 86

Table 4-13: WMC Metric Outliers Count, Range Band, and Consistency Values for Project 1-4 86

Table 4-14: NMO Metric Outliers Count, Range Band, and Consistency Values for Project 1-4 86

Table 4-15: Discovery (P_1) Most and Least Differentially Complex Files, with Outliers 87

Table 4-16: Net (P_2) Most and Least Differentially Complex Files, with Outliers 88

Table 4-17: IO (P_3) Most and Least Differentially Complex Files, with Outliers 88

Table 4-18: Compression (P_4) Most and Least Differentially Complex Files, with Outliers 88

Table 4-19: Differential Complexity Ratio 92

Table A-1: Apache Commons Discovery Complete Results 126

Table A-2: Apache Commons Net Complete Results 127

Table A-3: Apache Commons IO Complete Results 134

Table A-4: Apache Commons Compression Complete Results 140

Chapter 1 – Introduction

The software lifecycle includes phases of initiation, evolution, maturity, and phase-out. This lifecycle can be structured into distinct phases based on the types of activities that occur during the development process. Each phase of the software development process can be further decomposed into distinct, yet detailed activities (Pham, 2007). This approach to organizing the software development process makes the process easier to understand and increases efficiency for allocating resources to each phase.

As innovative and sophisticated computer systems are integrated into more daily business, governance, and social processes, there is an increased need to assess and guarantee the quality of software products to ensure they operate correctly and satisfy the expected functionality. Software testing is one way of achieving this goal before releasing the software to consumers. *Software testing* refers to the systematized execution of a software code or the entire system using controlled input to assess the quality, behavior, and potential problems of the software associated with its functioning (Munson & Elbaum, 1998). In the context of guaranteeing the quality and usability of software systems, it is reasonable to assume that more tests performed from a high-quality and multifaceted suite of software tests translate into higher confidence in the quality of the software product (Tahvili, Saadatmand, & Bohlin, 2015).

Understanding, measuring, and predicting the effect of changes to a software system become more challenging and demanding as systems evolve, making it important for software engineers, developers, and quality estimators to quantitatively describe the difference between each software system and its predecessor or successor (Munson &

Elbaum, 1998). When a software system is developed and released for the first time, it can be difficult to identify the location of flaws in the code due to the lack of sufficient actionable information. The absence of such information makes relative complexity measures a major fault surrogate, which suggests that software modules with a high level of complexity are more likely to harbor many hidden faults (Munson & Elbaum, 1998). This use of *complexity* refers to the extent to which a system or software source code involves a design or implementation that is difficult to verify and understand (Tashtoush, Al-Maolegi, & Arkok, 2014).

Faults are often identified as a software system progresses either iteratively or incrementally through sequential versions or builds, necessitating code revision to eliminate discovered flaws. The process of eliminating identified flaws requires changes to the software baseline through the addition of new code, but this new code can be as fault-prone as the initial code. Munson and Elbaum (1998) asserted software faults are routinely injected into the code throughout the evolutionary development process of the software product.

Furthermore, additional changes to a software baseline represent improvements in response to the evolution of system and user requirements. These incremental code changes also introduce more faults to the software baseline. These types of incremental changes frequently cause the comparative complexity of every module affected to increase as the system grows and evolves through versions or builds. In theory, the rate of change of the comparative complexity of a software baseline can be conceptualized and leveraged as an ideal measure of the rate of fault introduction into a software baseline.

Currently, it is difficult for a software engineering manager to determine when complex changes are being applied to a software baseline. This difficulty can cause system failure due to the inadequate testing of software components undergoing complex changes (Kaushik & Tyagi, 2016; Pravin & Srinivasan, 2013). Software testing is intended to facilitate the detection and correction of faults based on the premise that the rate of introducing faults to a software baseline should be less than the rate at which the faults are removed. This construct might be realistic, but the actual number of software flaws might increase in substantive software change cycles. Therefore, it cannot be asserted that software tests can monotonically reduce the total number of flaws in a software system. As Hassan (2009) observed, the complex process of changing software code reduces the quality of the change process—the software system itself.

The subject of software metrics has gained attention over the past few decades due to a basic interplay of fundamental dynamics. Engineering managers, software developers, and testers need the ability to measure and manage the process of developing software products quantitatively. This need is driven by requirements to estimate the resources and constraints associated with producing and delivering a software product, as well as the suitability and readiness of the software product for release. Inspiration for the development and study of software metrics stems from researchers' and scholars' interest in identifying software complexity metrics to quantitatively measure and define software attributes as a means of developing a better understanding of software engineering (Fenton & Pfleeger, 1998). The concept of software complexity metrics is a dynamic one, and this fluidity frequently undermines the development of a standard set of software metrics. Existing standards are being reviewed and criticized as new metrics are

proposed. Even though the models of software complexity metrics are understandable, coherent, and well defined, improving the complexity of a software baseline remains a challenge for software developers. It is preferable to build less complex software projects and products with desirable quality attributes to fulfill user requirements (Fernando, Wijayarathne, Fernando, Mendis, & Guruge, 2014).

A different type of remedy is needed to counteract the trend in software engineering that progressively limits the effectiveness and completeness of conventional quality assurance and software testing processes (Hallem, Park, & Engler, 2004). Some of the contributing trends are due to the increase in software complexity, expanded product functionality, and the comprehensive nature of supported configuration (Sohaney, 2015). Additionally, increased margin and business pressure dictate that software development projects should be completed with a nod to time and resource constraints. Although software developers might strive to acquiesce to these demands, complexities in the software development lifecycle can increase the costs of software product development and maintenance (Banker, Datar, Kemerer, & Zweig, 1993; Debbarma, Debbarma, Debbarma, Chakma, & Jamatia, 2013). Conveniently, quality metrics provide a critical insight into the progress of software development and testing efforts, thus fostering better control and informed decision making.

Complexity is an inherent attribute of the software lifecycle, from requirements engineering through implementation. Ordinarily, complexity is not desirable in software products because of its tendency to make software codes more difficult to read, understand, and change. In fact, code complexity of a software program or system might be a primary cause of defects, a construct that represents complexity as an inhibitor of

quality in software products (Madi, Zein, & Kadry, 2013). The increased complexity in both the process and the product gives rise to hazards that affect reliability and usability. Software testing is conducted in the hope of discovering software defects, but as the complexity of software systems increases, the testing effort reaches a point of diminishing returns (Zelkowitz, 2003).

Software reliability affects system reliability (Pan, 1999). Similarly, overly complex software code is a major source of software defects. Various techniques have been developed to measure software complexity and to help identify overly complex software source code as a first step to simplifying code and thereby reducing complexity, improving testability, and reducing software defects (Gonzalez, 1995; Tegarden, Sheetz, & Monarchi, 1992a; Zuse, 1994). Unfortunately, measurement in software is as controversial as it is complex; no reliable quantitative methods of software measurement have been developed and tested to represent software reliability without the encumbrances of significant limitations. While numerous quantitative approaches using measures of complexity could be used to enhance the reliability of software products, balancing budget against development time remains problematic and difficult. The software development industry instinctively uses the regression testing process as its foundation of ensuring quality and software reliability.

1.1 Regression Testing

Even the smallest change or modification to existing software source code can have unexpected consequences, and changes can be required due to a variety of reasons including bug fixes, new functionality, functionality improvements, integration, the introduction of new interfaces, and security patches. The purpose of regression testing is

to identify bugs that might be accidentally introduced into the software code when modifications or changes are made (Malishevsky, Rothermel, & Erlbaum, 2002). Regression testing involves the formal testing of current software application functionality to ensure that changes made to the software source code do not adversely affect existing functionalities. Regression testing typically divides the testing process into many separate test cases, each of which tests a specific portion of the functionality of an application. The regression testing process is performed at the end of a software development cycle to identify bugs or flaws that have been unintentionally introduced during the development of a new release, build, or patch. Performing the test case scenarios after known problems were fixed allows software testers to ensure changes to the application do not lead to a regression. A regression is said to have occurred when previously functional components or modules fail.

Regression testing is used to improve the quality of software systems that have undergone modification and changes (Malishevsky et al., 2002). Unfortunately, it is frequently the case on large software development efforts that the regression test process is mismanaged and ineffective (Butler et al., 2004). Frequently, these issues stem from resource constraints (Delamaro, 2014; Narciso, Delamaro, & Nunes, 2014). Complete software regression testing can be time-consuming because the tests must be conducted whenever changes are introduced to the code. Large software systems rarely undergo an entire software regression test because of budget and time constraints. Given the burden of costs and time associated with comprehensive software regression testing, the proper selection and prioritization of test cases increases the probability of detecting flaws and defects (Narciso et al., 2014; Pravin & Srinivasan, 2013; Vinothkumar & Galeebathullah,

2015). It is common for limited personnel, budget, and time resources to be allocated for software testing activities. This allocation emphasizes the importance of having available test cases and a robust test case selection mechanism. Furthermore, this test case selection mechanism should identify a subset of the test cases that maximizes the available resources.

To date, many regression testing test case prioritization models rank test cases by scheduling the execution order for the test cases with the greatest value according to a predefined criterion. This technique offers benefits such as earlier detection of faults and the provision of earlier feedback to software testers and test case designers. Regression test selection models are designed to reduce the costs of software tests by selecting and prioritizing only those test cases that are essential in testing a modified program (Delamaro, 2014; Karlsson & Ryan, 1997; Vinothkumar & Galeebathullah, 2015). Proposed regression testing models and software maintenance techniques include the test case minimization model, the test case selection model, and test case prioritization model. Empirical studies have been conducted to evaluate the cost efficiency of various approaches to regression testing (Do, Mirrarab, Tahvildari, & Rothermel, 2010; Elbaum, Malishevsku, & Rothermel, 2002; Malishevsky et al., 2002). Despite efforts to develop cost-efficient regression test models, no single model stands out as most efficient in terms of costs and time, a challenge that creates the potential for large cost savings through the selection of cost-effective techniques for every software version (Tahvili et al., 2015).

1.2 Purpose of this Research

The full regression test process is a critical step in formally verifying and validating large- or enterprise-scale software development projects. Full regression testing may

involve running many test cases using automated or manual processes to ensure the software being tested functions according to specifications. Conducting frequent regression test processes during the software development process ensures that defects are not introduced into previously working software components. The execution of regression tests is effective in reducing the number of defects identified at the end of formal software development (Bannerman & Martin, 2011; Kaushik & Tyagi, 2016; Pravin & Srinivasan, 2013). However, the constraints of time and cost make conducting repeated full regression testing on source code impractical (Pravin & Srinivasan, 2013). This constraint implies a substitute, such as automated complexity measurement, can be a useful, less expansive testing augmentation.

A high level of software complexity in software source code is correlated with the number of defects in a software product (Chhillar & Bhasin, 2011; Zuse, 1994), making it difficult to choose an appropriate set of complexity measurement metrics to serve as effective indicators of software defects (Shah & Morisio, 2012). The field of software complexity measurement is focused on identifying valuable metrics of software complexity so that complex source code can be simplified to improve both testability and the likelihood of software defects (Zuse, 1994).

The attributes and elements that make a piece of a software program difficult to understand are responsible for the resulting complexity of the software. There is a need to develop robust software measurements to mitigate the impact of software complexity while promoting optimal quality. The thesis of this praxis is that a quantifiable measurement is needed to determine the complexity of an incremental software change, and that this goal might be achieved by developing a software complexity model that

takes into account the complexity of the initial revision, the complexity of the subsequent revision, and the complexity of the incremental changes required to achieve the subsequent revision. This research involves developing a technique to identify which portions of a software baseline have undergone the most complex changes. This information is used by engineering managers and their software test teams as a key criterion in a test case selection model that prioritizes a set of test cases based on the complexity of changes made in the most recent updates to the source code of a software application.

This use of metrics presents a challenge because conventional measures of software complexity only focus on the complexity of a source code at a single point in time (Canfora, Cerulo, Cimitile, & Di Penta, 2014; Harrison, 1992). Complexity measurements on the same source can branch at different points in time; as such, the measurements might not necessarily mirror the aggregate complexity of the incremental software changes that are introduced in the software baseline (Canfora et al., 2014). Furthermore, even though software source code can cross several different complexity characterizations simultaneously, most conventional complexity measurement techniques are designed to assess a single or just a few aspects of complexity (Gonzalez, 1995; Tegarden et al., 1992a). These limitations call for a more quantifiable measurement approach to determining the complexity of incremental software changes.

1.3 Research Questions and Hypotheses

This praxis addresses three main research questions:

RQ1. What software complexity measurement techniques are best suited to measure the software source code changes performed between subsequent software revisions?

RQ2. How can an accurate measure of the complexity of the software source code changes performed between subsequent software revisions be developed?

RQ3. How can software changes with a large magnitude of measured complexity be identified in a software baseline?

The following hypotheses are evaluated to answer the primary research questions:

H1. The complexity of a software change can be expressed as a function of the complexity of multiple aspects of the software source code.

H2. The complexity of a software change can be expressed as a function of the complexity of the software source code at an initial point and the complexity of the subsequent sets of changes applied to the initial software source code.

H3. Significantly complex changes in a software baseline will differ statistically with respect to their software difference complexity values throughout a software development effort.

Chapter 2 – Literature Review

The field of software metrics and complexity measurements is relatively large. Following Halstead's (1977) seminal work, interest in research involving software complexity or quality metrics increased in response to recognition of these attributes as critical to effective management of the software process. Software metrics facilitate the use of a solid engineering approach to software product development by providing both quantitative and objective frameworks that were previously lacking in the field of software engineering (Riguzzi, 1996). Over time, the use of software metrics has become more widespread and mainstream research in software metrics has evolved. While initial research efforts focused on invention of new metrics applicable to different software attributes without much concern for the systematic and practical validity of the metrics (Riguzzi, 1996, p. 3), recent efforts have targeted the application of models of measurement and assessment to software metrics as well as how to ensure validity.

Software quality is multidimensional and holds great importance to the acceptability and usability of software products (Desai, 2014). Over time, researchers have underscored the need for the attributes of quality to fit into a well-defined framework of metrics to help software to achieve the expected parameters of quality assessments. Recent intensified research efforts on software complexity metrics and the significance of these attributes relative to assessing the overall quality of software have led to considerable literature on software metrics. In particular, there is a vast array of literature on the topic of measuring the complexity of software components and products.

Literature germane to this praxis is introduced in individual sections. First are sections that includes the definitions of the key concepts of the praxis, including the definition of complexity and an overview of the analytical hierarchy process. Following these are sections on software complexity modeling techniques and the application of these models to measuring the complexity of incremental software changes. Next is the use of the Analytical Hierarchy Process with complexity measurement techniques. The final section covers the literature on the application of measurement unification methodologies which seek to develop an optimal function that considers complexity metrics from multiple disparate complexity measurement techniques. This forms a relationship between quantifiably identifying when complex software changes are being applied to a software development baseline and particular software testing methodologies. Although the breadth of knowledge for each of these individual research areas is vast, relatively few directly applicable studies are available that offer guidance to an engineering manager seeking to monitor the complexity of changes being applied to a software development baseline.

The academic field of software complexity measurement is founded on improving the quality of software by being able to measure and thereby reduce the complexity of software source code (McCall, Richards, & Walters, 1977; Zuse, 1991). Complex software source code has been repeatedly and highly correlated to software defect rates (McCall et al., 1997). Numerous scholarly works have reported on regression test cases for optimal software defect identification. These test case identification techniques were based on heuristic schemes or statistical analysis of historical defect rates; they did not analyze the impacts of complex changes (Pravin & Srinivasan, 2013). Because

complexity is an intrinsic attribute of software products and modules, studies of complexity are central to the field of software engineering. This review of literature on software complexity measurements and metrics illuminates the following themes: theories of measurement and metrics, definition of software complexity, complexity in incremental software change, software complexity models, and critical gaps in the literature.

2.1 Defining Complexity

The term *complexity* merits attention to ensure understanding of its conceptualization and application. For many authors, complexity is about control flow complexity, which is an outcome of the internal attributes of a software product (Riguzzi, 1996). Other authors perceive complexity as a high-level term that encompasses almost all attributes of the software. For instance, Fenton and Kitchenham (1991) identified three kinds of complexities: psychological, representational, and computational. Psychological complexity is the most commonly studied type of complexity in the field of software engineering.

According to Riguzzi (1996), psychological complexity incorporates problem complexity, structural complexity, and programmer characteristics. Although the research on problem complexity is relatively young, scholars generally agree that programmer characteristics are difficult to measure, both quantitatively and objectively. In contrast, structural complexity has been studied expansively over time because it is the only aspect of psychological complexity that can be objectively and quantitatively assessed (Riguzzi, 1996). Because structural complexity has been heavily studied, many metrics and

measures have been developed and suggested for adoption; these metrics measure an array of internal attributes of software products.

A review of the literature suggests metrics that measure structural complexity can be categorized into two broad groups—module metrics and intermodule metrics. As the name implies, module metrics are used at the module level, such as the class and subprogram levels. This group of metrics includes data structure metrics, size metrics, style metrics, cohesion metrics, and metrics to measure control flow complexity. Intermodule metrics are used to assess the interconnections between distinct system modules and are commonly represented by coupling metrics. Scholars' interest in structural complexity prompted further research to ensure the validity of the metrics used in assessing the structural complexity of systems. For instance, Weyuker (1988) suggested a set of axioms against which any metrics for measuring structural complexity must be validated. These axioms would provide a mechanism for comparing and assessing newly developed metrics.

Weyuker (1988) emphasized that a measure must not be excessively broad because a measure that rates all software modules and programs as equally complex is not a good measure. Instead, Weyuker suggested there must be sufficient resolution and flexibility in the measurement scale to allow the metric to be useful in assessing the complexity of software products. Likewise, a measure of software complexity must not be too fine. Critics of Weyuker argued the fineness of a measure is not applicable to all metrics, especially if the binary representation of a program is considered as a metric. These first two axioms—that a measure be neither too broad nor too fine—are fundamental

properties that every measure ought to have; they do not outline anything particular about the measures of complexity (Riguzzi, 1996).

Another of Weyuker's (1988) axioms is that the functionality and complexity of a program are two distinct attributes. That is, the complexity of two programs depends on their implementation, even if both calculate the same function. Interaction increases program complexity, and the contribution of the program, module, or overall complexity of the system might depend on the rest of the modules or programs. Fenton and Kitchenham's (1991) criticisms of these axioms fostered a conducive environment for discussions on how to theoretically validate a metric.

Although researchers recognize the importance of measuring software complexity (Jiang, 2015), there is no universally defined and acceptable definition of software complexity. For instance, the IEEE conceptualized software complexity as the latitude by which a system, module, or component has an analysis, design, or implementation that is difficult to understand and the verification of which is difficult (Software complexity, 1990). For researchers such as Basili (1980), complexity refers to the number of resources used by a system during the interaction between system and a software module to perform a specific task. Basili posited that, if the programmer is perceived as the interacting system, complexity reflects the difficulty in performing major tasks such as writing the software code, debugging, verification, validation, or modifying the software.

Gonzalez's (1995) definition is predicated on Basili's (1980) and Shooman's (1983) conceptualization of software complexity. Gonzalez defined complexity as a measure of the resources expended in the development, testing, maintenance, debugging, correction, and operation of software products, as well as in user training. From this construct,

Gonzalez asserted complexity is a fundamental attribute of the software lifecycle and the degree of complexity in a system can be determined by measuring the properties of the program as a final product of the software development process (Shooman, 1983). Therefore, software complexity depends on many factors, including the number of objects, the relationships and interfaces between objects, the tools and methods used in the development process, the software environment, and project organization (Gonzalez, 1995).

Debbarma et al. (2013) defined software complexity as the measure of how difficult a program is to understand and use. Because there is no universally accepted definition of software complexity, Zuse (1991) concluded that complexity measure, as a term, is a misnomer. A review of the literature on software complexity measure reveals some discrepancies in notions of what constitutes software complexity. Software complexity gets mixed with the psychological complexity required to understand a problem. Most scholars have focused on complexity as a function of system design complexity, problem complexity, and procedural complexity, as measured by the inputs, outputs, structures, size, and references held by a particular component of software source code (Tegarden et al., 1992a). Other scholars investigated complexity in terms of defect rates, performance, and quality or value (Mukker, Singh, & Mishra, 2014; Pasupathy & Bhavani, 2013). This literature review is focused on the measurement of complexity in the classical interpretation using quantitative measures of source code attributes and relationships.

Zuse's (1991, 1994) seminal works document more than 90 different published techniques for evaluating software complexity. These measures of complexity range from the trivial measurements of source line of code (SLOC) to much more complex

measurement schemes such as Halstead complexity, cyclometric complexity, and function point analysis (Tashtoush et al., 2014). Many of these techniques are effective and complementary and have shown a high correlation in results when executed on identical datasets. The Halstead and cyclometric complexity measure methods are highly correlated with each other, despite their different measurement techniques. Other approaches, such as one involving object-oriented complexity measurement, are even more effective at capturing the complexity of object-oriented systems by accounting for the specifics of the programming technique (Tegarden et al., 1992b).

Although every software complexity analysis technique must account for programming language syntax to analyze a software system, a set of techniques is commonly established. Some of these complexity analysis techniques are appropriate for specific software development languages or styles, while others are more generic and universal. The object-oriented analysis technique accounts for the nature and inherent complexity of the object-oriented system itself. In contrast, a Halstead or cyclometric function does not take into account the object-oriented development scheme and is more generalized (Tashtoush et al., 2014; Tegarden et al., 1992b). It can be challenging to identify the best approach—specific or general—to apply to novel complexity analysis schemes such as leveraging fuzzy logic or entropy measurement techniques (Prasanth & Ramasubramanian, 2009). Canfora et al. (2014) modeled one such entropy measurement technique as a model of the extent of dispersed changes across a software baseline. In this model, software changes that affected many files would have a higher entropy measurement than changes made to a single file. This novel complexity measurement scheme is identified because it does not directly consider the logical constructs and

relationships within the source code of the software; instead, it uses statistical patterning techniques relating to where and when changes are made, but not the specific nature of the changes themselves (Canfora et al., 2014).

This literature review and praxis focuses on the more established techniques of software complexity measurement as covered in section 2.4. Nonetheless, myriad definitions and scoping of software complexity converge on the characterization of software complexity as the measure of the difficulty to maintain, modify, and understand software (Zuse, 1991). There is general agreement among scholars and researchers that software complexity has an impact on the costs of software acquisition—from development to maintenance activities (Banker et al., 1993). Early scholars of software complexity such as McCabe (1976) and Halstead (1977) established the foundation for those who followed, including Jiang (2015). Their proposed complexity metrics provide the basis for quantifying both the overall complexity of software components and the evolution of complexity throughout the lifecycle.

There is agreement in the literature that complexity is a critical attribute of software products because of its influence on other attributes such as modifiability, maintainability, understandability, cost, and testability (Riguzzi, 1996). Complexity measures allow estimates to be prepared for software development, testing, and quality assurance activities. Scholars agree that complexity measures help to determine the appropriateness of *refactoring*, which is the practice of making software architectural changes to better align the systems design with the current set of requirements. This practice is done to improve quality and prevent problems by simplifying the system. For many researchers, complexity is a measure of the correlation and interactions among

features or components of a software product. Therefore, software complexity can be broadly conceptualized as an indicator of the quality of software products and their associated cost, such that a complex code increases the risks of defects and the difficulty of adding extra functionality. Of greater concern is that software complexity complicates the process of validating, understanding, and maintaining software code.

This convergence explains researchers' interest in software complexity. Building on existing models and propositions, researchers have sought to describe complexity using a single number, an approach that Fenton (1994) criticized as being unrealistic and flawed. Fenton demonstrated that Weyuker's complexity model is flawed because it attempts to be a representative example of dissenting interpretations of complexity. Similarly, Durisic, Nilsson, Staron, and Hansson (2013) concluded that applying the software complexity and quality metrics after every major change not only enables testers and quality assessors to verify that defined quality attributes of the referent software system remain robust, but also helps to identify new testing procedures and areas. Consequently, the use of these metrics augments the quality attributes of the software products, such as reliability, maintainability, and stability, and reduces the long-term development and maintenance cost of software products.

There is no single measure for software complexity (Debbarma et al., 2013; Delange, 2015; Fenton, 1994; Riguzzi, 1996). Fenton (1994) demonstrated that a single real-valued complexity metric is unrealistic, even for a specific type of complexity such as control flow complexity. Based on this assertion, Weyuker's (1988) model is flawed because the properties of the model attempt to describe all the attributes of complexity using a single

metric. Zuse (1991) revealed Weyuker's axioms conflict within the context of the representational model of measurement.

Most of the foundational literature on the measurement of software complexity is focused on traditional software systems (Conte, Dunsmore, & Shen, 1986; Halstead, 1977; McCabe, 1976). More recent studies have focused on the measurement of software complexity for object-oriented systems. Some of these studies leveraged the traditional metrics for software complexity to object oriented systems (Tegarden et al., 1992b), some defined new metrics for object-oriented systems (Chidamber & Kemerer, 1991; Keyes, 1992), and others considered both approaches to software complexity measurement (Moreau & Dominick, 1989; 1990).

2.2 The Analytic Hierarchy Process

Research on testing the quality of software products has delved into the decision-making process for selecting quality attributes as well as the models used. This praxis leverages a multi-criteria decision-making (MCDM) approach to combine the variations in complexity measures between versions of software systems in the repository. This approach is based on mathematical frameworks and produces a weighted ranking of options or alternatives. There is an extensive body of research on the development of source codes metrics, but little consensus on the specific composition of metrics for any single programmatic language or technique.

MCDM encompasses a family of techniques that allow for the combination of qualitative judgments of the importance of the various software complexity metrics with the calculated quantitative values per metric between the different revisions of software source code files. The literature on software metrics proposes several methods, including

the weighted sum model (WSM) and the weighted product model (WPM). WSM is arguably the earliest and the most widely used method in the analytical process (Mateo, 2012). WPM was proposed as a modification to the WSM and to address some of its known weaknesses (Garmabaki, Ahmadi, & Ahmadi, 2015; Triantaphyllou & Mann, 1989). The analytical hierarchy process (AHP), a later development in the use of MCDM, has become a popular technique for integrating multiple criteria and attributes into the analytical process of decision making. Both AHP and WSM have been used for the evaluation and selection of software packages. However, AHP offers the advantage of systematically reducing the entire set of comparisons using a hierarchy structure, and the consistency of the responses is verified through a consistency ratio.

Proposed by Saaty (1972, 1977), the AHP model is a widely popular multi-criteria decision-making paradigm that is well suited to complex problems involving conflicting, subjective criteria. Jayaswal, Patton, and Forman (2007) described the AHP as an advanced model that supports complex decisions by evaluating choices involving multiple objectives, and quantifying intangible factors. The AHP model allows for the hierarchical structure of the factor or criterion, or subfactor/subcriterion, when allocating weights. AHP is a comprehensive, yet rational framework that provides a robust model for evaluating the relative worth among criteria, factors, or elements, especially when those elements are difficult to quantify (Jayaswal et al., 2007). The AHP model involves four main steps:

1. definition of the problem,
2. performing pair-wise comparisons,

3. computing the Eigenvector values of the relative importance of the referent criteria, and
4. checking for consistency.

AHP and its variants have been adopted to various decision situations. Apart from the management of software complexity, AHP has been extensively used in quality function deployment (QFD). Saaty (1972, 1977, 1980) explained AHP is a mathematical framework for structuring priority in a complex system, which can be functionally disintegrated into subsystems that are represented hierarchically. The topmost element is the goal, while the elements at every lower level are the criteria for the elements found at each next lower level. Saaty referred to the elements at the bottom level of the hierarchy as alternatives. Using this procedure, the AHP model structures the prioritization process by decomposing a multi-element complex software system into smaller constituent elements (Pham, 2007). Thus, process setting in the AHP model can be organized into three main phases of the structuring of the system, pair-wise comparison, and syntheses of priorities.

Many researchers acknowledge the importance of software engineers applying software testing techniques in the empirical context. Software verification and validation activities are conducted to improve the quality of software products throughout the software development life-cycle (SLDC). Researchers such as Jain, Behera, Mandal, and Mohapatra (2014) observed the success rate of a software system depends on the type of quality testing techniques employed during the verification and validation phase. Even though there are many methods for testing the quality of software products (e.g., white box techniques, black box techniques, and gray box techniques), choosing the right one is

a difficult task. Each method pursues a specific goal or objective; none are entirely appropriate for all types of software systems. To address this issue shortcoming, Jain et al. proposed a technique for selecting the methods for testing the quality of software systems through the AHP model. Their method considered several criteria such as the number of independent paths (NIP), new or modified system (NMS), the cost of requirements (COR), and the number of test cases (NTC).

Klindee and Prompoon (2015) asserted test case designers require time and effort when designing test case to cover all conditions and requirements defined in the test case structure. In light of this argument, Klindee and Prompoon proposed a model for identifying, storing, and retrieving test cases that are affected by changes to software requirements. Their model also defined a procedure for prioritizing the retrieved test cases through the AHP model to increase the efficiency of the ranking process. They concluded that the AHP-based model assists system testers in identifying test cases used for a comprehensive regression testing.

Kassab and Kilicay-Ergin (2015) conducted similar research on application of the analytical hierarchy process to prioritizing system quality requirements. They argued that conventional approaches to prioritizing quality attributes are qualitative by design and do not address the tradeoffs among design models. To address this gap, Kassab and Kilicay-Ergin proposed an AHP technique to quantitatively prioritize design decisions and paradigms without compromising the interrelationships between design principles and system quality requirements. They argued that this approach facilitates an objective prioritization of design principles and methods and reduces discrepancies between technical and business stakeholder assessments.

2.3 Complexity Models

Various measures of software complexity have been suggested over the past 20 years to better regulate and understand the products and practices of software development. Frequently cited measures of software complexity include the cyclomatic number metrics, which measure independent code paths (McCabe, 1976), programming effort (Halstead, 1977), cognitive functional complexity (Y. Wang & Shao, 2003), KLCID complexity metric (Klemola & Rilling, 2003), and the number of program statements (Weyuker, 1988). Banker and Slaughter (2000) measured the general data complexity of a software module using the number of data elements contained in every unit of application functionality. Tait and Vessey (1988) used three main variables to determine the software complexity measure: the complexity of processing, the difficulty in defining the information requirements of a software module, and the general complexity of the design. The most common software complexity measures that take into account expended time, memory, and effort borrows heavily from Halstead's (1977) software complexity metric.

Many diverse techniques of software complexity measurement are available and in use, as reflected in the depth and breadth of literature on the subject. Following the early of research by Halstead (1977) and others, subsequent research focused on the creation of unified metrics of software complexity as a way of measuring productivity, quality, and value in terms of software complexity (Gonzalez, 1995). Basili (1980) suggested attributes such as data structures, the flow of control, data flow, and program size could affect maintenance. The literature reveals many models have been developed to assess these attributes, and several hybrid models have been developed, notwithstanding the

inherent complexity of software systems. For example, Y. Wang and Shao (2003) assessed the cognitive weights of fundamental structures for software control. Their cognitive functional weights model led to cross-examination of the analysis of software size, complexity, and the ability to conceptualize and understand critical attributes during the design, implementation, and maintenance phases of an SDLC.

Jabbar and Sarala (2011) conceptualized software complexity metrics as the failure caused by anomalies or deviations in the source code. They argued that software complexity measurements tend to estimate the number of lines of code in the problem-solving task, even though there is a high interrelationship between the length of code in a software module and the structural complexity of the entire system. Canfora et al. (2014) acknowledged that software changes due to refactoring, fixing bugs, and the addition of new features might either reduce or increase the complexity and disorganization of a software source code baseline. Based on this proposition, they conducted an analytical evaluation of how the complexity and disorganization of source code, using a concept of software source code entropy, related to four variables. These variables include the number of programmers working on the source code, the application of software design patterns, the presence of software refactoring procedures, and the types of different changes being applied to the system.

Apart from the studies by Jabbar and Sarala (2011) and Canfora et al. (2014), the literature includes engineering and scientific studies by Buse and Weimer (2010), Dhillon and Singh (2012), and Honglei, Wei, and Yanan (2009), all of whom proposed several complexity metrics. Most of this literature acknowledges the importance of the predictive capability of measurement metrics and the need to integrate them into software testing

(Sagarna & Lozano, 2008). Complexity metrics are key to the computational assessment of software complexity and development of a reliability model (Munson & Khoshgoftaar, 1991). Many metrics have been proposed to measure software complexity. Some of the most common and influential metrics with respect to software design and code quality include line of code (LOC), Halstead complexity (HC), and cyclomatic complexity (CC).

The most popular complexity metrics can be broadly classified into two groups—traditional complexity metrics and object-oriented software complexity metrics. Traditional complexity measurement metrics are more commonly used and include McCabe's control flow complexity, Halstead's language complexity metric, Henry's interface complexity, Elshoff's data flow complexity, McClure's decisional complexity, and Chapin's data complexity. The following paragraphs introduce these traditional complexity measurement metrics.

McCabe's (1976, 1982) control flow complexity metrics are predicated upon a representation of a program in terms of control flow. McCabe used a graph to represent the flow of control throughout a program; each node in the program represents a processing task. The edges of the graph represent the characteristic control flow between distinct nodes (Babu, Ramakrishna, & Bharathi, 2013; McCabe, 1976, 1982). The model is based on the argument that lower values of control flow are better for measurement. McCabe proposed using 10 as the threshold value for the measurement of complexity, beyond which program modules should be organized into smaller units of measurement. McCabe used the total number of edges of the control flow graph as a measure of the control flow complexity of a software system.

Halstead (1977) proposed a language complexity model that uses the inputs of individual and total operands and operators to measure the volume, effort, and complexity of a piece of software code. This metric is broadly conceived as a method of determining the quantifiable or numerical measure of software complexity with a direct reference to the operands and operators in the software module (Babu et al., 2013; Jamali, 2006). The metric measures software complexity by aggregating the total number of operands and operators defined in a program. These measures are built upon four scalar numbers: n1 (number of distinct operators), n2 (number of distinct operands), N1 (the total number of distinct operators in the program), and N2 (the total number of a program's operands). These scalar numbers are derived from the source code of the referent program.

Halstead (1977) used these scalar numbers to develop five measures: (a) program length, denoted by N and represented mathematically as N = N1 + N2; (b) program vocabulary, denoted by n and represented mathematically as n= n1 + n2; (c) volume, denoted by V and represented as V = Nlog2(n); (d) difficulty, denoted by D and represented as D = (n1/2) * (N2/n2); and (e) effort, denoted by E and represented mathematically as E = D * V.

Other options include the fan-in/fan-out metric and Zuse's (1991) interface complexity metric, which measures software complexity as a function of both fan-ins and fan-outs. Zuse defined *fan-ins* as a sum of the numbers of local flows into a program procedure and the data structures that inform the procedure. The *fan-outs* were defined as the sum of the numbers of local flows out of a procedure and the data structures that are updated by the procedure. According to this model, complexity can be expressed as the

product of the procedure length and the square of the product of fan-in and fan-out. Fan-in can be conceptualized as the sum of the number of the procedures called by a program, the parameters read, and the global variables referenced. Fan-out can be conceptualized as the number of procedures that call some definite procedure, the parameters that are updated during that call, and the global variables to which the called procedures write. According to Zuse's interface, high values of fan-in and fan-out might suggest a software module that is complex in design and development and difficult to understand.

McClure's decisional complexity metric measures decisional complexity as the total number of control variables that a module references and the number of associations or comparisons in a module (Babu, Prasad, & Sudhakar, 2013). Complexity is mathematically expressed as the sum of comparisons (C) in a module and control variables (V) that are referenced in a module. This metric is comparable to McCabe's complexity metric, but with respect to control variables in a software module. Chapin's data complexity metric measures the amount of information required and used by a method based on the premise that a higher number of variables makes the software code more difficult to understand, update, and modify (Li & Cheung, 1987). The complexity measure is defined as $P + 2M + 3C + 0.5T$, where P represents the number of variables that a method uses to calculate the output, M represents the variables that the method creates or modifies, C represents the variables directly involved or referenced during the determination of control flow, and T represents the unused variables.

These traditional complexity metrics are largely applicable in determining software complexity at the method level (Babu et al., 2013). This limitation in scope has fostered the development and adoption of object-oriented (OO) complexity metrics that can be

used to examine source code as a basic indicator of software quality attributes. Chidamber and Kemerer's (1994) six metrics for measuring software complexity are widely acknowledged with respect to OO systems. They include the response for class (RFC), LCOM, number of children (NOC), weighted methods per class (MC), DIT, LOC, and coupling between objects (CBO).

The LOC measure of complexity is determined by counting the number of lines in a program code. This metric is used to estimate the size of a software program (Tashtoush et al., 2014), but is not sufficient for measuring software complexity (Mas y Parareda & Pizka, 2007). The DIT metric is used at the class level. DIT is conceptualized as a measure of the maximum distance between the base class or the root to a node. Although this distance is typically measured in terms of the sum of ancestor classes, it exhibits a high latent for the reuse of the inherited methods of the module (Babu et al., 2013). The RFC metric measures the total number of methods that are called in response to a local or remote procedure call; higher values of RFC suggest a greater complexity value of a class. The WMC counts the aggregate complexity of all methods that are localized to a class.

NOC is a complexity metric that defines the number of subordinate subclasses of a class (Fóthi, Nyéky-Gaizler, & Porkoláb, 2003), such that reuse increases as the value of NOC grows. For any given class, CBO is a metric that measures the number of collaborations or associations between two classes. The premise of this metric is that reuse decreases as collaboration increases. LCOM determines the granularity of the relationship between the instance variables that are localized to a class and the local methods. LCOM can be expressed as the difference between the number of empty

intersections and the number of non-empty intersections (Chidamber & Kemerer, 1994; Jamali, 2006).

There has been an increase in research on software metrics and their validity over the past 20+ years. Many researchers have attempted to determine what metrics are and how to determine their validity (Fenton & Kitchenham, 1991; Gustafson, Tan, & Weaver, 1996; Morris & Desharnais, 2001; Prather, 1984; Schneidewind, 1992; van der Berg & van der Broek, 1996). Albrecht (1979) and Albrecht and Gaffney (1983) proposed the function points model that measures the functionality of a system beginning with a description of the user requirements in natural language. This model offers a technology-independent assessment of the size of the final product and is probably the only measure of size as a non-code-related attribute of software programs. The measurement of these function points builds on the functions of identification and computation that the system is expected to perform (Albrecht, 1979; Albrecht & Gaffney, 1983).

Albrecht's (1979) model identifies five function types: external inputs, external outputs, logical internal files, external interface file, and external inquiry. Every function is classified according to three metrics of complexity—simple, average, or complex—by taking into account the aggregate number of data element types that the function uses and the file types referenced. A weight is then assigned to each function, based on function type and complexity. The unadjusted function point measure is obtained by totaling all the weights. Albrecht and Gaffney (1983) reported a high-level correlation between function point and work-hours and the size of the application according to the lines of code. They argued that the function point could be sufficient for estimating the software

development effort. Empirical studies, such one by as Stathis and Jeffrey (1993), revealed the unadjusted function count is an improved predictor of software development effort.

Munson and Elbaum (1998) reported on the importance of complexity metrics in informing the distinctions among software modules, especially in the context of the process of software testing. They argued that software metrics can be used as surrogate indicators of the quality of software products during the testing process. Their relative complexity metric and its derivatives, including code delta and code churn, are subtle measures of certain attributes of software quality that cannot be determined directly. Compared to other popular software complexity metrics, this model integrates all of the quality attributes of all other complexity metrics into one measure. Munson and Erlbaum proposed the relative complexity metric and its derivatives are closely related to measures of software quality.

High-level software design metrics are also well represented in the literature. Basili, Briand, and Melo (1996) proposed a framework for describing and validating metrics for traditional and object-oriented high-level software designs. The purpose of these metrics is to measure the quality of design in relation to the likelihood of errors in the software product (Emam, Melo, & Javam, 2000). Chidamber and Kemerer (1994) defined a set of complexity metrics, including cohesion, complexity, the number of children, response, coupling, and depth of inheritance—applicable to object-oriented designs.

Based on the IEEE software requirements specification (SRS) document ("IEEE recommended practice," 1994), Sharma and Kushwaha (2010) suggested a complexity metric to satisfy the requirements for an SDLC. The approach used by the metric defines software complexity immediately after the requirement freezing phase. Sharma and

Kushwaha's model is based on a definition of 10 attributes of complexity. The model compares the results of complexity measures to several cognitive- and code-based measures such as cognitive information complexity measure (CICM), kinds of lines of code identifier density (KLCID), cognitive functional size (CFS), McCabe's model, and Halstead's model. The metrics of Sharma and Kushwaha's model do not address software change differently from other metrics. The scope of the equations in this model include the estimates of the initial system requirements for the referent system. Singh, Sabharwal, and Sood (2004) developed an algorithm for assessing the strength of each distinct system requirement or a group of requirements. While Sharma and Kushwaha focused on the properties of each requirement independently, Singh et al. focused on the correlation between the collective requirements.

This review of the literature revealed the challenge of validating software complexity metrics has been addressed from different perspectives—empirical, theoretical, or mathematical—using varied interpretations of the concept of metrics validation, as proposed by Kitchenham, Pfleeger, and Fenton (1995). One of the most frequently cited validation approaches in the literature was that of Jacquet and Abran (1998), who proposed a process model for software management. Jacquet and Abran used their process model to position the validation criteria of several authors including Kitchenham, Fenton, and Zuse among others.

2.4 Complexity in Incremental Software Change

There have been numerous studies on the complexity of incremental software development or change. For example, Munson and Elbaum (1998) motivated research on code churn, adding to the knowledge base on the link between software faults and

software complexity (Khoshgoftaar, Allen, Kalaichelvan, & Goel, 1996). Buckley, Mens, Zenger, Rashid, and Kniesel (2005) proposed a taxonomy of software change. Ghosh, Sharma, and Mohabay (2011) discussed various software change models, even though none of the models in their literature review includes the process of validating the quality of the proposed change design before the implementation phase (Satapathy, Avadhani, Udgata, & Lakshminarayana, 2013).

Munson and Elbaum (1998) suggested a measure for approximating the effect of code change through an accurate measurement of the process of software development and product outcomes. Their study revealed a relationship between the relative complexity of a software module and the overall measure of a software fault, developing the premise that a higher relative complexity of a software module is likely to cause a greater number of latent software defects. Munson and Elbaum argued that the process of introducing new code through sequential software builds is as prone to faults as the initial code generation. That is, critical faults are introduced into the builds throughout the software evolution process. Similarly, as the system progresses through a sequence of changes, the relative complexity of every module that is affected by the change will also change. This construct provided a framework for Munson and Elbaum's index for measuring the rate of introducing faults to a software module, as indicated in the rate of change in the relative complexity of the system.

Munson and Elbaum's (1998) measurement index developed from a relative complexity procedure they developed over several years. The metric is derived from a complexity metric suite by a two-step process. In Step 1, the factor scores for each module or the referent software build are established for each set of complexity metrics,

using both orthogonal transformation matrix and principle component analysis. The domain metrics, or the orthogonal factor scores, are pooled together to form a relative complexity measure (Khoshgoftaar & Munson, 1992). Step 2 involves characterizing each program module in the system by a value that represents some cumulative complexity measure. This process leads to the development of a linear function, g, that is either inversely or directly related to software faults such as that g (x) = as + b, where x is a unitary complexity measure of the referent program. Researchers who preceded Munson and Elbaum proposed that a relative software complexity metric has attributes that are useful in predicting software faults during progressive software development. From a software evolution viewpoint, Munson and Elbaum proposed that their relative complexity metric facilitates the measurement of the amount of change in a software system, as well as the rate of change in the overall complexity of the system. This two-pronged solution makes relative complexity a useful and an appropriate surrogate measure of faults (Khoshgoftaar & Munson, 1990).

Research on the measure of software complexity in the analysis phase has also advanced in recent years. While some scholars have focused on measuring complexity at the requirements engineering phase (Sharma & Kushwaha, 2010), others have attended to measuring complexity in the analysis phase of SDLC (Tran-Cao, Lévesque, & Meunier, 2004) and the effects of change on complexity (Gupta, Singh, & Chauhan, 2010). Tran-Cao et al. (2004) analyzed the complexity of software products in the analysis phase by proposing three principle measures—the number of conditions (NOC), the entropy of the software system (EOS), and the number of data groups (NOD). This use of entropy is drawn from Shannon's (2001) statistical entropy concept. Tran-Cao et al.'s (2004) use of

entropy is substantially different from that used previously by Canfora et al. (2014). Tran-Cao et al.'s EOS technique was used to analyze the software source code and look specifically for how much data were entering and leaving function calls. From these data, Tran-Cao et al. created a directed graph that was used to calculate their metric of the entropy of system data communications. Systems in which all functions have similar levels of data entering and leaving them are considered to have a low entropy, whereas systems in which every function has a different level of data entering and leaving are considered to have a high entropy (Tran-Cao et al., 2004). Based on existing frameworks such as McCabe's model, Tran-Cao et al. (2002) suggested an entropy metric for describing the complexity that characterizes the associations between functions.

Gupta et al. (2010) proposed a model for measuring the effect of software change on complexity by reviewing both the type of change and its impact on the software system. They classified software change into four main categories: logical change impact, behavioral change impact, functional change impact, and structural change impact. The model analyzes software change after implementation to help testers and quality assessors identify the best regression test for every test case. Most of the literature on measures of software complexity focuses on either measuring the complexity of a new project through the definition of metrics extracted from the system requirements document (Singh et al., 2004) or using traceability links between system requirements to assess the impact of change on existing software (Maadawy & Salah, 2012).

Literature on software testing and quality assessment acknowledges the role of regression testing in delivering software products with a smaller chance that newly introduced changes do not adversely affect the behavior and function of the current,

unaffected modules of the software system (Pravin & Srinivasan, 2013). Literature also recognizes the importance of research on software testability measurement. The goal of software testability evaluation is to (a) identify and rank poor-quality elements of a software system and (b) determine where critical faults can hide. Most previous studies focused on defining techniques for measuring software testability at different stages of the software development lifecycle, such as design phase (Binder, 1994; Khan & Mustafa, 2009; Nazir, Khan, & Mustafa, 2010; Pettichord, 2002) and the coding phase (Baudry, Le Traon, Sunye, & Jezequel, 2003; Kout, Toure, & Badri, 2011; Y. Wang, King, Court, Ross, & Staples, 1997). Common among these studies is the use of OO software quality metrics for evaluating the testability of OO software systems. Other researchers such as Badri and Toure (2012), Bruntink and van Deursen (2006), and Singh and Saha (2010) sought to show the relationship between these quality metrics and unit testing.

2.5 Using Analytical Hierarchy Process with a Programmatic Model

The programmatic model presented in this praxis measures and combines multiple aspects of source code complexity for a software application at an initial revision and after incremental changes to the software source code. This model is limited to software baselines written in the Java OO programming language to confine the scope of work. This methodology was selected because of the availability of data, tooling, and prior work. A programmatic model was adapted from the data extraction model described by Canfora et al. (2014). Coincidently, Canfora et al.'s concept of an entropy measurement model employs a data extraction framework that is designed to identify the differences between software source code revisions. Specifically, Canfora et al.'s data extraction

framework was designed to analyze software version differences and thereby identify which files were changed between software versions. These data were used in Canfora et al.'s concept of an entropy measurement model to evaluate their aggregate metric of the entropy of a project. Canfora et al.'s concept of a data extraction framework is useful and is used in this praxis to develop a programmatic model. In this programmatic model, the concept of an entropy measurement model is replaced with a software difference complexity measurement model.

For this praxis, the programmatic model is an adaptation drawn from several successful prior efforts. The software difference complexity portions are drawn from the factors-criteria-metrics (FCM) model described in by McCall et al. (1977). The resulting software difference complexity model is built on an open-sourced version of the set of software complexity metrics specified by Chidamber and Kemerer (1992), which are commonly referred to as the CK metrics. These metrics are appropriate to the OO nature of Java and provide measures of encapsulation, inheritance, coupling, cohesion, size, complexity, and polymorphism. The output of the software difference complexity model is then introduced into an AHP model. This process provides a popular framework for the analysis of complex decisions based upon multiple criteria of both qualitative and quantitative evaluations. In effect, the AHP provides the framework for evaluating the impact and effect of multiple measure of complexity.

Application of the AHP to the field of software complexity measurement is not new. Suri and Singhani (2015) successfully applied the AHP to Java OO complexity measurement to develop a testability assessment of a software baseline. Suri and Singhani's paper has been cited repeatedly by scholars in the field of OO metrics analysis

and software estimation techniques. This praxis expands this area of work with development of the software difference complexity model to understand the differences of a software baseline as it evolves across revisions.

This model quantitatively measures the metrics of a software code for programs written in Java using a static analysis approach, which obviates the need for compiled software code. The programmatic model incorporates a broad array of complexity metrics, including the CK metrics as measure of encapsulation, inheritance, coupling, cohesion, size, complexity, and polymorphism.

- Encapsulation is represented by the aggregated number of methods (NOM) in a class.
- Inheritance is represented by the number of children (NOC) from the superclass.
- Coupling is represented by the coupling between objects (CBO), which is the number of other objects that a class internally references.
- Cohesion is represented by the level of cohesion metrics (LCOM), which considers how related one class is to another based on a comparison of the two classes set of methods and those parameters of the method.
- Size and complexity are represented by the weighted method complexity (WMC) which uses the aggregate cyclomatic complexity value of all methods in a class.
- Polymorphism is represented by the number of methods overridden (NMO), which is a count of the number of methods that a class overrides from its superclasses.

Table 2-1 represents an outline of the software complexity metrics used in this praxis and it is followed by a more detailed description of each of these metrics.

Table 2-1: Software Complexity Factors and Corresponding Metrics

Testability factor	Metrics name	Description
Encapsulation (M1)	Number of methods (NOM)	Count of all methods in a class
Inheritance (M2)	Number of children (NOC)	Count of children of a super-class in the project
Coupling (M3)	Coupling between objects (CBO)	Count of external classes referenced from class
Cohesion (M4)	Level of cohesion (LCOM)	Similarity of methods in a class based upon parameters
Size and complexity (M5)	Weighted method complexity (WMC)	Aggregate count of logical code paths through all methods in class
Polymorphism (M6)	Number of methods overridden (NMO)	Count of methods inherited from a class that have been overridden by its subclasses

2.5.1 Number of Methods Metric (NOM)

The first metric that the software difference complexity model measures is the number of methods (NOM) in a class irrespective of its modifiers. The model determines the number of methods defined in a class. It is desirable to maintain a NOM metric below 20, which is the recommended maximum accepted value of NOM (Baumeister, Marchesi, & Holcombe, 2005). The program counts all declared and overridden methods, constructors, and static initialization blocks using the principle outlined by Whittle, Clark, and Kühne (2011), as described below.

1. Obtain the inherited member operations, which return all inherited yet operational members of the superclass.
2. Obtain the total number of inherited operations, which returns the subset of the Step 1, excluding those operations that are overwritten or locally defined.
3. Identify the set of local methods, which returns a set of all locally defined operations in a class, except the elements of the set obtained in Step 2.

4. Calculate the NOM value as the size of Step 3. The number of operational methods localized to a class is the value returned by the NOM metric.

2.5.2 Number of Children Metric (NOC)

The second metric is the number of children (NOC), which refers to the measure of the number of immediate subclasses that are derived from the base class. This metric measures the breadth of a class hierarchy, especially when a maximum DIT is used as a measure of the depth of inheritance. The decision to use depth instead of breadth is based on the premise that depth, as an attribute, promotes the reuse of methods within the class through inheritance. A high value of NOC is perceived as an indicator of a high reuse of the base class, the need for more testing for the base class, an abstraction of the base class, and a potential misapplication of the attribute of subclassing.

Because a high NOC value is indicative of fewer faults—thus a smaller complexity—this model measures the trends on NOC value between versions of an OO software program written in Java. The complexity of a class is rated based on the acquired measures of NOC and WMC. Values of NOC and WMC at the top of the class hierarchy suggest the possibility of the base class influencing the properties, especially the complexity, of the descendant classes (Basili, Briand, & Melo, 1996). This attribute can also be used to measure the quality of the design of the software product.

2.5.3 Coupling Between Objects Metric (CBO)

Coupling between object classes (CBO) is the third metric of the programmatic model. A set of classes is coupled when each class can use the instance variables or methods defined in the other classes. For this model, multiple accesses to a class are quantified as a single access by counting only variable references and method calls to

improve the dependability of the data obtained. User-defined object types, event handling procedures, object instantiations, constants, and declaration of API calls are ignored when measuring CBO. A high CBO is detrimental to the design of modules and program constructs, and naturally limits reusability (Basili et al., 1996). The measurement procedure used in this model is a CBO benchmark of < 14, as proposed by Sahraoui, Godin, and Miceli (2000). This approach reflects the distribution of the fan-out values of classes to provide insight into the latitude of object orientation in the design of the module and the entire system.

Because conceptualization of CBO as a metric emphasizes all methods and instance variables in a class, the method in this praxis does not take into account class variables or shared variables. All methods—both shared and non-shared—are taken into consideration. This approach presents considerable ambiguity, but it only functionally limits the implementation of CBO—not its definition (Basili et al., 1996; Briand, Daly, & Wust, 1999). The model in this praxis references the definition of CBO and handles inheritance of class attributes as it would a conventional coupling.

2.5.4 Lack of Cohesion of Methods Metric (LCOM)

The lack of cohesion of methods (LCOM) as a complexity metric takes into account all methods of a class, both shared and non-shared, but only the instance variables of the class (Basili et al., 1996; Chidamber & Kemerer, 1992). Therefore, this program supposes that any defined procedure in a base class is a method if it contains code. Attributes such as functions, subs, and properties such as GET, SET, and LET are considered as methods for this task. Event declarations, DLL declarations, and the definitions of empty procedures are not viewed as methods when quantifying this metric.

The choice of LCOM is predicated on the disregard for the impressions of procedure calls and property accessors in a class.

This metric is used to quantitatively determine the cohesion between a pair of methods in a class. Widely referred to as LCOM1, the value of LCOM is calculated using the following equation: LCOM1 = P - Q, iff P > Q and LCOM1 = 0 otherwise. If the pair of methods references disjointed sets of instance variables in a class, the value of P is increased by 1. Comparatively, the value of Q is increased by 1 if the pair of methods shares at least one variable. The measure of the cohesiveness of a class is determined by comparing the values of LCOM to a benchmark of 0 such that LCOM = 0 and LCOM > 0 are indicative of a cohesive class and the ability or need to split a base class into a number of subclasses. A high value of LCOM suggests disparateness and disjointedness in the functionality of a class (Henderson-Sellers, Constantine, & Graham, 1996). This metric is used to isolate classes that attempt to achieve different objectives and are more likely to behave less predictably than classes with lower LCOM values.

2.5.5 Weighted Methods per Class Metric (WMC)

Weighted methods per class refers to the summation of the cyclomatic complexity scores for all of the methods in a class. The cyclomatic complexity is the total number of logical paths that can occur through the methods that are defined in a class. For instance, a method with a single "if (value < 5)" statement shows two possible paths through the code. The first occurs if the provided value is less than 5, and the second occurs if the provided value is greater than or equal to 5. The number of code paths through a method is equivalent to the WMC value for the method. It is a norm to keep cyclomatic values down—a high measure of cyclomatic score can lead to more flaws in the code. That is,

classes with several methods tend to be more application-specific, consequently limiting the prospects of reusability. Because this metric is a predictor of the amount of time and effort necessary for the development and maintenance of a class, a higher number of methods defined or contained in a class implies a greater possible impact on the application-specific derived classes, which will inherit a number of the methods of the referent class. Therefore, the weighting in this model highly values both WMC and NOC as pointers to the classes that could be reviewed and methodologically restructured into many functionally and structurally smaller classes.

Different limits for an adequate measure of WMC have been defined to accommodate both small and large classes. To achieve the objectives of this praxis, all relevant code blocks are included in the WMC measurement, including non-default constructors, anonymous implementations, static declaration blocks, and lambda functions. Every property accessor defined in the code is calculated separately because each provides a unique way of accessing its internal code paths and is a method of the referent class. While WMC could be treated as a weighted count by weighing each method using its size or overall complexity, in this praxis, WMC is treated by the original definition as the summation of the complexity of the declared methods of a Java class.

2.5.6 Number of Methods Overridden Metric (NMO)

The number of redefined class-level or routine-level operations, plays a role in the definition and specialization of the class and should be maintained in a proportion that justifies inheritance. NMO, which was introduced by Lorenz and Kidd (1994), is a metric for measuring inheritance and is defined as the number of inherited methods that have been overridden by a class. NMO counts the number of methods in a class that have been

overridden by its subclasses. A large value of NMO is indicative of a problem (Gao, Tsao, & Wu, 2003). A high number of redefined operations within a class implies a large difference between the subclass and the parent class; inheritance would make less sense in this scenario.

Before defining and using metrics, the attributes that will be measured and the entities to which the metrics apply must be identified. NMO is defined in terms of methods and classes, but Java has many attributes that are similar to classes, such as interfaces. There is no guarantee that Java methods have the necessary implementations. This problem causes confusion about what number of methods inherited means as a measure. For overriding to serve as a measure of complexity of Java source codes, the meaning of overriding must be established.

In this praxis, overriding is conceptualized as an outcome of the substitution of implementation. A method n in a class B overrides a method m from a superclass A, and the method n must be implemented in both A and B. Overriding only occurs when an implementation of a method is replaced by another method. In this sense, abstract methods are not potential candidates because there is no replacement of implementation taking place.

2.6 Principles of Measurement & Metrics

Assessment of software complexity has attracted software engineering researchers for many years. Software complexity contributes to the overall costs associated with development and maintenance of software products (Grady, 1992; Jiang, 2015). Al-Hajjaji, Alsmadi, and Samarah (2012) asserted it is becoming increasingly difficult to ignore the complexity of software products, leading to the proposition and development

of software metrics to show indicators for attributes of software products such as size, complexity, and quality among others. Coad and Yourdon (1991) posited that a good object-oriented software design should encourage and allow tradeoffs among the costs of system analysis, design, development, implementation, and maintenance throughout the software development lifecycle. Further, Coad and Yourdon argued the possibility of trade-offs would minimize the total lifetime costs associated with the system.

Tegarden et al. (1992b) observed the measurement of software complexity could contribute to these cost adjustments in two ways. First, software complexity measurement provides a quantitative approach to forecasting the difficulty of designing, implementing, and maintaining the system. Second, software complexity measurement provides a framework for implementing the tradeoffs necessary to reduce costs over the system lifecycle. Although few studies of complexity measures have expressly reported it, the need to improve the capability to predict the quality, effort, cost, and coding efficiency has spurred scholarly efforts (Kaur & Verma, 2016). This rationale has led to increased research in software complexity that includes psychological complexity, structural complexity, and functional complexity.

The unification of disparate software complexity measurement techniques has received some coverage in the academic body of knowledge. Relatively few papers have focused on this narrow research area; in a few cases, this activity was performed as part of a larger research scope. There is similarity between these fields of study: the overall concept of complexity in software is a function of data structures, control structures, module coupling, language, nesting level, and algorithms (Gonzalez, 1995). This similarity implies that software complexity is a multidimensional problem in which each

metric might have ordinal values independent of one another and that the relationships between metrics are important for the measure of complexity (Fenton, 1994). This implication is reinforced when viewed in aggregate.

Many programming language-specific techniques of software complexity measurement leverage overlapping subsets of commonly used techniques to measure software complexity (Fenton, 1991; Gonzalez, 1995; Zuse, 1994). The McCabe cyclometric complexity, Halstead complexity, SLOC, and fan-in/fan-out are frequently mentioned in the development of language-specific measurement techniques (Hsu & Huang, 2014; Shi & Xu, 2007). Although several novel techniques were presented, most of these scholars used a weighted combinational model to develop a unified metric (Gonzalez, 1995; Hsu & Huang, 2014; Jabbar & Sarala, 2011; Shi & Xu, 2007; Tegarden et al., 1992a).

Fenton (1994) emphasized the need to find more resilient software metrics, but many metrics are theoretically inconsistent. Riguzzi (1996) conceptualized measurement as the process of assigning symbols or numbers to qualities and elements of entities to describe them based on well-defined rules. For this assignment to be useful, it must preserve any empirical or intuitive observations about software entities and their attributes. Fenton pressed for a robust model of these attributes to avoid ambiguities. From Fenton's argument, Riguzzi proposed a basic representational model of measurement that accounts for the axioms that every measure of complexity must satisfy to characterize an attribute.

Riguzzi's (1996) argument is that empirical relations system must exhibit the aim of describing the meaning of an attribute by defining the empirical relationships that are true for that attribute. Apart from defining a numerical relation system denoted by (N, P),

Riguzzi conceptualized empirical relations systems as a couple denoted by (C, R). Measuring any attribute of this system involves establishing a relationship between (C, R) and (N, P) using a transformation (M). Riguzzi's representational measurement model reduces the temptation to characterize a poorly implicit yet intuitively perceptible attribute with respect to some numerical assignments. Riguzzi asserted that such a predisposition is both a critical and common flaw in the literature on software metrics, as exemplified in Halstead (1977) and McCabe's (1976) complexity models.

The review of the literature on complexity measurement models and metrics revealed there is no collectively recognized conception of validity. Although some approaches to the validation of complexity metrics have been suggested, the debate on metrics validation is still open. In most of the literature, the validation of metrics was conducted either analytically based on Weyuker's (1988) properties or empirically by demonstrating the correlation between a given complexity metric and some other prevailing measure. Many scholars have attempted to overcome the inconsistencies of Weyuker's properties and the flaws exhibited by other models of complexity metrics by using new frameworks for metrics validation. For instance, Fenton (1994) asserted that validation of an assessment-type complexity measure requires an empirical demonstration of the validity of the representation condition for the attribute being assessed.

Briand, Morasca, and Basili (1996) proposed a mathematical model to define complexity and other critical software attributes such as size, length, coupling, and cohesion. For every attribute, there is a set of properties that a realistic measure should exhibit. Even though their mathematical framework borrows heavily on Weyuker's (1988) axiomatic model, Briand et al. (1996) adopted a mathematical archetype of

modules and systems that enabled them to define the properties of attributes clearly and rigorously.

Kitchenam et al. (1995) proposed a structural framework for software complexity measurement that includes attributes, scale types, entities, units, instrument, values, and the properties of values, as well as their correlations with each other. They identified two types of validation based on this structural model. First, an empirical validation provides supporting evidence of the validity of metrics by establishing the existence of consistency between the assessed values of an attribute and the values projected by the models that involve the attribute. Second, theoretical validation of metrics involves attesting that the measure conforms to all properties of the definition models or the basic elements of measurement. Regardless of scholars' expansive efforts, a realistic validation of metrics is merely hypothetical; empirical frameworks can only substantiate the validity of a complexity metric (Riguzzi, 1996).

Gonzalez (1995) noted that a metric can be conceptualized as a function with two arguments. From a mathematical viewpoint, metrics are akin to measurements for software complexity and are functions with one argument; these measurements are the ordinal quantities of complexity. In this context, many metrics are inadequate because they quantify only attributes of program complexity regardless of the various sources of complexity in the software programs. The most appropriate metric uses the greatest quantity of the attributes of these complexities to describe the program and its complexity better (Gonzalez, 1995). Most metrics are basic linear composites of other metrics. Khoshgoftaar and Munson (1992) cautioned that the compounding of two or more

complexity measures does not provide additional information on the complexity of a software program.

To address the issue of unifying two or more metrics to develop a broader understanding of program complexity, Gonzalez (1995) proposed an integrated complexity metric that incorporates the dimensions, factors, and domains of software size. Gonzalez opined this unified complexity metric would be useful in providing a robust measurement system for comparison of the relative software complexity of two dissimilar programs. This model would help to obtain an indirect complexity measure to predict or estimate the productivity of coders or programmers involved in the project and the performance, value, reliability, and quality of the final product.

2.7 Critical Gaps in the Literature

The software complexity models or metrics described in the literature pose challenges in resolving issues related to complexity in several ways. First, the dependence of modeling languages on different semantics spur the need to adapt and customize existing software complexity metrics to the existing modeling languages. Second, modeling tools and frameworks need to create environments in which users can make better design choices while avoiding design patterns that might cause additional software complexity for the final system.

Despite a number of seemingly reasonable and workable options proposed over the years, the literature contains no effective strategy for measuring software complexity (Du & Wang, 2010). Most of the research on software complexity overlooks data scope complexity and data complexity. Most authors focused their research on assessing the information flow complexity and the impulsiveness of cognitive weight.

Measures of the attributes as well as quality parameters of the entire software development process are central to the software development process. Metrics, complexity measurements, and quality prototypes ensure that the software development process—through its successive phases of change—is under control and meets the defined quality objective of the software product. The use of these metrics, models, and measurements in software development assumes the availability of reliable data (Bilal & Black, 2005), but poor-quality test data remains an obstacle to software quality improvement. Each complexity metric or measurement must be evaluated before being used to measure the quality of software products and the effect of software change on key quality attributes. Improving data accuracy requires a robust change management protocol and a tracking system for the whole process of software development.

Apart from the CK, MOOD, and Henderson-Sellers metric suites, several empirical studies have been conducted on the relationship between complexity metrics and unit testing. Few empirical studies have focused on the UML diagram characteristics from prospects for improvisation of software testability during evaluation of diagrams (Suri & Singhani, 2015). Even less still scholarly effort has addressed testability analysis through the MCDM models such as AHP, although testability factors depend on an array of criteria.

Despite the sizable body of academic knowledge, there have been few attempts to link the application of software complexity analysis to agile methodology with the intent of measuring the software complexity of a specific software change. Given that ASD methodologies use many small but frequent changes, it would be helpful to understand the delta changes imparted in the complexity of a software source base for each of the

small changes in ASD process. Canfora et al. (2014) utilized a similar approach to measure the effect of differences in software entropy in a software baseline. Canfora et al. found the change in complexity and organization across software components can be either an increase or decrease in complexity. The task of developing complex software to re-factor a software component might reduce the complexity of the system as a whole. While further research is needed, it may be beneficial to further the entropy measurement model presented in this paper.

2.8 Conclusion

As systems become more reliant on software, the importance of measuring and managing software complexity increases, in part because of the potential for cost savings throughout the system lifecycle (Banker et al., 1993; Delange, 2015). The current literature leans toward the conclusion that software complexity affects both the development and maintenance processes, and that keeping complexity under control could reduce costs. Because the development of safety-critical computing systems is shifting toward the widespread use of model-based tools, it is important to design complexity metrics, models, and tools while adapting the environment for software development to detect and escape complexity (Delange et al., 2015).

Regardless of the state of the literature on software complexity, further research is needed to advance the strategies and techniques of software complexity measurement. First, there is need to develop new complexity metrics that refine the existing software complexity measures. Second, further research is needed to resolve concerns about the efficiency and reliability of traditional metrics in assessing software complexity, particularly in object-oriented systems and procedures. Third, object-oriented attributes

such as encapsulation, information hiding, and polymorphism require further research to nurture the development of object-oriented metrics while providing a framework for an empirical validation of the developed complexity metrics. Fourth, further investigation is needed to build a broader understanding of complexity metrics as well as the appropriate number to define a complexity limit in software development. Finally, further investigation is required to establish and assess the correlation between the structure of the code and the complexity of aspect-oriented programs.

This praxis adds to the literature on software complexity measures and metrics by developing a multi-aspect technique to determine the changes that occur within a software development baseline. Gaps in the literature on software complexity measures point to the need for more research on the metrics, models, and measures of the complexity of changes in software source code between successive revisions. This praxis adds to the knowledge base by establishing the correlation between the complexity of a component of software to attributes of the software development process. With added research, this process could be expanded to include the concept of ASD efficiency into a similar framework. This framework would provide linkage between the complexity evaluation of software with the determination of the efficiency of the ASD process.

Chapter 3 – Methodology

Software developers and architects have suggested many measurement methods and metrics for evaluating software complexity. These methods and metrics involve quantitative assessment of the effect of software complexity on the quality attributes of a software product, including performance, understandability, quality, and maintainability. To generate the necessary complexity metrics for evaluating software quality, the assessment tools extract data on coupling, the number of lines in the source code, and cohesion from the source or binary code to quantitatively evaluate the complexity and quality of the software. Although these single-metric approaches are effective, these tools, methods, and metrics are insufficient for managing model complexity, especially as the SDLC shifts toward model-based software engineering (Delange, 2014).

The domain of software engineering has shifted toward the techniques formerly associated with embedded systems. Developers of many software projects have adopted model-based software engineering tools and methods to quantify and capture the functional architecture of the software product (Delange, 2014). Courtesy of automated code generators, many of these systems use complex frameworks or are generated automatically from underlying models; the project's application code is then developed based on these frameworks or generated skeleton representations of software systems. Code metrics calculated from these types of implementations may not match the development or engineering effort required to create them. The introduction of a simple change to a component of a software model can trigger a disproportionate change to the

source code. Alternatively, some changes introduced to the models or frameworks do not always cause a change in the source code.

Traditional code analysis techniques based on a core complexity measurement technique, whether at the source- or binary-code level, should not be used as software systems become more embedded, complex, and dynamic. New methods are needed to assess the complexity and quality of auto-generated abstract models of software products. The goal of the method should be to evaluate the impact of the introduced change on the overall quality, measured quantitatively. To meet this need, Delange (2014) recommended the use of a multimetric approach. This praxis makes use of a programmatic model based on the analytical hierarchy process as a multicriteria approach to assessing the complexity and quality of software products.

This praxis analyzes the effect of factors associated with software development activities and software attributes. Of interest is the incidence of re-factoring software changes, the nature of the introduced change, the impact of the software change with respect to the different aspects of software complexity as measured by various metrics, and the interrelationship between classes and design patterns on complexity. The six software source code aspects of encapsulation, inheritance, coupling, cohesion, size and complexity, and polymorphism understandability and maintainability are the complexity quality attributes at the center of this praxis. The effect of the progressive introduction of changes to a software baseline on these quality metrics constitutes the third hypothesis that complex changes in a software baseline will be quantitatively different with respect to their software complexity values throughout a software development process.

3.1 Research Questions and Hypotheses

Three research questions guided this praxis:

RQ1. What software complexity measurement techniques are best suited to measure the software source code changes performed between subsequent software revisions?

RQ2. How can an accurate measure of the complexity of the software source code changes performed between subsequent software revisions be developed?

RQ3. How can software changes with a large magnitude of measured complexity be identified in a software baseline?

The following hypotheses were tested to answer the research questions:

H1. The complexity of a software change can be expressed as a function of the complexity of multiple aspects of the software source code.

H2. The complexity of a software change can be expressed as a function of the complexity of the software source code at an initial point and the complexity of the subsequent sets of changes applied to the initial software source code.

H3. Significantly complex changes in a software baseline will statistically differ with respect to their software difference complexity values throughout a software development effort.

3.2 Sources of Data

This praxis leveraged software source code data derived from publicly available open source Java software repositories. This approach ensured that source data were plentiful. Care was taken to identify a range of project sizes as well as projects with well-identified

releases for reproducibility of results. This praxis used a moderately large software project, Elasticsearch (https://github.com/elastic/elasticsearch), as test data for the development of the system. Elasticsearch is under active development and contains software changes ranging from minor documentation changes to major changes that impact hundreds of files. A survey of the recent commit logs revealed a mix of changes made to the software, including the addition of new classes, the modification of existing classes, and the deletion of classes. Each of these types of modification placed stress different complexity aspect metrics. Finally, the Elasticsearch project is actively curated: there is an admin enforcing and managing changes to the project baseline. Elasticsearch has a coding style guide that enforces the best practice of the Java development language.

Four separate Java projects were used to calculate and evaluate results. These projects were all drawn from subprojects of the Apache Commons project. The Apache Commons project is a set of 41 open-source projects focused on the creation of reusable Java software components. These projects are community-developed, widely used, and have few dependencies on external libraries. The subprojects used were the Discovery (P_1), Net (P_2), IO (P_3), and Compression (P_4) subprojects. These subprojects were chosen because they offered a range of size, complexity, length between releases, and maturity. Software source code data were collected from subsequent releases to these subprojects. These releases spanned many weeks or months of effort and contained a collection of many different commits, pull requests, or merges into the release branch. This variety of types of changes provided conditions similar to those common to moderately large software development effort. Details on the source code data included in this praxis are noted in Table 3-1.

Table 3-1: Data Sources, Revisions, and Average Project Size

Apache Commons project	Average size (NCSLOC)	First version (V_1) / date	Second version (V_2) / date
Discovery (P_1)	2,565	v0.4 / 11-29-2006	v0.5 / 4-27-2011
Net (P_2)	27,612	v2.4 / 5-1-2016	v2.5 / 2-11-2017
IO (P_3)	26,767	v2.4 / 6-12-2012	v2.5 / 4-14-2016
Compression (P_4)	36,639	v1.13 / 1-13-2016	v1.14 / 6-18-2016

Table 3-2 provides some heuristics from the four Apache Commons projects used in this praxis. This table identifies the number of Java source code files in each version of the project, as well as the number of files that experienced a change in calculated complexity scores between the specified versions. The number of files added to the project is also noted for convenience.

Table 3-2: Project 1-4 Version Heuristics

Project	V_1 #Files	V_2 #Files	#Changed	#Added
Discovery (P_1)	47	49	47	2
Net (P_2)	248	250	30	2
IO (P_3)	198	227	165	29
Compression (P_4)	282	314	51	32

An open-source complexity measurement tool chain was adapted to provide the basis of the software difference complexity model in the form of a data extraction framework. Multiple toolchains were evaluated, including Squale, Sourceforge's Metrics project, and Google's Testability Explorer, and several implementations of the CK metrics suite. Each of these tool chains provided coverage for the required metrics. The Github project Mauricioaniche/CK was selected as the basis for the data extraction framework for the following reasons. Mauricioaniche's implementation of the CK metric suite used the Apache 2.0 licensing model was conducive to developing derivative works. Furthermore, the implementation was simple, well encapsulated, and logically organized which made it easy to understand. The metrics calculations were based upon the mature Eclipse AST

Java parsing framework, which necessitated only minor modifications. Finally, the Mauricioaniche CK implementation did not have a user interface developed which was desirable for future automation possibilities.

Only delivered code—that is, software code that is obtained from an official released version—was acquired for use in this method. This limitation eliminated any temporary code while retaining parts of the code that might not have been included in the delivered software system but was perceived as sufficiently valuable to be saved, managed, and controlled in the repository. Such code might have included documented procedures used to recompile the system, programs that were used to generate test data, and utility routines, among others. The source code of each of the programs evaluated was used as the primary source of data for testing hypotheses and for answering the research questions.

In the context of this praxis, and by following the industry-accepted Gill and Kemerer's (1991) approach, a software module or unit chosen for being measured was selected only if found in a physical file in the repository. This praxis used non-comment source lines of code (NCSLOC) as a standard for measuring the size of the software modules. To improve the accuracy of the measurement and analysis of lines of code, the data definitions used disregarded blank lines and comments. The data definition conceptualized "lines of code" declarations, executable and non-executable program statements, and program headers. This data definition approach ensured simplicity in implementation of the methodology while fostering the ability to translate this methodology to other programming languages, such as C++ or Python.

The project data were acquired from the Apache Commons projects using subsequent versions to test the research hypothesis related to the objective of developing a technique that measures the complexity of the software source code changes performed between software revisions. The average size of the software maintenance projects was determined by quantitatively measuring the difference in the number of modified source code files between two successive versions.

3.3 Calculating Version Differences

Following the data acquisition, the next step involves calculating the source code differences between the various versions. The data extraction framework processes all revisions of all source files to produce matrixes conceptualized as software difference complexity models. These software difference complexity matrixes describe what part of the program have been affected between the specified versions, based on the observed effect of the changes in the source code on metrics. The data extraction framework used the metrics to collect complexity data at the class level, but does not evaluate the compiled code.

The software difference complexity matrix output of the data extraction framework defines the complexity for each source file using a set of four values per metric. The first and second values are the raw complexity scores for the specified metrics for the first revision of the source code file and the second revision of the source code file, respectively, where V is the value number, M_q is the metric function, F_{n1} is the first file revision, and F_{n2} is the second file revision. Equations for Values V_1 and V_2 are presented in Equation 3.1 and Equation 3.2, respectively.

$$V_1 = M_q(F_{n1}) \quad (3.1)$$

$$V_2 = M_q(F_{n2}) \quad (3.2)$$

The third value is the difference in the specified metric between the first and second revisions in absolute terms. The equation for value V_3 is presented in Equation 3.3.

$$V_3 = \left|M_q(F_{n2}) - M_q(F_{n1})\right| \quad (3.3)$$

This value is the complexity delta for the specified metric and between the two software versions. The evolving nature of a software baseline means there might not be a file in both revisions. Files are commonly added or removed from software baselines between revisions. In other words, the V_1 or V_2 values will return zero in the case where the file was added or deleted, respectively. The fourth and final value is the final complexity difference representing the chance from F_{n1} to F_{n2}. This value V_4 is represented in Equation 3.4.

$$V_4 = M_q(F_{n1}) + \left|M_q(F_{n2}) - M_q(F_{n1})\right| \quad (3.4)$$

This metric difference calculation process covers several corner or edge cases for the assessment of complexity that would be missed if metrics were run on individual versions. Retaining the initial complexity value of V_1 allows large or complex source code files to retain some aspect of complexity. Likewise, using the absolute value of the difference of calculated complexities allows for handling situations in which the complexity of a file decreases across revisions. This decrease accounts for the development effort applied towards the simplification of a source baseline, which can be a substantial effort. Handling this case is of interest to this praxis because substantial levels of effort can be applied to re-factoring an application to improve performance metrics. These efforts can result in a simpler code base. Only considering the complexity

metrics associated with the resultant code base might not accurately depict the complexity of the changes required to achieve this second revision.

3.4 The Analytical Hierarchy Process Model

Software complexity metrics are related such that efforts to achieve one set of quality attributes also stands a chance of either facilitating or hindering the achievement of another set of quality attributes that might be equally relevant in understanding the system. Consequently, it is important for engineering managers and systems engineers to have a process of identifying an acceptable set of software quality requirements and supporting design methodologies intended to meet such requirements. An ideal method for prioritizing quality attributes would be entirely quantitative and would address tradeoffs among design techniques during the process of quality attribute prioritization. This approach is often not the case because tradeoff analysis is between quality attributes of factors, is inherently qualitative, and does not adequately address design tradeoffs.

Given these conditions, this praxis made use of the AHP methodology to quantitatively prioritize software quality attributes pertaining to the measurement of complexity metrics associated with best practices in the Java object-oriented programming language. These best practices are represented through the factors of encapsulation, inheritance, coupling, cohesion, size, complexity, and polymorphism. The pair-wise comparison of these factors, when performed by subject matter experts, enables a weighing of the relative importance of these factors. This weighting is the subject matter expert's representation of the design and testing decisions that upholds the desired software quality attributes and enables the measurement of these factors through the use of the software complexity metrics of each factor.

This evaluation paradigm builds on Dromey's (1995) quality model, which has been extensively used for several software quality attributes as well as various testability models over time. To formalize the model in the context of the working thesis, the application of this model uses five phases, as represented in Figure 3-1.

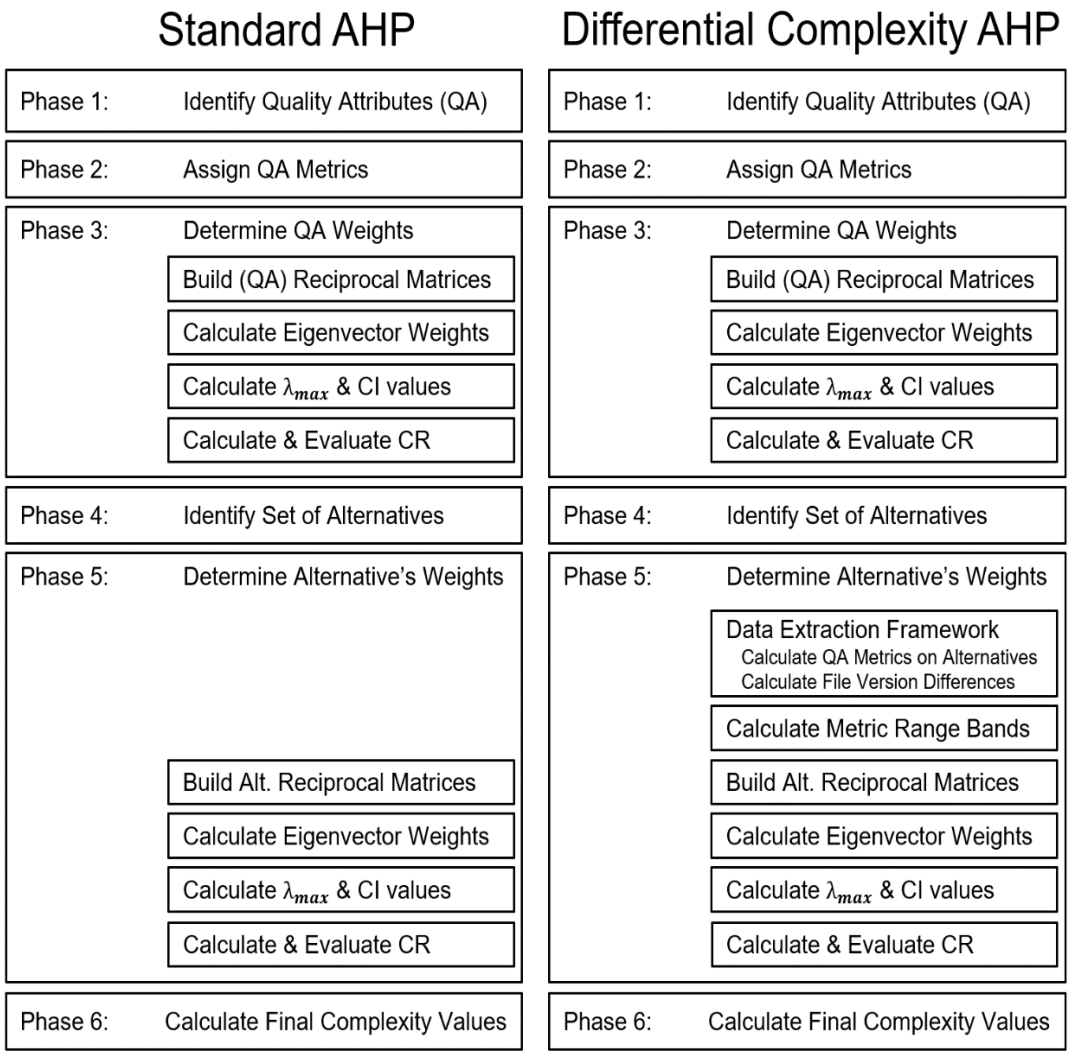

Figure 3-1: The standard AHP compared to Differential Complexity AHP.

3.4.1 AHP Phase 1: Define Evaluation Quality Attributes/Factors

The first phase involves defining the evaluation quality attributes or factors for the testability assessment of Java object-oriented software. A review of literature identified six attributes that define the testability of object-oriented software systems: inheritance,

polymorphism, coupling, encapsulation, cohesion, and size and complexity. These metrics have been accepted as standard attributes for the Java programming language by academic experts on the empirical validation of software systems and software engineers on "heavy usages and popularity" (Suri & Singhani, 2015, p. 77). This praxis uses these attributes in the same way as Suri and Singhani (2015). Figure 3-2 depicts the praxis AHP process concept of differential complexity as a decomposition of the six factors.

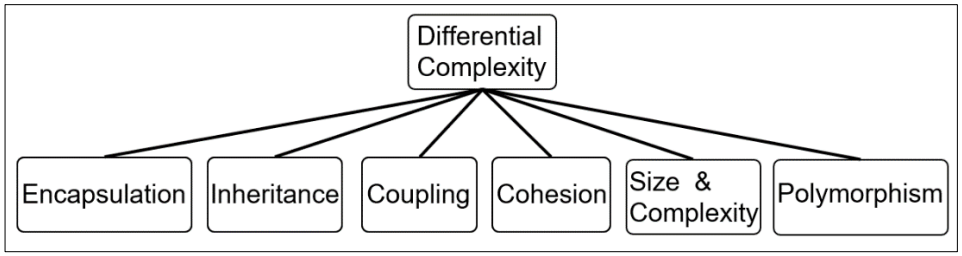

Figure 3-2: AHP differential complexity decomposition.

3.4.2 AHP Phase 2: Assign Factor Metrics

The second phase involved the selection of the applicable static metrics from the broad array of standard quality metrics for each. These factors are represented by their corresponding metric, as specified in Table 3-1. Figure 3-3 extends the AHP process concept of differential complexity in this praxis by relating each of the decomposed factors to a corresponding metric.

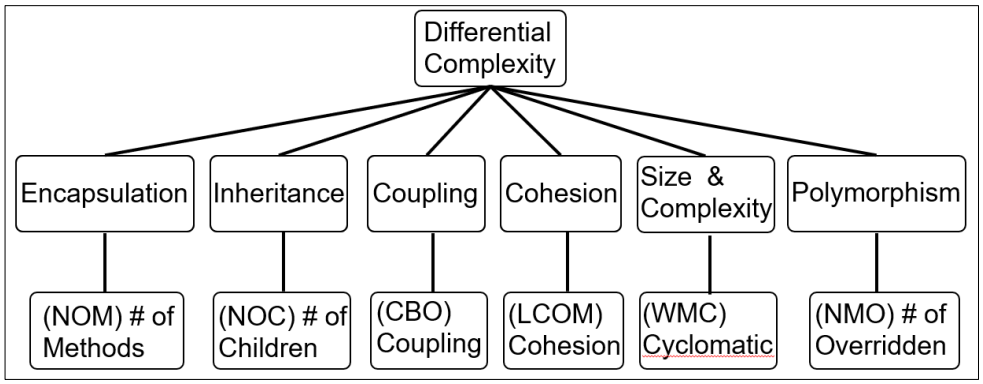

Figure 3-3: AHP differential complexity metric coupling.

3.4.3 AHP Phase 3: Determine Factor Weighting

The third phase involved the identification of factors and metrics with the process of determining the weights of the specified factors using a reciprocal matrix, which represents a pair-wise comparison made by quantitative or qualitative means of the factors under consideration. Also, this phase included the AHP calculation and verification process. Data for the pair-wise reciprocal matrix were drawn Suri and Singhani's (2015) conclusions, but the calculations performed as part of the AHP are enumerated for process transparency.

The AHP model calls for the division of the objective into a hierarchy of factors and subfactors that can be analyzed individually. After building the hierarchy in phases 1 and 2, the following five-step procedure is used to evaluate the problem.

3.4.3.1 Reciprocal Matrix Formation

The formation of the reciprocal matrix, which is a pair-wise evaluation matrix based on the identified factors, must be performed. In this process, every factor is compared with each other factor. The rubric for this pair-wise comparison is drawn from the Saaty's (1977, 1980) rating scale whose values range from 1 to 9 (see Table 3-3).

Table 3-3: Saaty's Rating Scale for the Analytical Hierarchy Process

Intensity of importance	Definition	Description
1	Equal importance	Elements Ci and Cj are equally significant
3	Weak importance of Ci over Cj	Both experience and evaluation slightly prioritize Ci over Cj
5	Strong or essential importance	Both experience and judgment strongly favor Ci over Cj
7	Demonstrated importance	Ci is very strongly favored over Cj
9	Absolute importance	The evidence that favors Ci over Cj is of the highest possible order of affirmation
2,4,6,8	Intermediate	Values between any two adjacent judgments are used when compromise is required.
Reciprocals of the above judgments	If Ci has one of the above judgments assigned to it relative to Cj, then Cj possesses the reciprocal value compared to Ci	A reasonable assumption

Drawing from the standard quality metrics suites mentioned by Suri and Singhani (2015), this praxis used six evaluation factors and their associated metrics (see Table 3-1): encapsulation, inheritance, coupling, cohesion, size, complexity, and polymorphism. The respective metrics are NOM (M_1), NOC (M_2), CBO (M_3), LCOM (M_4), WMC (M_5), and NMO (M_6). For factor F, if W_{ij} represents the relative weight of F_i F_i in relation to F_j, a reciprocal square matrix A of order equivalent to the number of factors in play is developed. This matrix is defined by $A = [W_{ij}]$ and is developed by the following equation, such that W_{ij} equals the reciprocal of W_{ji} and $W_{ii} = 1$. Note that i is not equal to j for all values of i. Equation 3.5 yields a reciprocal matrix.

$$A = {}^{F_1}_{F_n}W_{ij} = \begin{pmatrix} 1 & \cdots & W_{ij} \\ \vdots & \ddots & \vdots \\ 1/W_{ij} & \cdots & 1 \end{pmatrix} \quad (3.5)$$

Suri and Singhani (2015) performed this activity by surveying 10 professional academics performing research in object-oriented related subjects and processional software engineers with subject matter expertise in the field of object-oriented design and

development. The resulting mean matrix represents the evaluation factors. Validation of the results of these judgments is performed in Step 3.4.3.5 of the AHP process. That later validation step ensures that the pair-wise judgments captured in this step are consistent.

Table 3-4: Metric Pair-Wise Reciprocal Matrix and Eigenvector Weight Values

	M_1	M_2	M_3	M_4	M_5	M_6	6th root	Eigenvector
M_1	1.00	1.35	4.50	3.20	0.95	1.55	1.749	.221
M_2	1.23	1.00	4.30	3.30	1.07	1.40	1.722	.218
M_3	0.23	0.24	1.00	0.63	0.22	0.26	0.355	.044
M_4	0.32	0.31	1.95	1.00	0.27	0.31	0.502	.064
M_5	1.55	1.60	4.70	3.80	1.00	1.40	1.989	.256
M_6	0.95	1.10	3.90	3.30	1.10	1.00	1.567	.198
Total								1.0

3.4.3.2 Relative Weight Calculation

The next step in the AHP process is to determine the relative weights. This task is performed through the calculation of the Eigenvector from evaluation of the relative weights of each factor. There are many techniques for calculating this Eigenvector, and this praxis used the *n*th root of the product of values, which provided a good approximation of the correct answer. The Eigenvector is calculated by dividing the 6th root of each product of values for each metric by the sum of the 6th roots of all of the metrics. The resultant values are presented in Table 3-4. Comparison of the Eigenvector value column reveals the calculated relative weighting of the factors. M_1, M_2, and M_6 are similar, with values of .22, .22, and .20, respectively. M_5 is the most highly considered factor, with a weight of .26. M_3 and M_4 are the factors with the least impact, with weights of .04 and .06, respectively. These results were validated in the next step of the AHP by calculating the consistency scores.

3.4.3.3 Calculate AHP Consistency Basis (λ_{max})

The following three steps 3.4.3.3, 3.4.3.4, and 3.4.3.5 are part of the consistency calculation in the AHP process. The first step is the calculation of λ_{max}. This value provides the basis of consistency calculation for the AHP and enables calculation of the consistency index (CI) and consistency ratio (CR). To calculate λ_{max} for the first row in the matrix, the summation of the product of the pair-wise judgment and value are divided by the corresponding Eigenvector (see Equation 3.6).

$$M_1 = 1.00(.22)+1.35(.22)+4.50(.04)+3.20(.06)+0.95(.26)+1.55(.20) = 1.446 / .22 = 6.57 \quad (3.6)$$
$$M_2 = 1.23(.22)+1.00(.22)+4.30(.04)+3.30(.06)+1.07(.26)+1.40(.20) = 1.418 / .22 = 6.45$$
$$M_3 = 0.23(.22)+0.24(.22)+1.00(.04)+0.63(.06)+0.22(.26)+0.26(.20) = 0.290 / .04 = 6.59$$
$$M_4 = 0.32(.22)+0.31(.22)+1.95(.04)+1.00(.06)+0.27(.26)+0.31(.20) = 0.409 / .06 = 6.39$$
$$M_5 = 1.55(.22)+1.60(.22)+4.70(.04)+3.80(.06)+1.00(.26)+1.40(.20) = 1.649 / .26 = 6.34$$
$$M_6 = 0.95(.22)+1.10(.22)+3.90(.04)+3.30(.06)+1.10(.26)+1.00(.20) = 1.291 / .20 = 6.45$$
$$\lambda_{max} = (6.57 + 6.45 + 6.59 + 6.39 + 6.34 + 6.45) / 6 = 6.46$$

From the analytical hierarchy process theory, $A\omega = \lambda_{max}\omega$, $\lambda_{max} = n$, such that ω is the Eigenvector, λ_{max} is the Eigenvalue, and $\lambda_{max} >= n$ for a consistent matrix. From this calculation, 6.67 >= 6. This value indicates that the matrix is consistent.

3.4.3.4 Calculate the Consistency Index

The consistency index (CI) for the matrix is calculated using the formula in Equation 3.7. The CI value is used in the calculation of the consistency ratio (CR), which provides validation for the AHP technique.

$$CI = \frac{\lambda max - n}{n-1} = \frac{6.46-6}{6-1} = \frac{0.45}{5} = .09 \quad (3.7)$$

The CI score is an intermediary value used to perform the final calculation of the CR. Acceptable values may vary with respect to the number of alternatives in comparison, but will be greater than 0 and very likely less than .17. This is because the calculated CI value will be divided by a value taken from Saaty's random consistency index (RI) value

table where the table index is the number of items being compared. For a large number of items being compared the Saaty RI calculation technique goes asymptotic as it approaches a limit at .17. This would only be acceptable for a large number of compared items. For this case of six factors being compared the highest acceptable CI score would be .124. The calculated score of .09 is an acceptable value.

3.4.3.5 Calculate the Consistency Ratio

The final step in Phase 3 of the AHP process is to calculate the CR using both the previously calculated CI score and Saaty's average consistency index (RI) values shown in Table 3-5 using the formula shown in Equation 3.8. The judgments may be unreliable and inconsistent if the CR > 0.1. CR > 0.1 would denote a condition requiring the calculation of a new pair-wise comparison matrix. This process must continue until the condition CR < 0.1 is met. Once this condition is met, the AHP can be applied to predict verification and validation decisions based on the available alternatives.

$$CR = \frac{CI}{RI} = \frac{.09}{1.24} = .07 \qquad (3.8)$$

Saaty's (2008) consistency index (RI) is provided as a lookup in Table 3-5. There are six factors being compared, so the RI value used is 1.24.

Table 3-5: Saaty's Consistency Index

# factors	1	2	3	4	5	6	7	8	9	10
RI value	0.0	0.0	0.58	0.90	1.12	1.24	1.32	1.41	1.45	1.49

From the calculations performed using Equation 3.8, the consistency ratio was found to be .07, which is below the recalculation threshold of .1. As such, it is appropriate to accept the weighted values provided by the Eigenvectors of the pair-wise comparison matrix. The weighed factors of the pair-wise factor matrix of software quality attributes

of encapsulation, inheritance, coupling, cohesion, size, complexity, and polymorphism are .22, .22, .04, .06, .26, and .20, respectively.

3.4.4 AHP Phase 4: Identify Alternatives

The fourth phase of the AHP model involves defining the set of alternatives to be compared using the weighted set of factors. These alternatives include the superset of files of the two subsequent revisions of the software project code. As described in Section 3.2, data for this praxis were collected from four open-source Java software projects from the Apache Commons suite. Each project required an independent execution of the AHP to determine the complexity of the changes for the files in the subsequent revisions. Figure 3-4 extends the AHP process concept of differential complexity explained in this praxis by incorporating the comparison of the identified set of alternatives. The comparison of alternatives is done for each of the decomposed factors using the specified corresponding metric.

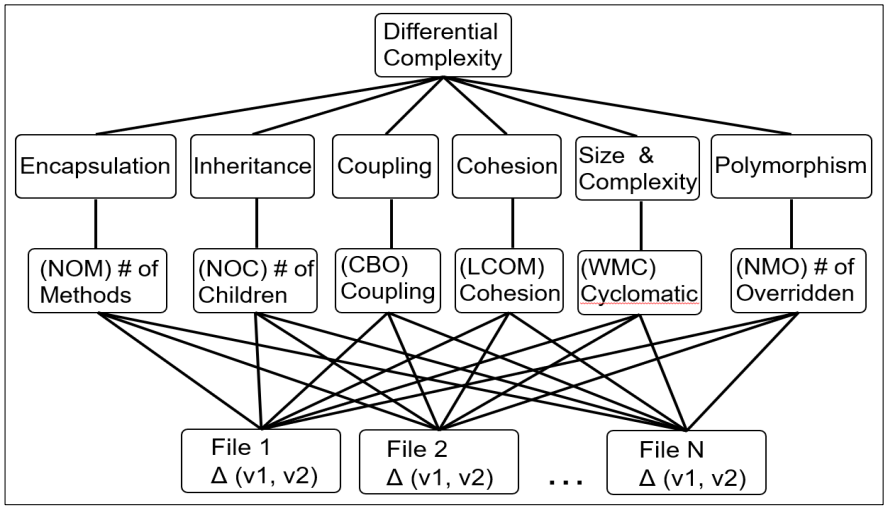

Figure 3-4: AHP differential complexity alternatives identification.

3.4.5 AHP Phase 5: Determine Alternative Weights

This phase of the AHP implementation involves quantitatively calculating a reciprocal matrix, which represents the pair-wise comparison of the alternatives under consideration for each of the identified factors. This phase consumes the output of the data extraction framework from Section 3.2 of this praxis, and performs the steps of the AHP calculations. This includes the creation of the pair-wise comparison reciprocal matrices, development of Eigenvectors, and verification steps of calculating the consistency index (CI) and consistency ratio (CR). The process of performing AHP calculations is identical in process to that described as Phase 3 in Section 3.3.3, but several steps are required to process the data collected from the data extraction framework into pair-wise comparison reciprocal matrices before these calculations can be performed. This phase of the AHP process requires a six-step process be performed for each of the identified metrics such that P_k is one of the projects specified in Section 3.2 Table 3-2. Each P_k contains a set of files F_1 to F_n. Each of these files has two revisions, denoted as F_{n1} or F_{n2}. There are six metrics denoted as M_q, where *q* is the number corresponding to the related software quality attribute or factor. For example, the Apache Commons Net project is P_2 and contains 250 Java source files, which would be denoted as F_1 to F_{250}.

3.4.5.1 Calculation of Range Bands

The first step is to determine the range bands for the basis of comparison between the calculated metrics. The basis for comparison for each reciprocal matrix is the calculated difference value between the first F_{n1} and second F_{n2} revisions of each metric M_q for each file F_n in the specified project P_k. From Section 3.3 and described again below, the data

extraction framework calculates a difference value, V_4, where V is the value number, M_q is the metric function, and F_{n1} is the first file revision and F_{n2} is the second file revision (see Equation 3.9).

$$V_4 = M_q(F_{n1}) + \left|M_q(F_{n2}) - M_q(F_{n1})\right| \quad (3.9)$$

The AHP calculations are performed on the pair-wise comparison matrix, which is in the form of an integer between 1 and 9 or the reciprocal of this integer value, according to Saaty's rating scale (see Table 3-3). The calculated difference values from the data extraction framework are in the output form of the metric as an integer >= 0 and must be converted to the format useable by the AHP calculation.

The following processes are performed. All of the V_4 values for files F_n in project P_k are included in the multiset S such that the V_4 value x for each F_n is included in S if and only if the V_4 is greater than zero and not a statistical upper outlier. A multiset is a set that can contain duplicate values (see Equation 3.10).

$$S = \{x \mid x > 0, x < \textit{upper outlier}\} \quad (3.10)$$

Outliers are calculated using the Tietjen-Moore test for upper outliers using the implementation described in Flanagan's (2015) Java scientific library. The Tietjen-Moore algorithm is a generalization of the Grubbs outlier test and uses a critical value that is generated through multiple runs (10,000 by default) of a randomly normal distribution comprising *T* data points, where *T* equals the number of values in the input data. Removing the outliers limits the overall range of the metric value set to a trimmed statistical range and prevents outliers from creating overly large range bands. Once the outliers are removed, the range of the set is determined from 0, the minimum value for a complexity metric difference value, to the maximum value of the set. This set range is

divided into eight equal range bands, R_n, which correspond to the values 1 through 8 in Saaty's (2008) rating scale. The value of 9 in Saaty's rating scale is reserved for outlier comparisons.

3.4.5.2 Reciprocal Matrix Calculation

The second step is the formation of the reciprocal matrix, which is a pair-wise evaluation matrix for each file F_n in project P_k. In this case, the comparisons are performed on the differences in the V_4 value for each F_n, including those F_n whose V_4 values are statistical outliers. For metric M_q with calculated range band R_b applied to project P_k, if W_{ij} represents the relative weight of file F_i in relation to file F_j, a reciprocal square matrix A of order equivalent to the number of factors in play is developed. W_{ij} is calculated as shown in Equation 3.11.

$$integer\ value\ \lfloor 1 \rfloor < W_{ij} < \lceil 9 \rceil = \frac{(V_{4i} - V_{4j})}{R_b} \qquad (3.11)$$

This calculation converts the comparison in metric values into the appropriate Saaty rating scale value. When files i and j are compared pair-wise, non-outlier V_4 values result in a Saaty rating scale value of 1 if the $V4$ values are within a single range band and a value of 2 if they are within two range bands. This calculation continues through the value 8. If an outlier is involved in the comparison, the difference may be greater than nine range bands. In these cases, the Saaty rating scale ceiling value of 9 is selected.

This matrix is defined by $A = [W_{ij}]$ and is developed as shown in Equation 3.12, such that W_{ij} equals the reciprocal of W_{ji} and $W_{ii} = 1$. Note that i is not equal to j for all values of i.

$$A = {}^{F_1}_{F_n}W_{ij} = \begin{pmatrix} 1 & \cdots & W_{ij} \\ \vdots & 1 & \vdots \\ 1/W_{ij} & \cdots & 1 \end{pmatrix} \quad (3.12)$$

3.4.5.3 Relative Weight Calculation

Having defined the reciprocal matrix, the next step in the AHP process is to determine the relative weights. This process is identical to that used in Phase 3, Step 2. In this step, the Eigenvector is calculated from evaluation of the relative weights of each factor. Again, this praxis makes use of the *n*th root of the product of values technique to calculate this Eigenvector. In this case, *n* equals the number of files in project P_k. The Eigenvector is calculated by dividing the *n*th root of each product of values for each metric by the sum of the *n*th roots of all the metrics (see Equation 3.13).

$$EV_{F_1} = \frac{\sqrt[n]{\prod_1^n w_{i1}}}{\sum_1^n \sqrt[n]{\prod_1^n w_{ij}}} \quad (3.13)$$

3.4.5.4 Calculation of λ_{max}, CI, and CR

Steps 4 through 6 are the verification and validation portions of the AHP. These calculations are performed in an identical manner as those in sections 3.4.3.3, 3.4.3.4, and 3.4.3.5, respectively. The validation values of λ_{max}, CI, and CR are calculated for the sets of alternatives are provided in Results, Section 4.2.

3.4.6 AHP Phase 6: Calculate Final Values

The sixth and final phase is to calculate the final weighted values for each factor alternative combination and to rank them. This phase applies the weighted factors for the calculated metrics of each alternative and establishes the relationship between the specified quality attributes of the software system and the metrics or attributes identified in the preceding step. The ranked hierarchy of alternatives from this final phase is the

output of the AHP in the form of an ordered multiset of files and combined complexity pairings for each file F_n in project P_k. Examples of the calculations and output are shown in Table 4-1. Additional examples of a subset of the output for all projects are presented in the aggregate results section in tables 4-14, 4-15, 4-16, and 4-17. The calculation is performed shown in Equation 3.14.

C_{F_n} = *combined complexity value for file* F_n *(3.14)*
$F_n = file\ n$
$M_1 – M_6$ = *metrics 1 – 6*
WF_{M_q} = *eigenvalue weight factor for metric q as from 3.4.3*
$M_qEV_{F_n}$ = *eigenvalue weight factor for metric q on file n as from 3.4.5.5*

$$C_{F_n} = \left(M_1EV_{F_n} * WF_{M1}\right) + \left(M_2EV_{F_n} * WF_{M2}\right) + \left(M_3EV_{F_n} * WF_{M3}\right) + \left(M_4EV_{F_n} * WF_{M4}\right) + \left(M_5EV_{F_n} * WF_{M5}\right) + \left(M_6EV_{F_n} * WF_{M6}\right)$$

3.5 Efficiency of the Method

Four factors impact the efficiency and robustness of the AHP model. The first is the assumption that the value of preference meets the reciprocal condition when two factors are compared (Belton & Gear, 1983). The second is that the comparison of importance is made on a bounded scale if there is no guarantee of a limited range of the compared qualities. The third is that elements at a lower level depend on elements at upper levels. The fourth assumption is that all objectives and goals of the analytical process are included in the corresponding levels (Belton & Gear, 1983).

Also, there are weaknesses in practical application of the AHP model. For instance, adding new entities or attributes for comparison to the AHP might inadvertently alter priorities, reducing reliability of results of the process (Belton & Gear, 1983; Harrison 1992). Another area of potential concern is proportionality between the measures of importance of the referent factors or elements. There can be no guarantee of consistency of these measurements because comparisons in the AHP model are made using a 9-point

scale as well as their corresponding reciprocals. Decision makers might have difficulty differentiating between middle values on Saaty's scale. Lastly, the consistency ratio (CR) value in AHP is suggested to ensure the consistency of responses, but the result is considered dependable only when CR values do not exceed 0.1. Satisfying the requirement for $CR < 0.1$ is difficult to achieve because an increase in the number of attributes or entities to compare increases the difficulty in maintaining $CR < 0.1$ (Harrison 1992). Because priorities are used for the CR values, biases might influence priorities before the comparisons are made to ensure that the CR value is maintained within 0.1.

Chapter 4 – Results

This chapter provides the results of the study and answers to the primary research questions. The chapter begins with a detailed examination of a representative set of files from the Apache Commons Net project that is intended to demonstrate how the techniques described in this praxis are applied and results calculated on individual Java source code files. Following the individual results, the aggregate project results and the final complexity value outputs are presented for each of the projects identified in Section 3.2. Finally, a discussion of the results focused on analysis of a graphical representation included as Figure 4-1 is performed.

4.1 Representative Individual Results

The assessment model was applied to four object-oriented software programs (P_1, P_2, P_3, and P_4) obtained from public repositories, as described in Section 3.2. The procedures outlined in sections 3.3 and 3.4 are described in detail in the following section for the org.apache.commons.net.ftp.FTPClient from the Apache Commons Net project (P_2). This class was selected because the model identified it as the most complexly modified file in the Apache Commons Net project.

First, the data extraction framework was used to extract metric data about the sample class. The obtained data are presented in Table 4-1.

Table 4-1: Calculated Metric Data for org.apache.commons.net.ftp.FTPClient

FTPClient metric data collected by data extraction framework	Version 1 (V_1) raw metric score from FTPClient version 1	Version 2 (V_2) raw metric score from FTPClient version 1	Version Δ (V_3): $V_3 = \|V_2 - V_1\|$	Complexity difference (V_4): $V_4 = V_1 + V_3$
Number of methods (NOM)	141	145	4	145
Number of children (NOC)	3	3	0	3
Coupling between objects (CBO)	23	21	2	23
Level of cohesion (LCOM)	9426	9986	560	9986
Cyclomatic complexity (WMC)	383	393	10	393
Number of methods overridden (NMO)	5	5	0	5

Once the metric data have been calculated by the data extraction framework, the next step in the process is to determine the range bands for pairing and comparison between the calculated metrics. In this second step, the difference values obtained from the previous calculation are presented in the pair-wise comparison or evaluation matrix for every file in the project. For the NOM metric, the non-zero calculated values for V_4 , x, are added to multiset S and the outliers are removed using the Tiegen-Moor algorithm for identifying upper outliers. The histogram in Figure 4-1 shows the counts of the V_4 metrics when grouped into bins of 3.5, which is the calculated width of the range band for this metric. The outliers are identified as the rightmost (ninth), bin which contains 14 calculated V_4 values above 28.0. Values in this bin comprise the set of outliers.

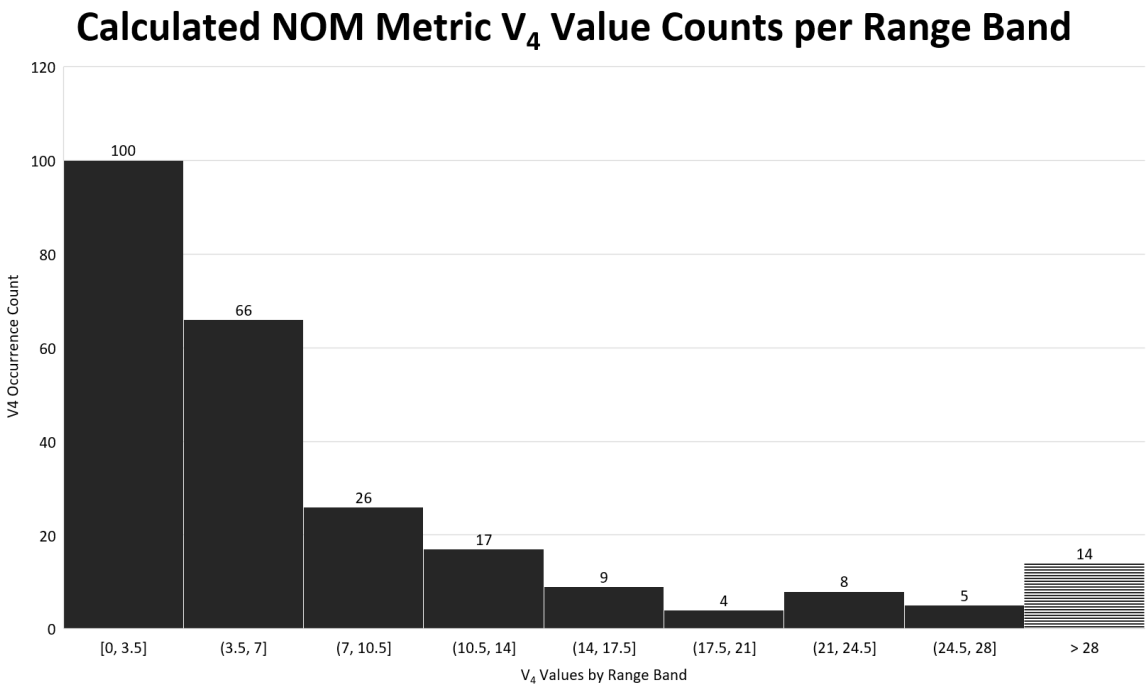

Figure 4-1: Apache Net project NOM metric V4 values per range band.

For the NOM metric data set, a total of 14 outliers were detected and not considered. The remaining values provide a set range between the metric floor of 0.0 and 28.0. Dividing this value by 8 for the number of discrete values in the AHP yields a calculated range band of 3.5. Note that 8 was used for the number of AHP values to preserve the maximum AHP difference value of 9, according to Saaty's rating scale, for use by outliers if any are present.

In the third step, the relative weights of each factor are determined using the reciprocal matrix, defined above by calculating the Eigenvector values from the evaluation of the relative weights of each factor, as presented in Table 4-2. There are 250 Java source files in the Apache Commons Net (P_2) project; the resulting reciprocal pair-wise comparison matrix is [250x250]—too large to include in this praxis. A subsection of this reciprocal matrix for the NOM metric is included in Table 4-2.

Table 4-2: Subsection of Reciprocal Matrix for NOM Metric

Java class	FTPClient	FTPSClient	FTP	Telnet	…	POP3Client
FTPClient	1	9	9	9		9
FTPSClient	.111111	1	.125	1		9
FTP	.111111	8	1	9		9
Telnet	.111111	1	.111111	1		9
…					1	
POP3Client	.111111	.111111	.111111	.111111		1

The subsection of the reciprocal matrix represented in Table 4-2, created from the pair-wise comparisons, includes several complexly changed files and several that were not as heavily modified. From the full matrix summarized in Table 4-2, the Eigenvectors for the NOM metric are calculated using the 250th root of the product of values for each row using the calculations specified in Section 3.4.5.2 of the AHP. The 250th root was used because there are 250 files in the Apache Common Net project. The results are provided in Table 4-3, along with the associated Eigenvector weight and the NOM Nth root and NOM judgment values used to calculate the NOM Eigenvector weights (see Equation 4.1).

$$EV_{F_1} = \frac{\sqrt[250]{\prod_1^{250} W_{i1}}}{\sum_1^{250} \sqrt[250]{\prod_1^{250} W_{ij}}} \qquad (4.1)$$

Table 4-3: Subset of NOM *n*th Root, Eigenvector, and Judgment Values

Java class	NOM *n*th root	NOM eigenvector	NOM judgment
FTPClient	8.921	0.0253	8.797
FTPSClient	7.910	0.0224	6.612
FTP	8.680	0.0246	8.335
Telnet	7.818	0.0221	6.437
…			
POP3Client	2.158	0.006	1.572

Having calculated the Eigenvector values weights, the next step is to validate the weights by determining the consistency index (CI) and consistency ratio (CR). According to the AHP theory, $A\omega = \lambda_{max}\omega$, $\lambda_{max} = n$, such that ω is the Eigenvector, λ_{max} is the

Eigenvalue, and λ_{max} > = n for a consistent matrix. Based on the results, there is some consistency with the sample of the reciprocal matrix. The consistency of FTPClient is more complex than the other files in the sample; correspondingly, the same file has higher Eigenvector and judgment values. The summation of the judgment for the NOM metric in the Apache Commons Net project was 64244.858, and there are 250 Java source files. This summation provides a λ_{max} of 256.979, which is >= 250 and indicates good reciprocal matrix consistency. The next step is to calculate the CI and CR (see Equation 4.2).

$$CI = \frac{\lambda max - n}{n-1} = \frac{256.979-250}{250-1} = \frac{6.979}{249} = .017 \quad (4.2)$$

Because Saaty's original random index (RI) table does not extend to the degree needed, this praxis used Donegan and Dodd's (1991) extended scales, which extend Saaty's original scales (Franek & Kresta, 2014). The closest RI value below the n of 250 is n = 100, which provides a RI value of 1.6339. Although it is possible to calculate a RI score specific to the number of alternatives, this calculation was not considered essential for the praxis. The calculated RI score increases with respect to the number of alternatives being considered. Because the RI value is the divisor, the use of a RI value corresponding to a lower number of alternatives inflates the CR score. If the resulting CR score is below the AHP consistency threshold of a CR <= .1, there is no benefit to calculating a specific RI value. Because the identification of an RI value corresponds to n = 100 on the extended scale, it is possible to determine the CR value for the NOM metric in the Apache Commons Net project, as shown in Equation 4.3.

$$CR = \frac{\text{CI}}{\text{RI}} = \frac{.017}{1.6339} = .010 \quad (4.3)$$

The results of the CR calculation of .010 is well below the CR recalculation threshold of CR <= .1. As such, it is appropriate to accept the weighted values provided by the Eigenvectors of the pair-wise comparison matrix. The weighed factors of the pair-wise factor matrix of alternatives can be accepted for the NOM metric. This process is repeated for the other five metrics, and the results are calculated and shown in Table 4-4.

Table 4-4: Sample Class Eigenvector Weight values for Identified Metrics

Class	NOM EV	NOC EV	CBO EV	LCOM EV	WMC EV	NMO EV
FTPClient	0.025	0.024	0.027	0.021	0.025	0.009
FTPSClient	0.022	0.003	0.005	0.020	0.022	0.018
FTP	0.025	0.008	0.008	0.021	0.022	0.005
Telnet	0.022	0.008	0.008	0.019	0.024	0.002
POP3Client	0.006	0.008	0.005	0.013	0.015	0.003

The λ_{max}, CI, and CR scores for all of metrics in the Apache Commons Net project were calculated and are provided in Table 4-5. These scores are calculated at the project level for each metric to provide consistency for all evaluation in that project, for that metric. All of the CR values were well under the CR >= .1 recalculation threshold. As such, the Eigenvector weight values can be accepted and the final weighting of complexity can be calculated for the files (see Table 4-5).

Table 4-5: Calculated CR, CI, and λmax Values for Selected Apache Commons Net Project

Metric	λ_{max}	CI	CR
NOM	256.979	0.028	0.017
NOC	250.689	0.003	0.002
CBO	256.281	0.025	0.015
LCOM	270.266	0.081	0.049
WMC	257.620	0.030	0.019
NMO	251.560	0.006	0.004

In the final step, the ranked and final weighted values for each evaluation factor are calculated. The prioritized hierarchy of alternatives developed from this step is the output of the AHP in the form of an ordered set of files and corresponding complexity values in a key-value pairing. This set of files contains entries for each file F_m in the Apache Commons Net project. This step combines the Eigenvectors of weighted criteria factors from Table 4-6 with the calculated Eigenvector values for each alternative file.

Table 4-6: Metric Pair-wise Reciprocal Matrix, 6th Root and Eigenvector Weight Values

	M_1	M_2	M_3	M_4	M_5	M_6	6th root	Eigenvector
M_1	1.00	1.35	4.50	3.20	0.95	1.55	1.749	.221
M_2	1.23	1.00	4.30	3.30	1.07	1.40	1.722	.218
M_3	0.23	0.24	1.00	0.63	0.22	0.26	0.355	.044
M_4	0.32	0.31	1.95	1.00	0.27	0.31	0.502	.064
M_5	1.55	1.60	4.70	3.80	1.00	1.40	1.989	.256
M_6	0.95	1.10	3.90	3.30	1.10	1.00	1.567	.198
Total								1.0

The final complexity value for each file alternative in the Apache Commons Net project can be calculated using the formula shown in Equation 4.4.

C_{F_n} = *combined complexity value for file* F_n *(4.4)*
$F_n = file\ org.apache.commons.net.ftp.FTPClient$
$M_1 – M_6$ = *metrics 1 – 6*
WF_{M_q} = *eigenvalue weight factor for metric q as from 3.4.3*
$M_qEV_{F_n}$ = *eigenvalue weight factor for metric q on file n as from 3.4.5.5*

$$C_{F_n} = \left(M_1EV_{F_n} * WF_{M1}\right) + \left(M_2EV_{F_n} * WF_{M2}\right) + \left(M_3EV_{F_n} * WF_{M3}\right) + \left(M_4EV_{F_n} * WF_{M4}\right) + \left(M_5EV_{F_n} * WF_{M5}\right) + \left(M_6EV_{F_n} * WF_{M6}\right)$$

Substituting the values for the FTPClient class yields the final complexity value (see Equation 4.5).

$$C_{F_m} = (.025 * .22) + (.024 * .22) + (.027 * .04) + (.021 * .06) + (.025 * .26) + (.009 * .20) = .02158 \quad (4.5)$$

Table 4-7 includes the results for the sample set of classes.

Table 4-7: Calculated Software Difference Based on Complexity Values for Sample Classes

Class	Complexity value
FTPClient	0.0215803
FTPSClient	0.0162935
FTP	0.0155419
Telnet	0.0150902
POP3Client	0.0085537

4.2 Aggregate Project Results

Table 4-8 presents the aggregate data as the output of the AHP modeling process upon data derived from the data extraction framework. Table 4-8 is a duplicate of Table 3-3, repeated here for convenience, and provides some initial hueristics that are referenced in these results.

Table 4-8: Project 1-4 Version Heuristics

Project	V_1 #Files	V_2 #Files	#Changed	#Added
Discovery (P_1)	47	49	47	2
Net (P_2)	248	250	30	2
IO (P_3)	198	227	165	29
Compression (P_4)	282	314	51	32

The values in tables 4-8, 4-9, 4-10, 4-11, 4-12, and 4-13 were calculated using the λ_{max}, CI, and CR according to the procedure outlined in Section 4.1. The first value is the Eigenvector weight value, which was calculated on a per-file basis for each metric. This value represents the relative weight of a given file in a project for the current metric, as compared to other files in the same project. Higher values for the Eigenvector weight correspond with greater relative importance. The other important value to consider is the CR score, which is calculated on a per-project basis. According to the AHP, the CR value

must be less than .1; a CR value of less than .1 indicates the pair-wise comparison matrix is internally consistent and recalculation is not needed.

Also included in tables 4-8, 4-9, 4-10, 4-11, 4-12, and 4-13 is the count of the number of outliers for the calculated metric values for the project/metric pairing, as well as the calculated value of the range band. The number of outliers value was determined as explained in Section 3.4.5.1 and is the number of values removed for a given metric range band calculation. The outliers were calculated using the same Tietjen-Moore algorithm, which is a generalization of the Grubbs outlier test. The identification of the outliers used, as an example, to calculate the NOM metric for the Apache Commons Net project had 14 statistical outliers from a total of 250 Java source files. The outliers are temporarily discarded, and the range of the data set for the NOM metric was determined. This set range is divided by the number of scoring vales from Saaty's rating table minus 1, which is reserved for outlier use. The process for these calculations was described in the calculate range band discussion in Section 3.4.5.1.

The outlier and range band values provide insights into the types of changes made to a code base between revisions by examining how many classes experience statistically significant changes in the complexity of a specific metric. Table 4-8 indicates changes in the Apache Commons Compression project. There are a moderate number of outliers—10—for the NOM metric, a measure of the encapsulation quality attribute. The same project experienced almost double the number of outliers for the WMC metric, a measure of size and algorithmic complexity. A qualitative comparison indicates the Apache Commons Compression project, in aggregate, experienced more change in algorithmic complexity than encapsulation.

The calculated NOM metric outlier counts, range bands, and consistency values for the four selected projects are shown in Table 4-9.

Table 4-9: NOM Metric Outliers Count, Range Band, and Consistency Values for Project 1-4

NOM values	Outliers	Range band	λ_{max}	CI	CR
Discovery (P_1)	1	2.125	49.895	0.018	0.011
Net (P_2)	14	3.5	256.979	0.028	0.017
IO (P_3)	15	3.875	234.694	0.020	0.034
Compression (P_4)	10	6	319.794	0.011	0.018

The calculated NOC metric outlier counts, range bands, and consistency values for the four selected projects are shown in Table 4-10.

Table 4-10: NOC Metric Outliners Count, Range Band, and Consistency Values for Project 1-4

NOC values	Outliers	Range band	λ_{max}	CI	CR
Discovery (P_1)	0	0.125	49.000	1.480E-16	8.971E-17
Net (P_2)	3	0.375	250.689	0.003	0.002
IO (P_3)	1	1.0	227.110	4.879E-4	2.957E-4
Compression (P_4)	1	1.125	314.115	3.681E-4	2.231E-4

The calculated CBO metric outlier counts, range bands, and consistency values for the four selected projects are shown in Table 4-11.

Table 4-11: CBO Metric Outliers Count, Range Band, and Consistency Values for Project 1-4

CBO values	Outliers	Range band	λ_{max}	CI	CR
Discovery (P_1)	1	1.625	50.225	0.026	0.015
Net (P_2)	8	1.125	256.281	0.025	0.015
IO (P_3)	3	1.5	233.083	0.027	0.016
Compression (P_4)	4	2.5	319.697	0.018	0.011

The calculated LCOM metric outlier counts, range bands, and consistency values for the four selected projects are shown in Table 4-12.

Table 4-12: LCOM Metric Outliers Count, Range Band, and Consistency Values for Project 1-4

LCOM values	Outliers	Range band	λ_{max}	CI	CR
Discovery (P_1)	4	6.5	51.177	0.045	0.027
Net (P_2)	35	8.875	270.266	0.081	0.049
IO (P_3)	21	23.25	238.303	0.050	0.030
Compression (P_4)	31	24.375	328.612	0.047	0.028

The calculated WMC metric outlier counts, range bands, and consistency values for the four selected projects are shown in Table 4-13.

Table 4-13: WMC Metric Outliers Count, Range Band, and Consistency Values for Project 1-4

WMC values	Outliers	Range band	λ_{max}	CI	CR
Discovery (P_1)	1	6.375	50.154	0.024	0.015
Net (P_2)	14	9.25	257.620	0.030	0.019
IO (P_3)	9	10.125	233.493	0.028	0.017
Compression (P_4)	19	13.25	322.521	0.016	0.027

The calculated NMO metric outlier counts, range bands, and consistency values for the four selected projects are shown in Table 4-14.

Table 4-14: NMO Metric Outliers Count, Range Band, and Consistency Values for Project 1-4

NMO values	Outliers	Range band	λ_{max}	CI	CR
Discovery (P_1)	0	0.75	49.793	0.016	0.010
Net (P_2)	1	1.375	251.560	0.006	0.004
IO (P_3)	4	2.5	229.806	0.012	0.007
Compression (P_4)	2	6.75	314.774	0.002	0.001

4.3 Project Calculated Complexity Differential Values

The following data represent the final outputs of the differential complexity AHP described in this praxis. The complexity values were calculated for each Java source file in each project using the process and corresponding equations from Section 3.4.5.5. The resulting complexity values were used to sort the files according to complexity value on a per-project basis. Files with the highest complexity score represent the most complexly

modified files within that project. Because the AHP uses a pair-wise comparison technique, it is not possible to compare the resulting complexity values across projects. The Eigenvector weights for each Java source file were generated from a matrix that only includes the source files from the single project. To compare complexity across projects, the AHP described in this praxis should be performed on a single combined project.

Tables 4-14, 4-15, 4-16, and 4-17 represent the final prioritized list of complexity scores for a given project, as calculated by the differential complexity AHP described in Section 3.4. This list, whether used for manual or automated testing, allows an engineering manager or test team to focus available testing resources on those portions of the project baseline that underwent the most complex changes. Tables 4-14, 4-15, 4-16, and 4-18 are edited for brevity to identify the seven files with the highest calculated differential complexity and project file with the lowest level of differential complexity. The full results are included in the Appendix. Also included in these tables are indicators of which complexity scores are statistical upper outliers. Outliers were calculated using the Tietjen-Moore algorithm, which is a generalization of the Grubbs outlier test.

Table 4-15: Discovery (P_1) Most and Least Differentially Complex Files, with Outliers

Apache Commons discovery class	Complexity
org.apache.commons.discovery.log.SimpleLog	*0.072622375
org.apache.commons.discovery.resource.names. DiscoverNamesInFile	*0.054651828
org.apache.commons.discovery.tools.DiscoverClass	0.048795669
org.apache.commons.discovery.tools.ManagedProperties	0.045799577
org.apache.commons.discovery.resource.classes. ResourceClassDiscoverImpl	0.039344126
org.apache.commons.discovery.test.TestAll	0.036988141
org.apache.commons.discovery.Resource	0.035631755
… 39 omitted class results	2 outliers / 47 total
org.apache.commons.discovery.test.TestInterface1	0.010720913
Total	1.0

Note. * Statistical upper outlier.

Table 4-16: Net (P_2) Most and Least Differentially Complex Files, with Outliers

Apache Commons net class	Complexity
org.apache.commons.net.ftp.FTPClient	*0.021580391
org.apache.commons.net.ftp.FTPSClient	*0.016293543
org.apache.commons.net.ftp.FTP	*0.01554193
org.apache.commons.net.telnet.Telnet	*0.015090213
org.apache.commons.net.nntp.NNTPClient	*0.014911443
org.apache.commons.net.SocketClient	*0.014675531
org.apache.commons.net.util.Base64	*0.012940319
… 242 omitted class results	11 outliers / 250 total
org.apache.commons.net.nntp.ArticlePointer	0.002060035
Total	1.0

Note. * Statistical upper outlier.

Table 4-17: IO (P_3) Most and Least Differentially Complex Files, with Outliers

Apache Commons IO class	Complexity
org.apache.commons.io.FileUtilsTestCase	*0.016903606
org.apache.commons.io.FileUtils	*0.016316664
org.apache.commons.io.IOUtils	*0.016117173
org.apache.commons.io.filefilter.FileFilterTestCase	*0.015649916
org.apache.commons.io.IOUtilsTestCase	*0.015330946
org.apache.commons.io.serialization. ValidatingObjectInputStreamTest	*0.014415967
org.apache.commons.io.testtools.TestUtils	*0.014337154
… 219 omitted class results	27 outliers / 227 total
org.apache.commons.io.filefilter.IOFileFilter	0.002379585
Total	1.0

Note. * Statistical upper outlier.

Table 4-18: Compression (P_4) Most and Least Differentially Complex Files, with Outliers

Apache Commons compression class	Complexity
org.apache.commons.compress.compressors.lz4. BlockLZ4CompressorOutputStream	*0.01598046
org.apache.commons.compress.compressors.lz4. FramedLZ4CompressorInputStreamTest	*0.014487463
org.apache.commons.compress.compressors.lz77support. LZ77Compressor	*0.012797215
org.apache.commons.compress.archivers.zip. Zip64SupportIT	*0.011897318
org.apache.commons.compress.archivers.zip. ZipArchiveOutputStream	*0.011321303
org.apache.commons.compress.compressors.lz4. FramedLZ4CompressorInputStream	*0.010814482
org.apache.commons.compress.archivers.tar. TarArchiveEntry	*0.010788506
… 306 omitted class results	40 outliers / 314 total
org.apache.commons.compress.compressors.bzip2. BZip2Constants	0.0019411362
Total	1.0

Note. * Statistical upper outlier.

4.4 Results Discussion

The CR values of all metrics were below the recalculation threshold of CR >=.1, implying that the Eigenvector weight values are acceptable, allowing the calculation of the final differential complexity values for each of the analyzed files.

As shown in tables 4-14, 4-15, 4-16, and 4-17, the summation of the differential complexity values for the Apache Commons Discovery (P_1), Net (P_2), IO (P_3), and Compression (P_4) projects is 1.0 and does conform to the expected result.

When the resulting ordered differential complexity values are overlaid on a graph, there are several similarities (see Figure 4-1). The Discovery project has fewer Java source files; therefore, the complexity values are greater on a per-file average because the summation of the complexity values of all files across a project is equal to 1.0. The Net, IO, and Compression projects are composed of a similar numbers of files, so the plots for these projects follow the same path on the y-axis in Figure 4-1. The plot line for the Discovery is positioned higher on the y-axis because of the greater per-file complexity values resulting from the Discovery project having a lower number of files.

Another interpretation of this graph is based on the asymptotic nature of the plots. The interpretation is based on the number of files changed or added in each project (see Table 4-1). For the plot of each project, the greatest rate of differential complexity increase occurs in the portions of the x-axis plot between 0 and the number of changed or added files for the given project. With the exception of the Net project, the graph values from the point on the x-axis greater than the number of files changed or added contains the portions of the graph that have become asymptotic. For example, according to Table 4-1, the Compression project has 314 total files, but only 83 were changed or newly

added. In Figure 4-1, the plot between 0 and 83 on the x-axis reflects most of the y-axis complexity value increases. The portions of the plot greater than x-axis value 83 are asymptotic. This behavior is consistent with software changes in a project over time as some of the baseline is heavily modified, some slightly modified, and some portions of the software baseline might undergo no change.

The complexity plot of the Net project warrants additional attention. According to Table 4-1, the Net project contains 250 files, but only 32 were changed or modified. Similar to the Compression project, the greatest increases in the y-axis differential complexity values occurred between 0 and 32 on the x-axis. In addition to these increases, the Net project also experienced some y-axis differential complexity increases between the range of 32 and 105 on the x-axis. As noted in the metric scores in tables 4-8, 4-9, 4-10, 4-11, 4-12, and 4-13, the Net project had moderate to high comparative differential complexity scores for NOM, NOC, CBO, LCOM, and WMC metrics. The high degree of complexity in the Net project kept y-axis complexity values high through much of the x-axis.

Also of note is the similarity of the curves of the three larger projects: Net, IO, and Compression. Each of these files yielded a similar internal ratio of major changes, as reflected in the graphs of their relative complexity scores.

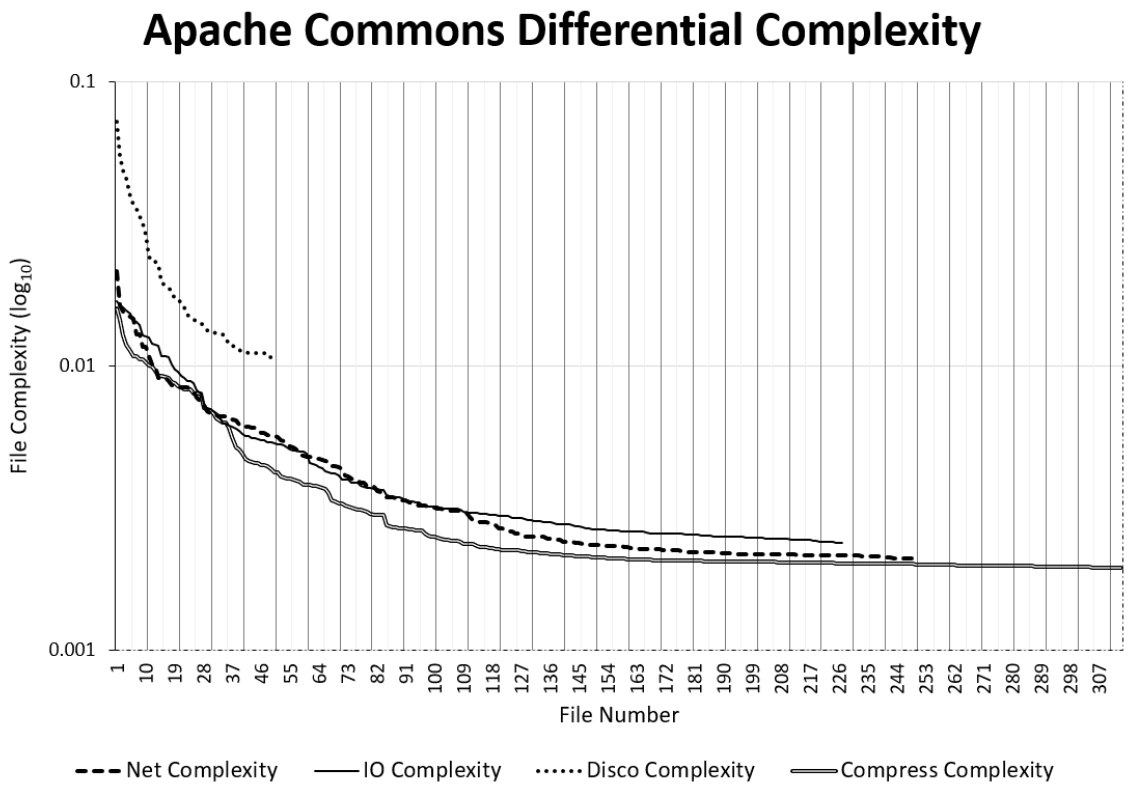

Figure 4-2: Differential complexity scores for selected Apache Commons projects.

From the results—and despite not being able to compare the per-file calculated differential complexity results directly—it is possible to interpret similarities by examining the relationships between the results of each individual project. An initial relationship is the differential complexity ratio of a project, a relationship between the most complexly modified file and the least complexly modified file in each project. Table 4-18 explores the differential complexity ratio for each project. While there is some variation, the differential complexity ratios of each project are in agreement with range and order of magnitude.

Table 4-19: Differential Complexity Ratio

Project	Greatest differential complexity value	Least differential complexity value	Differential complexity ratio
Discovery (P_1)	*0.072622375	0.010720913	6.78
Net (P_2)	*0.021580391	0.002060035	10.75
IO (P_3)	*0.016903606	0.002379585	7.35
Compression (P_4)	*0.01598046	0.0019411362	8.37

Note. * Statistical upper outlier.

The complexly modified files noted in tables 4-14, 4-15, 4-16, and 4-17 are, for the most part, automated unit test cases. For example, in Table 4-16, org.apache.commons.io.FileUtilsTestCase is the most complexly modified file in the Apache Commons IO project. In terms of complexity, the test case file is followed by org.apache.commons.io.FileUtils, which is the file used for testing in an automated test framework. The test case file and the file it tests both reflect complex modifications. Often, the results of unit test frameworks are used to validate appropriate performance of a system or subsystem. Upon obtaining these results, an engineering manager or test manager should consider manually confirming the test case performs satisfactorily because test cases are rarely subjected to testing with their own test cases.

The remainder of this section addresses answers to the individual research questions.

RQ1: What software complexity measurement techniques are best suited to measure the software source code changes performed between subsequent software revisions?

The choice of differential complexity measurement technique for software code depends on several factors, including programming languages used and quality attributes. The aspects of the technologies at play need to be considered. This praxis reduced the scope of this problem space by limiting the analysis to code written in the Java programming language because of the quality and availability of tooling. An analysis of

the differential complexity values and graphical plots reveals the interplay of quality attributes selected for representing the object-oriented Java programming language. Non-object-oriented programmatic languages that favor different quality attributes might require other metrics. Experts must take into account both design and structural simplicity of the language and software projects. They must ask and answer the question, "To what extent does the programmatic language dictate the structure of the software design?"

In determining the differential complexity measurement technique, this praxis extended Saaty's AHP process as implemented in Suri and Singhani's (2015) peer-reviewed work. This praxis offered an established framework for the complexity measurement of Java-based projects using a well-defined set of quality attributes and associated metrics (see Table 3-1). Regardless of the attributes selected, the AHP processes and the processes documented in this praxis can be augmented or extended to cover many other quality attributes, provided they conform to several factors:

- Relevance of the measurement metric used. The metric should be meaningful to the quality attribute it represents.
- Consistency of the complexity metric. Software products with equal values of the complexity metric will exhibit a similar level of software quality.
- Measurement processes automation. The process of measuring should be both repeatable and capable of being automated.

The final concern in developing a differential complexity measuring technique is identifying the differences in complexity between software revisions. This praxis adapted the concepts from Canfora et al.'s (2014) data extraction framework to perform at a more

granular level. The data extraction framework developed as part of this praxis can be used to execute the metrics of a set of quality attributes across the Java files in subsequent revisions of a software baseline. This technique was able to calculate the differences in complexity between the files in the specified software revisions regardless of complexity differences between beginning and ending revisions. This technique has an advantage over static complexity measurement techniques with respect to determining the complexity of changes between subsequent software revisions.

RQ2: How can an accurate measure of the complexity of the software source code changes performed between subsequent software revisions be developed?

This praxis demonstrated a novel technique that combines aspects from several peer-reviewed sources to enable the measurement of differences in complexity between software revisions. This objective was accomplished by providing a framework for the measurement of multiple quality attributes that collectively represent the complexity of software source code changes made between two subsequent revisions of a Java application.

The technique defined in this praxis allows for valuations of one software quality factor relative to another to be adopted or determined specific to the organization. If a software organization makes extensive use of commercial frameworks that deemphasize measurements of internal cohesion, they might wish to shift their emphasis to measurements of encapsulation using the NOM metric or algorithmic complexity using the WMC metric. This praxis technique is easily adaptable to the development of quality attribute weighting factors specific to the needs of individual organizations. Incorporating the AHP process into software testing allows for such alterations. Changes to the quality

attribute weight factors might change the aggregate complexity calculations that combine the multiple quality factors to produce a single differential complexity value across two software baseline revisions.

The data extraction framework described in this praxis is foundational for accurately measuring the complexity of software source differences between subsequent software revisions. Identifying differences between software revisions expands the scope of traditional software complexity measurement. Files are added, moved, or removed as software evolves. Substantial efforts in software development yield simpler systems (Canfora et al., 2014). Static measurement using traditional metrics before and after re-factoring only identifies that the resulting system is simpler and disregards the effort required to make the system simpler. The data extraction framework described in this praxis accounts for these cases and conditions.

RQ3: How can software changes with a large magnitude of measured complexity be identified in a software baseline?

In large, unfamiliar, or quickly changing code bases, the manual process of identifying source code segments with excessive complexity is difficult and time-consuming. The techniques proposed in this praxis can be automated into existing development processes. This praxis combines several techniques, each successful in its own right, into a novel system that measures the complexity of changes made to software source code between versions. This system combines a long-standing multiple-criteria decision analysis process, the AHP, with a set of quality attributes and accompanying metrics appropriate for the Java programming language. By incorporating the output of a data extraction framework designed to measure the differences between two software

revisions, this praxis revealed a process for identifying the portions of a software baseline experiencing the largest magnitude of complexity. Organizations that choose to use a different valuation than the peer-reviewed quality attributes used this praxis will find the processes described herein are easily adaptable.

The outputs of the technique described in this praxis can be introduced to a manual test process or integrated into an automated process. The approach described in this praxis can be used as an initial technique to focus potentially limited test or analysis resources on software source code areas that have undergone complex change. Finally, these complexity results can also be tracked over time as indicators of the progress and direction of the complexity of the source code.

4.5 Threats to Validity

This section outlines potential threats to the overall validity and generalizability of the results of this praxis. Common threats to the validity of the conclusions of this praxis are categorized as threats to the internal validity, external validity, and conclusion. Threats to the internal validity of the findings include factors that can influence the results of the study, but are ignored or cannot be controlled. Assumptions made during incorporation of the AHP models, including the software quality attribute weight factors, have an impact on the validity of the conclusions stemming from implementations of this model. In addition, this praxis evaluated several moderately sized software code bases. The practicality and extent of measurement, as well as the analysis of changes in the source code, depends on the efficiency of class sampling and code inspection techniques. The nth root Eigenvector estimation technique, while effective, requires an increasing level of decimal precision as the number of evaluated alternatives increases.

Several external validity threats reduce the potential for generalization of the results of this praxis. Given the conclusions of this empirical evaluation were derived purely from the analysis of four open-source Java programs, it cannot be assumed these results can be generalized to other programs created in a different programming language or later advances within the Java programming language. For example, Java's extension to support lambda expressions is not supported by the current set of metrics, and the recent movement towards dependency injection frameworks such as J2EE or Spring might have an impact on metrics intended for traditional Java object-oriented design.

Threats to the validity of the conclusions relate to the statistical relevance of correlations that were evaluated in this praxis. These threats cover the correlation of data and outcome that can prevent the reproducibility of the results and conclusions of this praxis. These threats also include concerns of the relationship between underlying theories of complexity measurement and execution of the empirical analysis. Examples include the potential misinterpretation or flawed implementation of AHP processes or validation calculations for the values of λ_{max}, CI, and CR, and could lead to inaccurately weighted values.

The primary challenge to the validity of conclusions of this praxis is the set of projects used for evaluation. These projects consisted exclusively of successfully developed and implemented open-source Java programs from the Apache Commons project. These are community-driven projects with many knowledgeable and self-policing contributors. Flawed or inelegant software implementations are quickly identified and removed. This behavior sets the Apache Commons projects on a pedestal as the ultimate model of a Java software implementation. Results of this praxis are

applicable and relevant to at least the selected set of subprojects, which offered a range of attributes including size, complexity, the period between any two distinct releases, and maturity. Because of the high-quality nature of the Apache Commons projects, the author is confident that the conclusions drawn from these results apply to all other similarly sized and structured Java programs. However, there can be no guarantees made that the model put forth in this praxis will be as effective for other programming languages, or even all Java programmatic techniques. Recent Java language additions or new concepts in application frameworks or programmatic techniques may use different stylistic properties from an idealistic Java application for which this praxis was evaluated.

Chapter 5 − Conclusion

Developing and maintaining software requires multiple small but rapid development iterations of source code modification (Tahvili et al., 2015). These source code modifications introduce defects. To mitigate the introduction of defects, software is frequently tested to ensure introduced changes do not regress functionality. Regression tests are commonly used in software development processes as tools for ensuring the quality of software products. As software baselines and corresponding sets of regression tests increase in size, resource constraints can prevent the execution of a full set of regression tests after every incremental set of source code modifications. In these cases, prioritized regression test cases or automated unit tests are used to improve the efficiency of the testing process.

The technique described in this praxis involved combining an AHP model with a data extraction framework that measures complexity between successive software code changes. This process provides engineering managers and their test teams the ability to identify where complex changes occur in a software baseline between revisions. These revisions might span a single day in the case of a nightly build, multiple week sprints, or even major release cycles spanning half a year or more. Frequent testing is costly and can be labor-intensive. The process described in this praxis can be automated or performed manually, as resources permit or needs demand, to provide engineering managers with readily available data for use in prioritizing their teams' work.

Additionally, the technique described in this praxis identifies and provides an accurate measure for the complex modifications required to simplify a software baseline. This technique identified the FTPClient file from the Apache Commons Net project as

having been complexly changed. The results in Table 4-1 reveal that while the CBO metric decreased in the subsequent revision, the data extraction framework accounted for the change in the complexity between the revisions. This change in complexity would not be considered by existing metric measurement techniques.

The technique described in this praxis was applied to the four Apache Commons projects: Discovery, Net, IO, and Compression. The application provided insights to an engineering manager or test lead. The developers of the Discovery, IO, and Compression project all identified system unit test classes as being within the top six complexly modified files in each project. The Net project developers identified a system unit test class as the 12th most complexly modified file. It is typical for unit tests to be developed by the same teams in the same conditions as the software the tests are intended to evaluate. The unit tests are no less subject to software defects.

As resource constraints force smaller and shorter regression tests, many test managers are augmenting the test process with automated unit tests. The identification of complexly modified unit tests might spur test manager to verify that the automated test is functioning properly. Informed by a list of the most complexly modified files for a given release, engineering managers or test leads can focus resources on verification and validation activities, or direct a reduction in complexity, of the most complexly modified files in a software baseline.

These observations compel the use of MCDM techniques as decision support tools rather than a means of developing the final answer. The ideal solution to MCDM problems might not exist or be realistic. The conclusions presented here should be treated as indications of the best solution to the problem of measuring the complexity of software

codes and the effect of changes in the software baseline on the complexity of the final version. While the quest for finding an optimal and universally acceptable MCDM technique might never end (Triantaphyllou & Mann, 1995), scholarly effort in this domain of decision making remains important and valuable in many research, engineering, and scientific applications.

Chapter 6 – Future Research

The technique described in this praxis builds upon an extensive body of existing knowledge, but offers multiple opportunities for future research. These opportunities are identified below and expanded upon in the following sections:

- Improving the implementation described in this praxis
- Extension of the techniques described in this praxis to other programming languages or styles
- A comparative analysis of this praxis technique with alternate techniques
- Expansion of the concept of differential complexity

Because this praxis focused solely on identifying differential complexity between revisions of software developed in Java programming languages, there is a need for future research to identify the applicability of the technique to other programming languages. Although there have been efforts to characterize the quality attributes and corresponding metrics for many programming languages, no other studies were found in the literature that incorporated the quality attributes and corresponding metrics for non-Java programming languages into an AHP methodology focusing on differential complexity. Researchers in this area might wish to focus on the best set of quality attributes and corresponding metrics, as well as the recommended Eigenvector weighting values needed to expand the technique described in this praxis to other programming languages. This praxis used open-sourced Java projects of a traditional structure, which might not be representative of commercial Java development. Extending the technique described in this praxis to support new Java language constructs such as lambda

expressions or advances in Java enterprise dependency injection frameworks is another potential area for future research.

Improvements to the techniques described in this praxis are encouraged. Although the AHP methodology in this praxis was fit for purpose and supported by previous research in a related testability area (Suri & Singhani, 2015), there are many other multiple-criteria decision analysis methodologies available. Scholars have suggested complementing the weaknesses and limitations of the AHP model to maintain the acceptable values of CR while reducing ambiguities in the pair-wise evaluation procedure of the AHP model. While studies have been conducted on the complexities associated with the hierarchy structure and rank reversal issues (Y.-M. Wang & Elhag, 2006), few have pertained to identifying issues with comparing the AHP hierarchies, and more research is needed to address this problem. An additional issue with hierarchy structure is that the CR values vary depending on the grouping of the attributes or elements, and the values of each sub-attribute are modified based on the values of their upper attributes. Complementary models or processes developed from further research should focus on addressing this problem.

The application of improved, derived, or alternates of the AHP to techniques employed in this praxis is an additional area for future research. Additional opportunities are available for improvements to the data extraction framework. These opportunities might focus on expanding this framework beyond the currently supported set of metrics, as well as developing improvements to its complexity difference calculation algorithms.

The opportunity exists for future research in comparative analysis of the technique described in this praxis and alternate techniques of test case selection. There is an active

community of researchers building a body of knowledge of test case selection techniques. The logical extension of the technique described in this praxis is to perform an automated test case selection combined with a comparative analysis with other test case selection techniques.

Finally, complexity measurement during software production remains a difficult task. Challenges occur throughout the software development lifecycle that include the design of a measurement procedure to exploit the results that inform decision making. While these difficulties have been subject to research over time, researchers have focused on objectivity, the correctness of approach, and limits constraining the measurement procedures (McCabe, 1976; Abran, Lopez, & Habra, 2004). The literature documents difficulties related to the process of software measurement, but the exploitation and application of the results of a differential complexity measurement process have yet to be addressed. Furthermore, while the field of software complexity is rich, there is comparatively little information reported on differential complexity. As such, expansion of the concept of differential complexity throughout the evolution of a software baseline warrants additional research.

From the initial analysis of the graph in Figure 4-1, there were similarities in the curves of the three larger projects—Net, IO, and Compression. There are additional similarities in the differential complexity ratios for each project, as presented in Table 4-18. These similarities might be indicators of acceptable limits to the complexity changes between software revisions. This possibility warrants future research into the topic of differential complexity to further illuminate how complexity evolves within a software baseline.

References

Abran, A., Lopez, M., & Habra, N. (2004). An analysis of the McCabe cyclomatic complexity number. *Proceedings of the 14th International Workshop on Software Measurement (IWSM) IWSM/Metrikon,* 391–405. Berlin, Germany.

Albrecht, A. J. (1979, October). Measuring application development productivity. *Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium,* 83–92. Monterey, CA: IBM.

Albrecht, A. J., & Gaffney, J. E. (1983). Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering, SE-9,* 639–648. https://doi.org/10.1109/tse.1983.235271

Al-Hajjaji, M., Alsmadi, I., & Samarah, S. (2012). Evaluating software complexity based on decision coverage. *Informatica Economica, 16*(1), 5–13. Retrieved from http://revistaie.ase.ro/

Babu, P. C., Prasad, A. N., & Sudhakar, D. (2013, August). Software complexity metrics: A survey. *International Journal of Advanced Research in Computer Science and Software Engineering, 3,* 1359–1361. Retrieved from http://www.ijarcsse.com

Babu, P. C., Ramakrishna, S., & Bharathi, K. (2013). Assessment of maintainability factor. *International Journal of Computer Science Engineering and Information Technology Research, 3*(3), 29–42. Retrieved from http://www.tjprc.org/journals.php?jtype=2&id=14

Badri, M., & Toure, F. (2012). Empirical analysis of object-oriented design metrics for predicting unit testing effort for classes. *Journal of Software Engineering and Applications, 5,* 513–526. https://doi.org/10.4236/jsea.2012.57060

Banker, R. D., Datar, S. M., Kemerer, C. F., & Zweig, D. (1993). Software complexity and maintenance costs. *Communications of the ACM, 36*(11), 81–94. https://doi.org/10.1145/163359.163375

Banker, R. D., & Slaughter, S. A. (2000). The moderating effects of structure on volatility and complexity in software enhancement. *Information Systems Research, 11,* 219–240. https://doi.org/10.1287/isre.11.3.219.12209

Bannerman, S., & Martin, A. (2011). A multiple comparative study of test-with development product changes and their effects on team speed and product quality. *Journal of Empirical Software Engineering, 16,* 177–210. https://doi.org/10.1007/s10664-010-9137-5

Basili, V. R. (1980). *Qualitatives software complexity models: A summary in tutorial on models and methods for software management and engineering.* Los Alamitos, CA: IEEE Computer Society Press.

Basili, V. R., Briand, L. C., & Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering, 22,* 751–761. https://doi.org/10.1109/32.544352

Baudry, B., Le Traon, Y., Sunyé, G., & Jézéquel. J.-M. (2003). Measuring and improving design patterns testability. *Proceedings of the Ninth International Software Metrics Symposium,* 50–59. https://doi.org/10.1109/metric.2003.1232455

Baumeister, H., Marchesi, M., & Holcombe, M. (Eds.). (2005). *Extreme programming and agile processes in software engineering: 6th International Conference, XP 2005.* Sheffield, England: Springer. https://doi.org/10.1007/b137278

Belton, V., & Gear, T. (1983). On a shortcoming of Saaty's method of analytic hierarchies. *International Journal of Management Science, 11,* 228– 230. https://doi.org/10.1016/0305-0483(83)90047-6

Bilal, H., & Black, S. E. (2005). Using the ripple effect to measure software quality. *13th Software Quality Management Conference, 13,* 183–198. Retrieved from https://www.semanticscholar.org/paper/Using-the-Ripple-Effect-to-Measure-Software-Qualit-Bilal/a36f07dd1156314ecf300865ac2d63dfb1165955

Binder, R. V. (1994). Design for testability in object-oriented systems. *Communications of the ACM, 37*(9), 87–100. https://doi.org/10.1145/182987.184077

Briand, L. C., Daly, J. W., & Wust, J. (1999). A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering, 25,* 91–121. https://doi.org/10.1109/32.748920

Briand, L. C., Morasca, S., & Basili, V. R. (1996). Property-based software engineering measurement. *IEEE Transactions on Software Engineering 1996, 22,* 68–85. https://doi.org/10.1109/32.481535

Bruntink, M., & van Deursen, A. (2006). An empirical study into class testability. *Journal of Systems and Software, 79,* 1219–1232. https://doi.org/10.1016/j.jss.2006.02.036

Buckley, J., Mens, T., Zenger, M., Rashid, A., & Kniesel, G. (2005). Towards a taxonomy of software change. *Journal of Software: Evolution and Process, 17,* 309–332. https://doi.org/10.1002/smr.319

Buse, R. P., & Weimer, W. R. (2010). Learning a metric for code reliability. *IEEE Transactions on Software Engineering 2010, 36,* 546–558. https://doi.org/10.1109/tse.2009.70

Butler, B. R., Blair, P. K., Fox, A. J., Hall, I. A., Henry, K. N., McDermid, J. A., . . . Rodd, M. (2004). The challenges of complex IT projects. *Relatório técnico.* London, England: Royal Academy of Engineering and British Computer Society.

Canfora, G., Cerulo, L., Cimitile, M., & Di Penta, M. (2014). How changes affect software entropy: An empirical study. *Journal of Empirical Software Engineering, 19,* 1–38. https://doi.org/10.1007/s10664-012-9214-z

Chhillar, U., & Bhasin, S. (2011). Establishing relationship between complexity and faults for object-oriented software systems. *International Journal of Computer Science Issues, 8,* 437–442. Retrieved from http://www.ijcsi.org/

Chidamber, S. R., & Kemerer, C. F. (1991). Towards a metrics suite for object oriented design. *ACM SigPlan Notices, 26*(11), 197–211. https://doi.org/10.1145/118014.117970

Chidamber, S. R., & Kemerer, C. F. (1992, December). *A metrics suite for object oriented design* (Sloan Working Paper No. 3524-93). Retrieved from https://dspace.mit.edu/bitstream/handle/1721.1/48424/metricssuiteforo00chid.pdf

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering, 20,* 476–493. https://doi.org/10.1109/32.295895

Coad, P., & Yourdon, E. (1991). *Object-oriented design.* Englewood Cliffs, NJ: Prentice Hall.

Conte, S. D., Dunsmore, H. E., & Shen, V. Y. (1986). *Software engineering metrics and models.* Redwood City, CA: Benjamin/Cummings.

Debbarma, M. K., Debbarma, S., Debbarma, N., Chakma, K., & Jamatia, A. (2013). A review and analysis of software complexity metrics in structural testing. *International Journal of Computer and Communication Engineering, 2,* 129–133. https://doi.org/10.7763/ijcce.2013.v2.154

Delamaro, E. (2014). Test case selection : A systematic literature review. *International Journal of Software Engineering and Knowledge Engineering, 24,* 653–676. https://doi.org/10.1142/S0218194014500259

Delange, J. (2014, December 15). Managing model complexity [Blog post]. Retrieved from https://insights.sei.cmu.edu/sei_blog/2014/12/managing-model-complexity.html

Delange, J. (2015, September 14). Managing software complexity in models [Blog post]. Retrieved from https://insights.sei.cmu.edu/sei_blog/2015/09/managing-software-complexity-in-models.html

Delange, J., Hudak, J., Nichols, W., McHale, J., & Nam, M. Y. (2015). *Evaluating and mitigating the impact of complexity in software models* (Technical Report No. CMU/SEI-2015-TR-013). Retrieved from https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=448083

Desai, C. G. (2014). Object oriented design metrics, frameworks and quality models. *Information Science and Technology, 3*(2), 66–70. Retrieved from https://www.bioinfopublication.org/journal.php?jouid=BIJ0000002

Dhillon, A., & Singh, A. (2012). Analysis of software metrics for bubble sort and selection sort. *International Journal of Computer Applications & Information Technology, 1*(1), 38–40. Retrieved from www.ijcait.com

Do, H., Mirrarab, S., Tahvildari, L., & Rothermel, G. (2010). The effects of time constraints on test case prioritizaton: A series of controlled experiments. *IEEE Transactions on Software Engineering, 36,* 593–617. https://doi.org/10.1109/tse.2010.58

Donegan, H. A., & Dodd, F. J. (1991). A note on Saaty's random indexes. *Mathematical and Computer Modelling, 15*(10), 135–137. https://doi.org/10.1016/0895-7177(91)90098-R

Dromey, R. G. (1995). A model for software product quality. *IEEE Transactions on Software Engineering, 21,* 146–162. https://doi.org/10.1109/32.345830

Du, Q., & Wang, F. (2010, December). *Software power: A new approach to software complexity metrics.* Paper presented at the Second World Congress on Software Engineering, Hubei, Wuhan, China. https://doi.org/10.1109/wcse.2010.61

Durisic, D., Nilsson, M., Staron, M., & Hansson, J. (2013). Measuring the impact of changes to the complexity and coupling properties of automotive software systems. *Journal of Systems and Software, 86,* 1275–1293. https://doi.org/10.1016/j.jss.2012.12.021

Elbaum, S., Malishevsku, A. G., & Rothermel, G. (2002). Test case prioritization: A family of emprical studies. *IEEE Transactions on Software Engineering, 28,* 159–182. https://doi.org/10.1109/32.988497

Emam, K. E., Melo, W., & Javam, C. M. (2001). The prediction of faulty classes using object-oriented design metrics. *The Journal of Systems and Software, 56,* 63–75. https://doi.org/10.1016/s0164-1212(00)00086-8

Fenton, N. E. (1991). *Software metrics: A rigorous approach.* London, England: Chapman and Hall.

Fenton, N. E. (1994). Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering, 20,* 199–206. https://doi.org/10.1109/32.268921

Fenton, N. E., & Kitchenham, B. (1991). Validating software measures. *Software Testing, Verification and Reliability, 1*(2), 27–42. https://doi.org/10.1002/stvr.4370010204

Fenton, N. E., & Pfleeger, S. L. (1998). *Software metrics: A rigorous & practical approach* (2nd ed.). Boston, MA: PWS.

Fernando, W. K. S. D., Wijayarathne, D. G. S. M., Fernando, J. S. D., Mendis, M. P. L., & Guruge, I. (2014). The importance of software metrics: Perspective of a software development project in Sri Lanka. *4th SAITM Research Symposium on Engineering Advancements - 2014, 4,* 91–95. Retrieved from http://saitm.edu.lk/fac_of_eng/RSEA/SAITM_RSEA_2014/index.htm

Flanagan, M. T. (2015). *Michael Thomas Flanagan's Java scientific library.* Retrieved October 30, 2017, from https://www.ee.ucl.ac.uk/~mflanaga/java/

Fóthi, Á., Nyéky-Gaizler, J., & Porkoláb, Z. (2003). The structured complexity of object-oriented programs. *Mathematical and Computer Modeling, 38,* 815–827. https://doi.org/10.1016/s0895-7177(03)90066-5

Franek, J., & Kresta, A. (2014). Judgment scales and consistency measure in AHP. *Procedia Economics and Finance, 12,* 164–173. https://doi.org/10.1016/s2212-5671(14)00332-3

Gao, J. Z., Tsao, H.-S. J.,·& Wu, Y. (2003). *Testing and quality assurance for component-based software.* Boston, MA: Artech.

Garmabaki, A. H. S., Ahmadi, A., & Ahmadi, M. (2015). Maintenance optimization using multi-attribute utility theory. In U. Kumar, A. Ahmadi, A. K. Verma, & P. Varde (Eds.), *Current trends in reliability, availability, maintainability and safety* (pp. 13–25). New York, NY: Springer.

Ghosh, S. M., Sharma, H. R., & Mohabay, V. (2011). Software change management—Technological dimension. *International Journal of Smart Home, 5*(2), 29–36. Retrieved from http://www.sersc.org/journals/IJSH/

Gill, G. K., & Kemerer, C. F. (1991). Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering, 17,* 1284–1288. https://doi.org/10.1109/32.106988

Gonzalez, R. R. (1995). A unified metric of software complexity: Measuring productivity, quality, and value. *Journal of Systems and Software, 29,* 17–37. https://doi.org/10.1016/0164-1212(94)00126-8

Grady, R. B. (1992). *Practical software metrics for project management and process improvement.* Englewood Cliffs, NJ: Prentice Hall.

Gupta, C., Singh, Y., & Chauhan, D. S. (2010). A dynamic approach to estimate change impact using type of change propagation. *Journal of Information Processing Systems, 6,* 597–608. https://doi.org/10.3745/jips.2010.6.4.597

Gustafson, D. A., Tan, J. T., & Weaver, P. (1996). Software metric specification. In A. Melton (Ed.), *Software measurement* (p. 179). Toronto, ON, Canada: International Thomson.

Hallem, S., Park, D., & Engler, D. (2004). Uprooting software defects at the source. *Queue, 1*(8), 64–71. https://doi.org/10.1145/966712.966722

Halstead, M. H. (1977). *Elements of software science.* New York, NY: North-Holland.

Harrison, W. (1992). An entropy-based measure of software complexity. *IEEE Transactions on Software Engineering, 18*(11), 1025–1029. https://doi.org/10.1109/32.177371

Hassan, A. E. (2009). Predicting faults using the complexity of code changes. *ICSE '09 Proceedings of the 31st International Conference on Software Engineering, 31,* 78–88. https://doi.org/10.1109/icse.2009.5070510

Henderson-Sellers, B., Constantine, L. L., & Graham, I. M. (1996). Coupling and cohesion: Towards a valid metrics suite for object-oriented analysis and design. *Object Oriented Systems, 3,* 143–158. Retrieved from http://www.ibrarian.net/navon/paper/Coupling_and_cohesion__towards_a_valid_metrics_su.pdf?paperid=1090060

Honglei, T., Wei, S., & Yanan, A. (2009). The research on software metrics and software complexity metrics. *International Forum on Computer Science-Technology and Applications, 9,* 131–136. https://doi.org/10.1109/ifcsta.2009.39

Hsu, C.-J., & Huang, C.-Y. (2014). Optimal Weighted Combinational Models for Software Reliability Estimation and Analysis. *IEEE Transactions on Reliability*, *63*(3), 731–749. Retrieved from http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6786506

IEEE recommended practice for software requirements specifications—IEEE Std. 830-1993. (1994). https://doi.org/10.1109/IEEESTD.1994.121431

Jabbar, A. P., & Sarala, S. (2011). Advanced program complexity metrics and measurement. *International Journal of Computer Applications, 23*(2), 29–33. https://doi.org/10.5120/2860-3679

Jacquet, J.-P., & Abran, A. (1998). Metrics validation proposals: A structured analysis. In R. Dumke & A. Abran (Eds.), *Software measurement: Current trends in Research and Practice* (pp. 43-59). https://doi.org/10.1007/978-3-663-08949-0_3

Jain, L. C., Behera, H. S., Mandal, J. K., & Mohapatra, D. P. (Eds.). (2014). Computational intelligence in data mining—Volume 1: Proceedings of the International Conference on CIDM, 20-21 December 2014. https://doi.org/10.1007/978-81-322-2205-7

Jamali, S. M. (2006). *Object oriented metrics: A survey approach* [Technical report]. Tehran, Iran: Department of Computer Engineering, Sharif University of Technology.

Jayaswal, B. K., Patton, P. C., & Forman, E. H. (2007). *The analytic hierarchy process (AHP) in software development.* Upper Saddle River, NJ: Pearson/Prentice Hall.

Jiang, R. (2015). Research and measurement of software complexity based on Wuli, Shili, Renli (WSR) and information entropy. *Entropy, 17,* 2094–2116. https://doi.org/10.3390/e17042094

Karlsson, J., & Ryan, K. (1997). A cost-value approach for prioritizing requirements. *IEEE Software, 14*(5), 67–74. https://doi.org/10.1109/52.605933

Kassab, M., & Kilicay-Ergin, N. (2015). Applying analytical hierarchy process to system quality requirements prioritization. *Innovations in Systems and Software Engineering, 11,* 303–312. https://doi.org/10.1007/s11334-015-0260-8

Kaur, H., & Verma, G. N. (2016). Software complexity measurement: A critical review. *International Journal of Engineering and Applied Computer Science, 1,* 12–16. https://doi.org/10.24032/ijeacs/0101/03

Kaushik, S., & Tyagi, K. (2016). Critical review on test case generation systems and techniques. *International Journal of Computer Applications, 133*(7), 24–29. https://doi.org/10.5120/ijca2016907916

Keyes, J. (1992). New metrics needed for new generation. *Software Magazine, 12*(6), 42–56. Retrieved from http://www.softwaremag.com/

Khan, R. A., & Mustafa, K. (2009). Metric based testability model for object oriented design (MTMOOD). *ACM SIGSOT Software Engineering Notes, 34*(2), 1. https://doi.org/10.1145/1507195.1507204

Khoshgoftaar, T. M., Allen, E. B., Kalaichelvan, K. S., & Goel, N. (1996). Early quality prediction: A case study in telecommunications. *IEEE Software, 13,* 65–71. https://doi.org/10.1109/52.476287

Khoshgoftaar, T. M., & Munson, J. C. (1990). Predicting software development errors using software complexity metrics. *IEEE Journal on Selected Areas in Communications, 8,* 253–261. https://doi.org/10.1109/49.46879

Khoshgoftaar, T. M., & Munson, J. C. (1992). A measure of software system complexity and its relationship to faults. *Proceedings of the 1992 International Simulation Technology Conference* (pp. 267-272). San Diego, CA: Society for Computer Simulation.

Kitchenham, B., Pfleeger, S. L., & Fenton, N. (1995). Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering, 21,* 929–944. https://doi.org/10.1109/32.489070

Klemola, T., & Rilling, J. (2003). A cognitive complexity metric based on category learning. *Proceedings on the Second IEEE International Conference on Cognitive Informatics, 2,* 106–112. https://doi.org/10.1109/coginf.2003.1225966

Klindee, P., & Prompoon, N. (2015). Test cases prioritization for software regression testing using analytic hierarchy process. *12th International Joint Conference on Computer Science and Software Engineering (JCSSE), 12,* 168–173. https://doi.org/10.1109/jcsse.2015.7219790

Kout, A., Toure, F., & Badri, M. (2011). An empirical analysis of a testability model for object-oriented programs. *ACM SIGSOFT Software Engineering Notes, 36*(4), 1–5. https://doi.org/10.1145/1988997.1989020

Li, H. F., & Cheung, W. K. (1987). An empirical study of software metrics. *IEEE Transactions on Software Engineering, SE-13,* 697–708. https://doi.org/10.1109/tse.1987.233475

Lorenz, M., & Kidd, J. (1994). *Object oriented metrics.* Englewood, NJ: Prentice Hall.

Maadawy, S., & Salah, A. (2012). Measuring change complexity from requirements: A proposed methodology. *International Magazine on Advances in Computer Science and Telecommunications (IMACST), 3*(1), 1–11. Retrieved from https://pdfs.semanticscholar.org/7826/f68a3c8e8af981e427b1ec38b8f9b5b47392.pdf

Madi, A., Zein, O. K., & Kadry, S. (2013). On the improvement of cyclomatic complexity metric. *International Journal of Software Engineering and Its Applications, 7*(2), 67–82. Retrieved from http://www.sersc.org/journals/IJSEIA/

Malishevsky, A. G., Rothermel, G., & Elbaum, S. (2002). Modeling the cost-benefits tradeoffs for regression testing techniques. *Proceedings of the International Conference on Software Maintenance, ISCM, 2002,* 204–213. https://doi.org/10.1109/icsm.2002.1167767

Mas y Parareda, B., & Pizka, M. (2007, December). *Measuring productivity using the infamous lines of code metric.* Poster session presented at APSEC2007, 1st International Workshop on Software Productivity Analysis and Cost Estimation, Nagoya, Japan.

Mateo, J. R. S. C. (2012). Weighted sum method and weighted product method. *Multi criteria analysis in the renewable energy industry* (pp. 19–22). https://doi.org/10.1007/978-1-4471-2346-0_4

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering, SE-2,* 308–320. https://doi.org/10.1109/tse.1976.233837

McCabe, T. J. (1982). Structured testing: A software testing methodology using the cyclomatic complexity metric (NBS Special Publication No. pp. 500–99). https://doi.org/10.6028/nbs.sp.500-99

McCall, J. A., Richards, P. K., & Walters, G. F. (1977). *Factors in software quality: Concept and definitions of software quality* (Technical Report No. RADC-TR-77-369). Retrieved from http://www.dtic.mil/dtic/tr/fulltext/u2/a049014.pdf

Moreau, D. R., & Dominick, W. D. (1989). Object-oriented graphical information systems: Research plan and evaluation metrics. *The Journal of Systems and Software, 10,* 23–38. https://doi.org/10.1016/0164-1212(89)90059-9

Moreau, D. R., & Dominick, W. D. (1990). A programming environment evaluation methodology for object-oriented systems: Part I—The methodology. *Journal of Object-Oriented Programming, 3*(1)*,* 38–52. Retrieved from https://dl.acm.org/citation.cfm?id=J472&CFID=998706242&CFTOKEN=75378168

Morris, P. M., & Desharnais, J. M. (2001). Function point analysis: Validating the results. *Proceedings of the IFPUG Spring Conference,* Atlanta. GA. Retrieved from https://www.totalmetrics.com/function-points-downloads/Function-Points-Validation-Auditing.pdf

Mukker, A. R., Singh, L., & Mishra, A. K. (2014). Systematic review of metrics in software agile projects. *Compusoft, 3,* 533–539. http://dx.doi.org/10.6084/ijact.v3i2.252

Munson, J. C., & Elbaum, S. G. (1998). Code churn: A measure for estimating the impact of code change. *ICSM '98: Proceedings of the International Conference on Software Maintenance,* 24–31. https://doi.org/10.1109/icsm.1998.738486

Munson, J. C., & Khoshgoftaar, T. M. (1991). The use of software metrics in software reliability modeling. *Proceedings of the International Symposium on Software Reliability Engineering, 1991,* 2–11. https://doi.org/10.1109/issre.1991.145346

Narciso, E. N., Delamaro, M. E., & Nunes, F. D. L. D. S. (2014). Test case selection: A systematic literature review. *International Journal of Software Engineering and Knowledge Engineering, 24,* 653–676. https://doi.org/10.1142/s0218194014500259

Nazir, M., Khan, R. A., & Mustafa, K. (2010). Testability estimation framework. *International Journal of Computer Applications, 2*(5), 9–14. https://doi.org/10.5120/668-937

Pan, J. (1999). *Software reliability.* Retrieved from https://users.ece.cmu.edu/~koopman/des_s99/sw_reliability/#reference

Pasupathy, S., & Bhavani, R. (2013). Measuring the quality of software through analytical design by OOAD metrics. *International Journal of Computer Applications, 63*(13), 39–44. https://doi.org/10.5120/10530-5514

Pettichord, B. (2002). Design for testability. *Proceedings of the 20th Annual Pacific Northwest Software Quality Conference*, 1–28. Retrieved from http://www.uploads.pnsqc.org/proceedings/pnsqc2002.pdf

Pham, H. (2007). *System software reliability.* New York, NY: Springer. https://doi.org/10.1007/1-84628-295-0

Prasanth, N. N., & Subramanian, P. R. (2009). Fuzzy based framework for software complexity analysis. *Journal of Software Engineering (3),* 11–16. Retrieved from http://scialert.net/jindex.php?issn=1819-4311

Prather, R. E. (1984). An axiomatic theory of software complexity metrics. *Computer Journal, 27,* 42–45. https://doi.org/10.1093/comjnl/27.4.340

Pravin, A., & Srinivasan, S. (2013). Effective test case selection and prioritization in regression testing. *Journal of Computer Science, 9,* 654–659. https://doi.org/10.3844/jcssp.2013.654.659

Riguzzi, F. (1996). *A survey of software metrics* (Technical Report No. DEIS-LIA-96-010). Bologna, Italy: Università degli Studi di Bologna.

Saaty, T. L. (1972). An eigenvalue allocation model for prioritization and planning (Working paper). Philadelphia: University of Pennsylvania Energy Management and Policy Center.

Saaty, T. L. (1977). A scaling method for priorities in hierarchical structures. *Journal of Mathematical Psychology, 15,* 234–281. https://doi.org/10.1016/0022-2496(77)90033-5

Saaty, T. L. (1980). *The analytic hierarchy process.* New York, NY: McGraw-Hill.

Saaty, T. L. (2008). Decision making with the analytic hierarchy process. *International Journal of Services Sciences, 1,* 83–98. https://doi.org/10.1504/ijssci.2008.017590

Sagarna, R., & Lozano, J. A. (2008). Software metrics mining to predict the performance of estimation of distribution algorithms in test data generation. In C. Cotta, S. Reich, R. Schaefer, & A. Ligęza (Eds.), *Studies in computational intelligence* (pp. 235–254). https://doi.org/10.1007/978-3-540-77475-4_15

Sahraoui, H. A., Godin, R., & Miceli, T. (2000). Can metrics help to bridge the gap between the improvement of OO design quality and its automation? *Proceedings of the International Conference on Software Maintenance, 2000,* 154–162. https://doi.org/10.1109/icsm.2000.883034

Satapathy, S. C., Avadhani, P. S., Udgata, S. K., & Lakshminarayana, S. (Eds.). (2013). ICT and critical infrastructure. Proceedings of the 48th Annual Convention of Computer Society of India- Vol II. https://doi.org/10.1007/978-3-319-03095-1

Schneidewind, N. F. (1992). Methodology for validating software metrics. *IEEE Transactions on Software Engineering, 18,* 410–442. https://doi.org/10.1109/32.135774

Shah, S. M. A., & Morisio, M. (2012). Complexity metrics significance for defects: An empirical view. *Proceedings of the 2012 International Conference on Information Technology and Software Engineering, 2012,* 29–37. https://doi.org/10.1007/978-3-642-34531-9_4

Shannon, C. E. (2001). A mathematical theory of communication. *ACM Sigmobile Mobile Communication Review, 5*(1), 3–55. https://doi.org/10.1145/584091.584093

Sharma, A., & Kushwaha, D. S. (2010). A complexity measure based on requirement engineering document. *International Conference on Computer and Communication Technology (ICCCT), 2010,* 608–615. https://doi.org/10.1109/iccct.2010.5640472

Shi, Y., & Xu, S. (2007). A new method for measurement and reduction of software complexity. *Tsinghua Science and Technology, 12*(S1), 212–216. https://doi.org/10.1016/S1007-0214(07)70112-8

Shooman, M. L. (1983). *Software engineering: Design, reliability and management* (2nd ed.). New York, NY: McGraw-Hill Ryerson.

Singh, Y., Sabharwal, S., & Sood, M. (2004). A systematic approach to measure the problem complexity of software requirement specifications of an information system. *Information and Management Sciences, 15*(1), 69–90. Retrieved from http://ijims.ms.tku.edu.tw/IJIMS/main.php

Singh, Y., & Saha, A. (2010). Predicting testability of eclipse: Case study. *Journal of Software Engineering, 4,* 122–136. https://doi.org/10.3923/jse.2010.122.136

Sohaney, P. (2015, January 26). *Importance of software quality metrics*. Retrieved from https://www.linkedin.com/pulse/importance-software-quality-metrics-piyush-sohaney

Stathis, J., & Jeffrey, D. R. (1993). An empirical study of Albrecht's function points, in measurement for improved IT management. *Proceedings of 1st ASMA Australian Conference on Software Metrics, 1993,* 96–117.

Suri, P. R., & Singhani, H. (2015). Testability assessment of object oriented software using static metric model and analytic hierarchy process. *International Journal of Computer Science Issues, 12*(3), 76–83. Retrieved from http://www.ijcsi.org/

Tahvili, S., Saadatmand, M., & Bohlin, M. (2015). Multi-criteria test case prioritization using fuzzy analytic hierarchy process. *The Tenth International Conference on*

Software Engineering Advances, 10, 290–296. Retrieved from https://www.iaria.org/conferences2015/ICSEA15.html

Tait, P., & Vessey, I. (1988). The effect of user involvement on system success: A contingency approach. *MIS Quarterly, 12,* 91–108. https://doi.org/10.2307/248809

Tashtoush, Y., Al-Maolegi, M., & Arkok, B. (2014). The correlation among software complexity metrics with case study. *International Journal of Advanced Computer Research, 4,* 414–419. Retrieved from http://www.ijacr.com/

Tegarden, D. P., Sheetz, S. D., & Monarchi, D. E. (1992a). A software complexity model of object-oriented systems. *Decision Support Systems, The International Journal, 13,* 241–262. https://doi.org/10.1016/0167-9236(93)e0045-f

Tegarden, D. P., Sheetz, S. D., & Monarchi, D. E. (1992b). The effectiveness of traditional metrics for object-oriented systems. *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, 4,* 359–368. https://doi.org/10.1109/hicss.1992.183365

Tran-Cao, D., Lévesque, G., & Meunier, J.-G. (2004, January). Software functional complexity measurement with the task complexity approach. *Actes de la Deuxieme Conference Internationale Associant Chercheurs Vietnamiens et Francophones en Informatique,* 77–86. Hanoi, Vietnam. Retrieved from https://www.researchgate.net/publication/220704489_Software_Functional_Complexity_Measurement_with_the_Task_Complexity_Approach

Triantaphyllou, E., & Mann, S. H. (1989). An examination of the effectiveness of multi-dimensional decision-making methods: A decision-making paradox. *Decision Support Systems, 5,* 303–312. https://doi.org/10.1016/0167-9236(89)90037-7

Triantaphyllou, E., & Mann, S. H. (1995). Using the analytic hierarchy process for decision making in engineering applications: Some challenges. *International Journal of Industrial Engineering: Applications and Practice, 2*(1), 35–44. Retrieved from http://journals.sfu.ca/ijietap/index.php/ijie

Van der Berg, K. G., & van der Broek, P. M. (1996). Axiomatic validation in the software metric development process. In A. Melton (Ed.), *Software measurement* (pp. 157–177). Toronto, ON, Canada: International Thomson.

Vinothkumar, N., & Galeebathullah, B. (2015). Prioritizing test cases for regression testing: A model based approach. *International Journal for Trends in Engineering & Technology, 4*(1), 1–4. Retrieved from http://ijtet.com/

Wang, Y., King, G., Court, I., Ross, M., & Staples, G. (1997). On testable object-oriented programmaing. *ACM SIGSOFT Software Engineering Notes, 22*(4), 84–90. https://doi.org/10.1145/263244.263270

Wang, Y., & Shao, J. (2003, August). Measurement of the cognitive functional complexity of software. *Proceedings of the Second IEEE International Conference on Cognitive Informatics, 2,* 67–74. https://doi.org/10.1109/coginf.2003.1225955

Wang, Y.-M., & Elhag, T. M. S. (2006). An approach to avoiding rank reversal in AHP. *Decision Support Systems, 42,* 1474–1480. https://doi.org/10.1016/j.dss.2005.12.002

Weyuker, E. J. (1988). Evaluating software complexity measures. *IEEE Transactions on Engineering Management, 14,* 1357–1365. https://doi.org/10.1109/32.6178

Whittle, J., Clark, T., & Kühne, T. (Eds.). (2011, October). Model driven engineering languages and systems. *Proceedings of the 14th International Conference, MODELS 2011, 14,* 1–730. https://doi.org/10.1007/978-3-642-24485-8

Zelkowitz, M. V. (Ed.). (2003). *Highly dependable software: Advances in computers.* Houston, TX: Gulf Professional.

Zuse, H. (1991). *Software complexity: Measures and methods.* New York, NY: Dde Gruyter.

Zuse, H. (1994). Foundations of the validation of object-priented software measures. In R. Dumke & H. Zuse (Eds.), *Theorie und Praxis der Softwaremessung* (pp. 136–213). https://doi.org/10.1007/978-3-663-14540-0_9

Appendix – Complete Results

Table A-1: Apache Commons Discovery Complete Results

Class	Complexity
org.apache.commons.discovery.log.SimpleLog	0.072622375
org.apache.commons.discovery.resource.names.DiscoverNamesInFile	0.054651828
org.apache.commons.discovery.tools.DiscoverClass	0.048795669
org.apache.commons.discovery.tools.ManagedProperties	0.045799577
org.apache.commons.discovery.resource.classes.ResourceClassDiscoverImpl	0.039344126
org.apache.commons.discovery.test.TestAll	0.036988141
org.apache.commons.discovery.Resource	0.035631755
org.apache.commons.discovery.resource.names.ResourceNameDiscoverImpl	0.033802666
org.apache.commons.discovery.jdk.JDK12Hooks	0.031745639
org.apache.commons.discovery.DiscoveryException	0.028886386
org.apache.commons.discovery.resource.ResourceDiscoverImpl	0.024293324
org.apache.commons.discovery.jdk.JDK11Hooks	0.023330746
org.apache.commons.discovery.tools.DiscoverSingleton	0.023231782
org.apache.commons.discovery.tools.SPInterface	0.022902818
org.apache.commons.discovery.resource.ClassLoaders	0.019594015
org.apache.commons.discovery.resource.names.DiscoverNamesInDictionary	0.019142532
org.apache.commons.discovery.resource.classes.DiscoverClasses	0.018857812
org.apache.commons.discovery.jdk.PsuedoSystemClassLoader	0.01824453
org.apache.commons.discovery.tools.ClassUtils	0.017202504
org.apache.commons.discovery.resource.DiscoverResources	0.016993166
org.apache.commons.discovery.resource.names.NameDiscoverers	0.016697332
org.apache.commons.discovery.resource.names.DiscoverMappedNames	0.016054395
org.apache.commons.discovery.ResourceClassIterator	0.015098108
org.apache.commons.discovery.tools.Service	0.014745935
org.apache.commons.discovery.test.TestImpl3	0.014506218
org.apache.commons.discovery.test.TestInterface3	0.014398634
org.apache.commons.discovery.log.DiscoveryLogFactory	0.014199079
org.apache.commons.discovery.resource.names. DiscoverNamesInManagedProperties	0.013814332
org.apache.commons.discovery.tools.ResourceUtils	0.013400796
org.apache.commons.discovery.resource.names.DiscoverServiceNames	0.013146523
org.apache.commons.discovery.jdk.JDKHooks	0.01313916
org.apache.commons.discovery.ResourceClass	0.013019088
org.apache.commons.discovery.tools.DefaultClassHolder	0.012926248
org.apache.commons.discovery.resource.names.DiscoverNamesInSystemProperties	0.012559248

Table A-1: Apache Commons Discovery Complete Results (cont.)

Class	Complexity
org.apache.commons.discovery.ant.ServiceDiscoveryTask	0.011885874
org.apache.commons.discovery.tools.PropertiesHolder	0.011795014
org.apache.commons.discovery.tools.EnvironmentCache	0.01168743
org.apache.commons.discovery.ResourceClassDiscover	0.011365045
org.apache.commons.discovery.ResourceDiscover	0.011236807
org.apache.commons.discovery.ResourceIterator	0.011167205
org.apache.commons.discovery.test.TestImpl1_2	0.011108781
org.apache.commons.discovery.test.TestImpl1_1	0.011108781
org.apache.commons.discovery.test.TestImpl2_1	0.011108781
org.apache.commons.discovery.test.TestImpl2_2	0.011108781
org.apache.commons.discovery.ResourceNameDiscover	0.011108781
org.apache.commons.discovery.ResourceNameIterator	0.011059621
org.apache.commons.discovery.test.TestInterface2	0.010720913
org.apache.commons.discovery.test.TestInterface1	0.010720913

Table A-2: Apache Commons Net Complete Results

Class	Complexity
org.apache.commons.net.ftp.FTPClient	0.021580391
org.apache.commons.net.ftp.FTPSClient	0.016293543
org.apache.commons.net.ftp.FTP	0.01554193
org.apache.commons.net.telnet.Telnet	0.015090213
org.apache.commons.net.nntp.NNTPClient	0.014911443
org.apache.commons.net.SocketClient	0.014675531
org.apache.commons.net.util.Base64	0.012940319
org.apache.commons.net.telnet.TelnetInputStream	0.012891066
org.apache.commons.net.nntp.NNTP	0.011719148
org.apache.commons.net.tftp.TFTPServer	0.011667075
org.apache.commons.net.ntp.NtpV3Impl	0.01086878
org.apache.commons.net.ftp.parser.FTPParseTestFramework	0.01013838
org.apache.commons.net.smtp.SMTP	0.010018276
org.apache.commons.net.util.SubnetUtils	0.009100571
org.apache.commons.net.ftp.parser.UnixFTPEntryParserTest	0.009099659
org.apache.commons.net.DatagramSocketClient	0.009096342
org.apache.commons.net.telnet.TelnetClientTest	0.008873909
org.apache.commons.net.pop3.POP3Client	0.008553788
org.apache.commons.net.ftp.FTPFile	0.008490155
org.apache.commons.net.nntp.Article	0.008436924

Table A-2: Apache Commons Net Complete Results (cont.)

Class	Complexity
org.apache.commons.net.imap.IMAP	0.008413003
org.apache.commons.net.ftp.parser.ConfigurableFTPFileEntryParserImpl	0.008397013
examples.ftp.FTPClientExample	0.00839517
org.apache.commons.net.imap.IMAPClient	0.00828875
org.apache.commons.net.telnet.TelnetClient	0.008137435
org.apache.commons.net.ftp.parser.CompositeFTPParseTestFramework	0.007705276
org.apache.commons.net.ntp.NtpV3Packet	0.007496804
examples.mail.IMAPExportMbox	0.007320377
org.apache.commons.net.ftp.parser.NTFTPEntryParserTest	0.006960988
org.apache.commons.net.ftp.FTPFileEntryParserImpl	0.006796025
org.apache.commons.net.io.FromNetASCIIInputStream	0.006730695
org.apache.commons.net.tftp.TFTPClient	0.006690027
org.apache.commons.net.telnet.TelnetOptionHandler	0.006672937
org.apache.commons.net.nntp.Threader	0.006664281
org.apache.commons.net.smtp.SMTPClient	0.006641584
org.apache.commons.net.ftp.parser.FTPTimestampParserImplTest	0.006553748
org.apache.commons.net.ftp.parser.FTPTimestampParserImpl	0.006478955
org.apache.commons.net.util.Base64Test	0.006407771
org.apache.commons.net.pop3.POP3SClient	0.006203275
org.apache.commons.net.ftp.FTPClientConfig	0.006134544
org.apache.commons.net.ftp.parser.MVSFTPEntryParserTest	0.006092357
org.apache.commons.net.ftp.parser.VMSFTPEntryParserTest	0.006089698
org.apache.commons.net.imap.IMAPSClient	0.006072272
org.apache.commons.net.smtp.SMTPSClient	0.006055146
org.apache.commons.net.io.DotTerminatedMessageReader	0.005957737
org.apache.commons.net.io.ToNetASCIIInputStream	0.00582861
org.apache.commons.net.pop3.POP3	0.00580719
org.apache.commons.net.ftp.parser.OS400FTPEntryParserTest	0.005702488
org.apache.commons.net.ntp.TimeStamp	0.005667479
org.apache.commons.net.bsd.RCommandClient	0.005651486
org.apache.commons.net.ftp.parser.DefaultFTPFileEntryParserFactory	0.005599539
org.apache.commons.net.ftp.parser.OS400FTPEntryParserAdditionalTest	0.005459759
org.apache.commons.net.pop3.POP3ClientCommandsTest	0.00543855
org.apache.commons.net.ftp.parser.MLSxEntryParserTest	0.005300739
org.apache.commons.net.ftp.parser.MVSFTPEntryParser	0.005195793
org.apache.commons.net.tftp.TFTPServerMain	0.005147662
org.apache.commons.net.io.FromNetASCIIOutputStream	0.005064577
org.apache.commons.net.telnet.TelnetOptionHandlerTestAbstract	0.004880222
org.apache.commons.net.ftp.parser.MacOsPeterFTPEntryParserTest	0.004812289

Table A-2: Apache Commons Net Complete Results (cont.)

Class	Complexity
org.apache.commons.net.ftp.FTPClientTest	0.004810389
org.apache.commons.net.ftp.parser.EnterpriseUnixFTPEntryParserTest	0.004798523
org.apache.commons.net.ntp.TimeInfo	0.004768239
org.apache.commons.net.io.DotTerminatedMessageWriter	0.004723431
org.apache.commons.net.ftp.parser.OS2FTPEntryParserTest	0.004703176
org.apache.commons.net.ftp.parser.MLSxEntryParser	0.004653667
org.apache.commons.net.ftp.parser.NetwareFTPEntryParserTest	0.004641854
org.apache.commons.net.ftp.ListingFunctionalTest	0.00460045
org.apache.commons.net.smtp.AuthenticatingSMTPClient	0.004450664
org.apache.commons.net.tftp.TFTPPacket	0.004427331
org.apache.commons.net.util.KeyManagerUtils	0.004408686
examples.telnet.TelnetClientExample	0.00433398
org.apache.commons.net.DefaultSocketFactory	0.004133019
org.apache.commons.net.ftp.parser.VMSFTPEntryParser	0.004075203
org.apache.commons.net.util.UtilTest	0.004008856
org.apache.commons.net.telnet.TelnetOutputStream	0.003959474
org.apache.commons.net.tftp.TFTP	0.003952112
org.apache.commons.net.io.Util	0.003896926
org.apache.commons.net.ftp.FTPSSocketFactory	0.003881538
org.apache.commons.net.imap.AuthenticatingIMAPClient	0.003790391
org.apache.commons.net.io.ToNetASCIIOutputStream	0.003765735
org.apache.commons.net.tftp.TFTPRequestPacket	0.00374839
org.apache.commons.net.finger.FingerClient	0.003623293
examples.ftp.TFTPExample	0.003597387
org.apache.commons.net.tftp.TFTPDataPacket	0.003524774
org.apache.commons.net.discard.DiscardUDPClient	0.003459155
examples.ntp.SimpleNTPServer	0.003456341
examples.mail.Utils	0.003455081
org.apache.commons.net.ftp.parser.MLSDComparison	0.003434628
org.apache.commons.net.tftp.TFTPTest	0.003420307
org.apache.commons.net.discard.DiscardTCPClient	0.003374514
org.apache.commons.net.nntp.NewGroupsOrNewsQuery	0.003362353
org.apache.commons.net.ftp.FTPHTTPClient	0.003296594
org.apache.commons.net.PrintCommandListener	0.003274197
org.apache.commons.net.ftp.FTPListParseEngine	0.003256298
org.apache.commons.net.telnet.WindowSizeOptionHandlerTest	0.003228456
org.apache.commons.net.SubnetUtilsTest	0.003224253
org.apache.commons.net.ftp.parser.OS400FTPEntryParser	0.003223848
org.apache.commons.net.telnet.TerminalTypeOptionHandlerTest	0.003206285

Table A-2: Apache Commons Net Complete Results (cont.)

Class	Complexity
examples.mail.IMAPUtils	0.003182249
org.apache.commons.net.nntp.NewsgroupInfo	0.003169634
org.apache.commons.net.ftp.parser.DefaultFTPFileEntryParserFactoryTest	0.003151971
org.apache.commons.net.ftp.parser.UnixFTPEntryParser	0.003144253
examples.mail.IMAPImportMbox	0.003127034
org.apache.commons.net.io.DotTerminatedMessageReaderTest	0.003126186
org.apache.commons.net.telnet.SuppressGAOptionHandlerTest	0.003103775
org.apache.commons.net.telnet.EchoOptionHandlerTest	0.003103775
org.apache.commons.net.telnet.SimpleOptionHandlerTest	0.003103775
org.apache.commons.net.ftp.FTPSServerSocketFactory	0.003095911
examples.ftp.ServerToServerFTP	0.003068938
org.apache.commons.net.io.SocketOutputStream	0.003038177
org.apache.commons.net.bsd.RExecClient	0.002942973
org.apache.commons.net.tftp.TFTPErrorPacket	0.002865852
org.apache.commons.net.tftp.TFTPAckPacket	0.002836662
examples.ntp.NTPClient	0.002832258
examples.mail.POP3Mail	0.002832258
org.apache.commons.net.imap.IMAPReply	0.002809824
org.apache.commons.net.time.TimeTestSimpleServer	0.002801518
org.apache.commons.net.ftp.parser.MacOsPeterFTPEntryParser	0.002781866
org.apache.commons.net.ntp.TestNtpClient	0.002743416
org.apache.commons.net.smtp.SimpleSMTPHeaderTestCase	0.002692524
examples.MainTest	0.002674615
org.apache.commons.net.ftp.parser.RegexFTPFileEntryParserImpl	0.002655644
org.apache.commons.net.ftp.FTPClientConfigFunctionalTest	0.002615798
org.apache.commons.net.ntp.TestNtpPacket	0.002615395
org.apache.commons.net.ftp.parser.NTFTPEntryParser	0.002572068
org.apache.commons.net.nntp.ThreadContainer	0.002561492
org.apache.commons.net.ftp.FTPClientConfigTest	0.002541865
org.apache.commons.net.pop3.POP3ClientTest	0.002515436
org.apache.commons.net.ftp.parser.VMSVersioningFTPEntryParser	0.002513368
examples.nntp.PostMessage	0.002512527
org.apache.commons.net.ntp.NtpUtils	0.00250731
org.apache.commons.net.imap.IMAPCommand	0.002502423
examples.nntp.ExtendedNNTPOps	0.002500546
org.apache.commons.net.nntp.Threadable	0.002499958
org.apache.commons.net.ntp.TestTimeInfo	0.002465561
org.apache.commons.net.nntp.ReplyIterator	0.002461045
examples.Main	0.002446443
org.apache.commons.net.smtp.SimpleSMTPHeader	0.002440846

Table A-2: Apache Commons Net Complete Results (cont.)

Class	Complexity
org.apache.commons.net.telnet.TelnetTestSimpleServer	0.002438497
org.apache.commons.net.ftp.parser.DownloadListings	0.002414105
examples.nntp.ArticleReader	0.002411993
examples.nntp.MessageThreading	0.002402223
org.apache.commons.net.ntp.NTPUDPClient	0.002395046
org.apache.commons.net.telnet.TelnetClientFunctionalTest	0.002387236
org.apache.commons.net.ftp.parser.NetwareFTPEntryParser	0.00237709
org.apache.commons.net.util.TrustManagerUtils	0.002372275
org.apache.commons.net.pop3.POP3ConstructorTest	0.00235634
org.apache.commons.net.pop3.ExtendedPOP3Client	0.002353193
org.apache.commons.net.nntp.TestThreader	0.00235225
org.apache.commons.net.ftp.FTPReply	0.002350662
examples.unix.chargen	0.002344637
org.apache.commons.net.ftp.parser.OS2FTPEntryParser	0.002342644
org.apache.commons.net.time.TimeTCPClientTest	0.0023287
examples.unix.echo	0.00232805
org.apache.commons.net.ProtocolCommandSupport	0.002324378
org.apache.commons.net.nntp.SimpleNNTPHeader	0.002322905
org.apache.commons.net.io.ToNetASCIIInputStreamTest	0.002321057
org.apache.commons.net.telnet.WindowSizeOptionHandler	0.002317322
examples.mail.SMTPMail	0.002308243
org.apache.commons.net.io.CopyStreamAdapter	0.002303639
org.apache.commons.net.ProtocolCommandEvent	0.002290312
examples.unix.finger	0.00228972
org.apache.commons.net.echo.EchoUDPClient	0.002287433
org.apache.commons.net.nntp.NNTPReply	0.002271496
org.apache.commons.net.smtp.SMTPReply	0.002271496
org.apache.commons.net.telnet.TerminalTypeOptionHandler	0.002270214
org.apache.commons.net.ftp.parser.CompositeFileEntryParser	0.002268096
org.apache.commons.net.ntp.TimeStampTest	0.002264348
org.apache.commons.net.ftp.parser.FTPConfigEntryParserTest	0.002264206
org.apache.commons.net.tftp.TFTPServerPathTest	0.002260221
org.apache.commons.net.nntp.ArticleIterator	0.002249104
org.apache.commons.net.nntp.NewsgroupIterator	0.002249104
examples.ntp.TimeClient	0.002248954
examples.unix.daytime	0.002248954
examples.unix.rdate	0.002248954

Table A-2: Apache Commons Net Complete Results (cont.)

Class	Complexity
org.apache.commons.net.ftp.parser.EnterpriseUnixFTPEntryParser	0.002239503
org.apache.commons.net.time.TimeUDPClient	0.002231974
org.apache.commons.net.chargen.CharGenUDPClient	0.002231974
org.apache.commons.net.io.CRLFLineReader	0.002219274
org.apache.commons.net.util.SSLSocketUtils	0.00221667
org.apache.commons.net.pop3.POP3MessageInfo	0.002214177
org.apache.commons.net.SocketClientFunctionalTest	0.002212354
org.apache.commons.net.util.ListenerList	0.002206837
org.apache.commons.net.io.CopyStreamEvent	0.002206837
examples.nntp.ListNewsgroups	0.002206706
org.apache.commons.net.tftp.TFTPReadRequestPacket	0.002204898
org.apache.commons.net.tftp.TFTPWriteRequestPacket	0.002204898
org.apache.commons.net.telnet.TelnetTestResponder	0.002203077
org.apache.commons.net.ftp.FTPFileFilters	0.002199636
org.apache.commons.net.daytime.DaytimeTCPClient	0.002197747
org.apache.commons.net.io.SocketInputStream	0.002185611
org.apache.commons.net.SocketClientTest	0.002184085
org.apache.commons.net.util.SSLContextUtils	0.002183502
org.apache.commons.net.imap.IMAPTest	0.00218056
org.apache.commons.net.smtp.RelayPath	0.002179467
org.apache.commons.net.ftp.TestConnectTimeout	0.002175
examples.mail.IMAPMail	0.002175
org.apache.commons.net.telnet.TelnetOption	0.002174687
org.apache.commons.net.ftp.parser.FTPFileEntryParserFactory	0.002173867
org.apache.commons.net.daytime.DaytimeUDPClient	0.002173867
org.apache.commons.net.telnet.InvalidTelnetOptionExceptionTest	0.00217373
org.apache.commons.net.DefaultDatagramSocketFactory	0.002173103
org.apache.commons.net.time.TimeTCPClient	0.002173103
org.apache.commons.net.whois.WhoisClient	0.002173103
org.apache.commons.net.ftp.FTPFileEntryParser	0.002173103
org.apache.commons.net.bsd.RLoginClient	0.002173103
org.apache.commons.net.ftp.FTPCommand	0.002170652
examples.unix.fwhois	0.002170076
examples.unix.rlogin	0.002170076
examples.unix.rexec	0.002170076
examples.unix.rshell	0.002170076
org.apache.commons.net.ftp.FTPSTrustManager	0.002169127
org.apache.commons.net.telnet.InvalidTelnetOptionException	0.002163258
examples.cidr.SubnetUtilsExample	0.002163242

Table A-2: Apache Commons Net Complete Results (cont.)

Class	Complexity
examples.util.IOUtil	0.002158378
org.apache.commons.net.ftp.parser.ParserInitializationException	0.002157369
org.apache.commons.net.DatagramSocketFactory	0.002157369
examples.nntp.NNTPUtils	0.002156681
org.apache.commons.net.pop3.POP3Command	0.00215548
org.apache.commons.net.smtp.SMTPCommand	0.002155343
org.apache.commons.net.io.CopyStreamException	0.00215347
org.apache.commons.net.telnet.TelnetCommand	0.00215347
org.apache.commons.net.util.Charsets	0.002149919
org.apache.commons.net.telnet.EchoOptionHandler	0.002147334
org.apache.commons.net.telnet.SimpleOptionHandler	0.002147334
org.apache.commons.net.echo.EchoTCPClient	0.002147334
org.apache.commons.net.ftp.FTPCommandTest	0.002147334
org.apache.commons.net.chargen.CharGenTCPClient	0.002147334
org.apache.commons.net.io.CopyStreamListener	0.002147334
org.apache.commons.net.telnet.TelnetOptionTest	0.002147334
org.apache.commons.net.telnet.SuppressGAOptionHandler	0.002147334
org.apache.commons.net.ProtocolCommandListener	0.002147334
examples.telnet.WeatherTelnet	0.002136709
org.apache.commons.net.smtp.SMTPConnectionClosedException	0.0021316
org.apache.commons.net.nntp.NNTPConnectionClosedException	0.0021316
org.apache.commons.net.MalformedServerReplyException	0.0021316
org.apache.commons.net.ftp.FTPConnectionClosedException	0.0021316
org.apache.commons.net.tftp.TFTPPacketException	0.0021316
org.apache.commons.net.nntp.NNTPCommand	0.002131463
org.apache.commons.net.ftp.Configurable	0.002119065
org.apache.commons.net.ftp.FTPFileFilter	0.002119065
org.apache.commons.net.ftp.FTPCmd	0.002119065
org.apache.commons.net.pop3.POP3Constants	0.002103333
org.apache.commons.net.nntp.ArticleInfo	0.002103331
org.apache.commons.net.telnet.TelnetInputListener	0.002103331
org.apache.commons.net.ftp.parser.FTPTimestampParser	0.002103331
org.apache.commons.net.pop3.POP3Reply	0.002103331
org.apache.commons.net.telnet.TelnetNotificationHandler	0.002103331
org.apache.commons.net.ftp.FTPSCommand	0.002103331
org.apache.commons.net.nntp.ArticlePointer	0.002060035
org.apache.commons.io.FileUtilsTestCase	0.016903606
org.apache.commons.io.FileUtils	0.016316664
org.apache.commons.io.IOUtils	0.016117173

Table A-3: Apache Commons IO Complete Results

Class	Complexity
org.apache.commons.io.filefilter.FileFilterTestCase	0.015649916
org.apache.commons.io.IOUtilsTestCase	0.015330946
org.apache.commons.io.serialization.ValidatingObjectInputStreamTest	0.014415967
org.apache.commons.io.testtools.TestUtils	0.014337154
org.apache.commons.io.FilenameUtils	0.013945298
org.apache.commons.io.DirectoryWalkerTestCase	0.0127693
org.apache.commons.io.input.BOMInputStreamTest	0.012704102
org.apache.commons.io.DirectoryWalkerTestCaseJava4	0.012549659
org.apache.commons.io.FileSystemUtilsTestCase	0.011948133
org.apache.commons.io.IOUtilsWriteTestCase	0.011924496
org.apache.commons.io.filefilter.FileFilterUtils	0.011760021
org.apache.commons.io.input.compatibility.XmlStreamReader	0.010849067
org.apache.commons.io.input.CharSequenceInputStreamTest	0.010830476
org.apache.commons.io.serialization.ValidatingObjectInputStream	0.010779692
org.apache.commons.io.input.XmlStreamReader	0.010084407
org.apache.commons.io.FilenameUtilsTestCase	0.009678092
org.apache.commons.io.filefilter.AbstractFileFilter	0.009533217
org.apache.commons.io.comparator.AbstractFileComparator	0.009250867
org.apache.commons.io.input.ProxyInputStream	0.009038351
org.apache.commons.io.comparator.ComparatorAbstractTestCase	0.00888123
org.apache.commons.io.input.BoundedReaderTest	0.00882985
org.apache.commons.io.input.BOMInputStream	0.0087074
org.apache.commons.io.testtools.FileBasedTestCase	0.008229246
org.apache.commons.io.input.UnixLineEndingInputStreamTest	0.007989864
org.apache.commons.io.monitor.FileAlterationObserver	0.007336224
org.apache.commons.io.LineIteratorTestCase	0.007047382
org.apache.commons.io.input.WindowsLineEndingInputStreamTest	0.007031532
org.apache.commons.io.IOUtilsCopyTestCase	0.006907885
org.apache.commons.io.input.ProxyReader	0.006817336
org.apache.commons.io.EndianUtils	0.006657546
org.apache.commons.io.IOCaseTestCase	0.006282469
org.apache.commons.io.output.ProxyWriter	0.006273698
org.apache.commons.io.input.XmlStreamReaderTest	0.006140351
org.apache.commons.io.FileSystemUtils	0.006103032
org.apache.commons.io.input.WindowsLineEndingInputStream	0.006005894
org.apache.commons.io.output.ProxyWriterTest	0.005910747
org.apache.commons.io.output.ProxyOutputStream	0.005772002

Table A-3: Apache Commons IO Complete Results (cont.)

Class	Complexity
org.apache.commons.io.Java7Support	0.005688057
org.apache.commons.io.serialization.ClosingBase	0.0056525
org.apache.commons.io.input.NullInputStream	0.005595464
org.apache.commons.io.input.UnixLineEndingInputStream	0.005578287
org.apache.commons.io.output.LockableFileWriter	0.005518825
org.apache.commons.io.input.TailerTest	0.005473823
org.apache.commons.io.input.CharSequenceInputStream	0.005462202
org.apache.commons.io.filefilter.AndFileFilterTestCase	0.005389319
org.apache.commons.io.filefilter.OrFileFilterTestCase	0.005389319
org.apache.commons.io.EndianUtilsTest	0.005355701
org.apache.commons.io.input.Tailer	0.005319407
org.apache.commons.io.filefilter.IOFileFilterAbstractTestCase	0.005318248
org.apache.commons.io.output.FileWriterWithEncoding	0.005247219
org.apache.commons.io.input.XmlStreamReaderUtilitiesTest	0.005176135
org.apache.commons.io.DirectoryWalker	0.005060381
org.apache.commons.io.output.TeeOutputStream	0.005051376
org.apache.commons.io.output.ByteArrayOutputStream	0.005029391
org.apache.commons.io.input.BoundedReader	0.004994633
org.apache.commons.io.input.ReversedLinesFileReader	0.004982245
org.apache.commons.io.input.NullReader	0.004923217
org.apache.commons.io.serialization.MoreComplexObjectTest	0.00456175
org.apache.commons.io.input.ReversedLinesFileReaderTestParamBlockSize	0.004500698
org.apache.commons.io.input.BoundedInputStream	0.004495606
org.apache.commons.io.input.TeeInputStream	0.004397086
org.apache.commons.io.filefilter.ConditionalFileFilterAbstractTestCase	0.004376479
org.apache.commons.io.output.NullWriter	0.004247098
org.apache.commons.io.serialization.RegexpClassNameMatcherTest	0.004234776
org.apache.commons.io.input.NullInputStreamTest	0.004187661
org.apache.commons.io.input.NullReaderTest	0.004187661
org.apache.commons.io.monitor.FileEntry	0.004134965
org.apache.commons.io.output.WriterOutputStream	0.003986958
org.apache.commons.io.output.ChunkedWriterTest	0.003974232
org.apache.commons.io.output.ChunkedOutputStreamTest	0.003974232
org.apache.commons.io.FileCleaningTrackerTestCase	0.003893908
org.apache.commons.io.SelectorAdapter	0.003873766
org.apache.commons.io.monitor.AbstractMonitorTestCase	0.003869757
org.apache.commons.io.input.ReaderInputStream	0.003784267
org.apache.commons.io.output.FileWriterWithEncodingTest	0.003754224

Table A-3: Apache Commons IO Complete Results (cont.)

Class	Complexity
org.apache.commons.io.input.CharSequenceReader	0.003720262
org.apache.commons.io.input.SwappedDataInputStreamTest	0.003715909
org.apache.commons.io.CopyUtilsTest	0.003699472
org.apache.commons.io.output.StringBuilderWriter	0.003695729
org.apache.commons.io.input.SwappedDataInputStream	0.003671402
org.apache.commons.io.output.XmlStreamWriterTest	0.003654082
org.apache.commons.io.monitor.FileAlterationMonitor	0.003522025
org.apache.commons.io.monitor.CollectionFileListener	0.003489971
org.apache.commons.io.FileCleaningTracker	0.00346084
org.apache.commons.io.output.ThresholdingOutputStream	0.003449323
org.apache.commons.io.IOCase	0.003444718
org.apache.commons.io.output.XmlStreamWriter	0.003426249
org.apache.commons.io.output.DeferredFileOutputStreamTest	0.003376352
org.apache.commons.io.DemuxTestCase	0.003326222
org.apache.commons.io.filefilter.AndFileFilter	0.003324121
org.apache.commons.io.output.StringBuilderWriterTest	0.00332064
org.apache.commons.io.output.WriterOutputStreamTest	0.003307514
org.apache.commons.io.output.DeferredFileOutputStream	0.003265107
org.apache.commons.io.FileUtilsCleanSymlinksTestCase	0.003208181
org.apache.commons.io.filefilter.SuffixFileFilter	0.003192732
org.apache.commons.io.filefilter.NameFileFilter	0.003192732
org.apache.commons.io.filefilter.WildcardFileFilter	0.003192732
org.apache.commons.io.filefilter.PrefixFileFilter	0.003192732
org.apache.commons.io.filefilter.OrFileFilter	0.003158471
org.apache.commons.io.input.CountingInputStreamTest	0.003155242
org.apache.commons.io.HexDumpTest	0.003148088
org.apache.commons.io.CopyUtils	0.003147665
org.apache.commons.io.serialization.WildcardClassNameMatcherTest	0.003138028
org.apache.commons.io.output.AppendableOutputStreamTest	0.003136016
org.apache.commons.io.monitor.FileAlterationObserverTestCase	0.003128468
org.apache.commons.io.output.TeeOutputStreamTest	0.00307352
org.apache.commons.io.output.ChunkedOutputStream	0.003044689
org.apache.commons.io.output.ChunkedWriter	0.003044689
org.apache.commons.io.FilenameUtilsWildcardTestCase	0.003040203
org.apache.commons.io.FileUtilsDirectoryContainsTestCase	0.003037958
org.apache.commons.io.FileUtilsCleanDirectoryTestCase	0.003027629
org.apache.commons.io.serialization.RegexpClassNameMatcher	0.00302733
org.apache.commons.io.serialization.MockSerializedClass	0.003026094

Table A-3: Apache Commons IO Complete Results (cont.)

Class	Complexity
org.apache.commons.io.output.LockableFileWriterTest	0.002999463
org.apache.commons.io.output.AppendableOutputStream	0.002996442
org.apache.commons.io.testtools.YellOnFlushAndCloseOutputStream	0.002992417
org.apache.commons.io.output.ByteArrayOutputStreamTestCase	0.00296576
org.apache.commons.io.FileUtilsListFilesTestCase	0.002964747
org.apache.commons.io.LineIterator	0.002958652
org.apache.commons.io.input.ReaderInputStreamTest	0.002954808
org.apache.commons.io.ByteOrderMark	0.002923994
org.apache.commons.io.input.BrokenInputStream	0.002916438
org.apache.commons.io.input.CountingInputStream	0.002914868
org.apache.commons.io.FileDeleteStrategyTestCase	0.002911422
org.apache.commons.io.ByteOrderMarkTestCase	0.002903532
org.apache.commons.io.input.ProxyReaderTest	0.002878199
org.apache.commons.io.comparator.DirectoryFileComparatorTest	0.002862395
org.apache.commons.io.filefilter.MagicNumberFileFilter	0.002856459
org.apache.commons.io.input.ClassLoaderObjectInputStreamTest	0.002846789
org.apache.commons.io.Java7SupportTest	0.002844903
org.apache.commons.io.serialization.FullClassNameMatcherTest	0.002830781
org.apache.commons.io.monitor.FileAlterationMonitorTestCase	0.002828953
org.apache.commons.io.FileUtilsFileNewerTestCase	0.002812184
org.apache.commons.io.input.AutoCloseInputStream	0.002798667
org.apache.commons.io.serialization.FullClassNameMatcher	0.002779937
org.apache.commons.io.serialization.WildcardClassNameMatcher	0.002779937
org.apache.commons.io.filefilter.WildcardFilter	0.002775693
org.apache.commons.io.comparator.CompositeFileComparatorTest	0.002769098
org.apache.commons.io.serialization.MoreComplexObject	0.002765458
org.apache.commons.io.filefilter.RegexFileFilterTestCase	0.00274859
org.apache.commons.io.output.TaggedOutputStreamTest	0.002734944
org.apache.commons.io.input.CharSequenceReaderTest	0.002727048
org.apache.commons.io.input.TaggedInputStreamTest	0.002707853
org.apache.commons.io.HexDump	0.002684285
org.apache.commons.io.input.TeeInputStreamTest	0.0026725
org.apache.commons.io.filefilter.AgeFileFilter	0.002666128
org.apache.commons.io.input.ReversedLinesFileReaderTestParamFile	0.002662505
org.apache.commons.io.output.CountingOutputStreamTest	0.002659734
org.apache.commons.io.output.ThresholdingOutputStreamTest	0.0026529
org.apache.commons.io.FileDeleteStrategy	0.002651692
org.apache.commons.io.input.BrokenInputStreamTest	0.002641162

Table A-3: Apache Commons IO Complete Results (cont.)

Class	Complexity
org.apache.commons.io.comparator.SizeFileComparatorTest	0.002639585
org.apache.commons.io.filefilter.DelegateFileFilter	0.002638345
org.apache.commons.io.monitor.FileAlterationListenerAdaptor	0.00263223
org.apache.commons.io.filefilter.RegexFileFilter	0.002625535
org.apache.commons.io.CharsetsTestCase	0.002617608
org.apache.commons.io.monitor.FileAlterationListener	0.002617608
org.apache.commons.io.comparator.NameFileComparatorTest	0.00261673
org.apache.commons.io.comparator.ExtensionFileComparatorTest	0.00261673
org.apache.commons.io.comparator.PathFileComparatorTest	0.00261673
org.apache.commons.io.comparator.SizeFileComparator	0.002612686
org.apache.commons.io.input.AutoCloseInputStreamTest	0.002608537
org.apache.commons.io.comparator.CompositeFileComparator	0.002589479
org.apache.commons.io.output.CountingOutputStream	0.002581309
org.apache.commons.io.FileCleaner	0.002576642
org.apache.commons.io.output.BrokenOutputStreamTest	0.002572538
org.apache.commons.io.output.ProxyOutputStreamTest	0.002571039
org.apache.commons.io.comparator.LastModifiedFileComparatorTest	0.002567328
org.apache.commons.io.comparator.ExtensionFileComparator	0.002567077
org.apache.commons.io.comparator.PathFileComparator	0.002567077
org.apache.commons.io.comparator.NameFileComparator	0.002567077
org.apache.commons.io.ThreadMonitor	0.002566388
org.apache.commons.io.input.BoundedInputStreamTest	0.002564586
org.apache.commons.io.filefilter.NotFileFilter	0.002561841
org.apache.commons.io.output.DemuxOutputStream	0.002560296
org.apache.commons.io.TaggedIOException	0.00255698
org.apache.commons.io.input.XmlStreamReaderException	0.002556619
org.apache.commons.io.output.BrokenOutputStream	0.002543733
org.apache.commons.io.filefilter.SizeFileFilter	0.002541361
org.apache.commons.io.input.TailerListenerAdapter	0.002539399
org.apache.commons.io.ThreadMonitorTestCase	0.002532783
org.apache.commons.io.filefilter.CanReadFileFilter	0.002531695
org.apache.commons.io.output.NullOutputStream	0.002518868
org.apache.commons.io.filefilter.EmptyFileFilter	0.002514274
org.apache.commons.io.FileUtilsWaitForTestCase	0.002514074
org.apache.commons.io.input.compatibility. XmlStreamReaderUtilitiesCompatibilityTest	0.002513971
org.apache.commons.io.comparator.DefaultFileComparatorTest	0.002513303
org.apache.commons.io.input.TaggedInputStream	0.002509094
org.apache.commons.io.output.TaggedOutputStream	0.002509094

Table A-3: Apache Commons IO Complete Results (cont.)

Class	Complexity
org.apache.commons.io.input.ClassLoaderObjectInputStream	0.002507577
org.apache.commons.io.input.ReversedLinesFileReaderTestSimple	0.002507315
org.apache.commons.io.input.DemuxInputStream	0.002499187
org.apache.commons.io.input.TailerListener	0.002498986
org.apache.commons.io.testtools.YellOnCloseInputStream	0.002493231
org.apache.commons.io.filefilter.HiddenFileFilter	0.00248963
org.apache.commons.io.filefilter.CanWriteFileFilter	0.00248963
org.apache.commons.io.comparator.ReverseComparator	0.002485637
org.apache.commons.io.output.CloseShieldOutputStreamTest	0.002484793
org.apache.commons.io.FileCleanerTestCase	0.002482172
org.apache.commons.io.input.CloseShieldInputStream	0.002476051
org.apache.commons.io.output.CloseShieldOutputStream	0.002476051
org.apache.commons.io.input.CloseShieldInputStreamTest	0.002473844
org.apache.commons.io.IOExceptionWithCauseTestCase	0.002470781
org.apache.commons.io.output.NullWriterTest	0.002470114
org.apache.commons.io.filefilter.ConditionalFileFilter	0.002467245
org.apache.commons.io.filefilter.DirectoryFileFilter	0.002457203
org.apache.commons.io.filefilter.FileFileFilter	0.002457203
org.apache.commons.io.filefilter.TrueFileFilter	0.002453601
org.apache.commons.io.filefilter.FalseFileFilter	0.002453601
org.apache.commons.io.output.NullOutputStreamTest	0.002452933
org.apache.commons.io.comparator.DirectoryFileComparator	0.002452608
org.apache.commons.io.comparator.LastModifiedFileComparator	0.002447223
org.apache.commons.io.Charsets	0.002441727
org.apache.commons.io.input.compatibility.XmlStreamReaderException	0.002437075
org.apache.commons.io.serialization.ClassNameMatcher	0.002422532
org.apache.commons.io.FileExistsException	0.002422403
org.apache.commons.io.IOExceptionWithCause	0.00241146
org.apache.commons.io.comparator.DefaultFileComparator	0.002405037
org.apache.commons.io.output.ClosedOutputStream	0.002402472
org.apache.commons.io.input.ClosedInputStream	0.002402472
org.apache.commons.io.output.ClosedOutputStreamTest	0.002397024
org.apache.commons.io.input.ClosedInputStreamTest	0.002387856
org.apache.commons.io.TaggedIOExceptionTest	0.002387856
org.apache.commons.io.filefilter.IOFileFilter	0.002379586

Table A-4: Apache Commons Compression Complete Results

Class	Complexity
org.apache.commons.compress.compressors.lz4. BlockLZ4CompressorOutputStream	0.01598046
org.apache.commons.compress.compressors.lz4. FramedLZ4CompressorInputStreamTest	0.014487463
org.apache.commons.compress.compressors.lz77support.LZ77Compressor	0.012797215
org.apache.commons.compress.archivers.zip.Zip64SupportIT	0.011897318
org.apache.commons.compress.archivers.zip.ZipArchiveOutputStream	0.011321303
org.apache.commons.compress.compressors.lz4. FramedLZ4CompressorInputStream	0.010814482
org.apache.commons.compress.archivers.tar.TarArchiveEntry	0.010788506
org.apache.commons.compress.compressors.lz4. BlockLZ4CompressorOutputStreamTest	0.010576681
org.apache.commons.compress.archivers.zip.ZipArchiveEntry	0.010552864
org.apache.commons.compress.compressors.snappy. SnappyCompressorOutputStream	0.01027011
org.apache.commons.compress.utils.ByteUtilsTest	0.010002203
org.apache.commons.compress.compressors.lz77support. LZ77CompressorTest	0.009962006
org.apache.commons.compress.archivers.zip.ZipFile	0.009658721
org.apache.commons.compress.AbstractTestCase	0.009208974
org.apache.commons.compress.archivers.zip.ZipArchiveInputStream	0.009200479
org.apache.commons.compress.archivers.sevenz.SevenZFile	0.009168089
org.apache.commons.compress.compressors.lz77support.Parameters	0.009093729
org.apache.commons.compress.archivers.dump.DumpArchiveEntry	0.008756994
org.apache.commons.compress.archivers.sevenz.SevenZOutputFile	0.008689227
org.apache.commons.compress.archivers.cpio.CpioArchiveEntry	0.008468225
org.apache.commons.compress.compressors.lz4. FramedLZ4CompressorOutputStream	0.008386419
org.apache.commons.compress.compressors.bzip2. BZip2CompressorOutputStream	0.008272042
org.apache.commons.compress.compressors.bzip2. BZip2CompressorInputStream	0.008269106
org.apache.commons.compress.compressors.CompressorStreamFactory	0.008252447
org.apache.commons.compress.changes.ChangeSetTestCase	0.007985083
org.apache.commons.compress.archivers.tar.TarArchiveInputStream	0.007983767
org.apache.commons.compress.compressors.lz77support. AbstractLZ77CompressorInputStream	0.007967632
org.apache.commons.compress.archivers.sevenz.SevenZOutputFileTest	0.007091085
org.apache.commons.compress.utils.ByteUtils	0.007066399
org.apache.commons.compress.compressors.bzip2.BlockSort	0.00701258
org.apache.commons.compress.archivers.tar.TarArchiveOutputStream	0.006801084
org.apache.commons.compress.archivers.ArchiveStreamFactory	0.006511585
org.apache.commons.compress.compressors.CompressorInputStream	0.00641464
org.apache.commons.compress.compressors.CompressorOutputStream	0.006336947
org.apache.commons.compress.archivers.sevenz.SevenZArchiveEntry	0.006306565
org.apache.commons.compress.compressors.lz77support.ParametersTest	0.00605536

Table A-4: Apache Commons Compression Complete Results (cont.)

Class	Complexity
org.apache.commons.compress.archivers.cpio.CpioArchiveInputStream	0.005617947
org.apache.commons.compress.archivers.tar.TarUtils	0.005187552
org.apache.commons.compress.archivers.zip.X5455_ExtendedTimestamp	0.005084503
org.apache.commons.compress.archivers.dump.DumpArchiveInputStream	0.004895361
org.apache.commons.compress.archivers.zip.ResourceAlignmentExtraField	0.004695226
org.apache.commons.compress.archivers.tar.TarArchiveOutputStreamTest	0.00461305
org.apache.commons.compress.archivers.cpio.CpioArchiveOutputStream	0.004591675
org.apache.commons.compress.archivers.zip.StreamCompressor	0.004569643
org.apache.commons.compress.archivers.ArchiveInputStream	0.004556136
org.apache.commons.compress.archivers.zip.PKWareExtraHeader	0.004496888
org.apache.commons.compress.archivers.sevenz.Coders	0.004470246
org.apache.commons.compress.archivers.dump.DumpArchiveSummary	0.00445839
org.apache.commons.compress.archivers.ArchiveStreamFactoryTest	0.004313811
org.apache.commons.compress.archivers.ar.ArArchiveInputStream	0.004239629
org.apache.commons.compress.compressors.snappy. SnappyCompressorInputStream	0.004231958
org.apache.commons.compress.compressors.lzw.LZWInputStream	0.00408284
org.apache.commons.compress.archivers.zip.X000A_NTFS	0.004044311
org.apache.commons.compress.compressors.snappy. FramedSnappyCompressorInputStream	0.00401996
org.apache.commons.compress.archivers.zip.ZipUtil	0.004008441
org.apache.commons.compress.archivers.zip.UTF8ZipFilesTest	0.003992322
org.apache.commons.compress.compressors.snappy. FramedSnappyCompressorOutputStream	0.00394413
org.apache.commons.compress.compressors.snappy.SnappyRoundtripTest	0.003904257
org.apache.commons.compress.archivers.dump.DumpArchiveException	0.003829654
org.apache.commons.compress.compressors.lz4.XXHash32	0.003822271
org.apache.commons.compress.compressors.brotli. BrotliCompressorInputStream	0.003803592
org.apache.commons.compress.archivers.ZipTestCase	0.003801832
org.apache.commons.compress.archivers.arj.ArjArchiveInputStream	0.00378534
org.apache.commons.compress.archivers.zip. Zip64ExtendedInformationExtraField	0.003762584
org.apache.commons.compress.compressors.brotli. BrotliCompressorInputStreamTest	0.003719931
org.apache.commons.compress.archivers.zip.ZipFileTest	0.003679448
org.apache.commons.compress.archivers.sevenz.SevenZFileTest	0.003556617
org.apache.commons.compress.archivers.ArchiveOutputStream	0.003366502
org.apache.commons.compress.archivers.tar.TarUtilsTest	0.003334873
org.apache.commons.compress.compressors.gzip. GzipCompressorInputStream	0.003284876
org.apache.commons.compress.archivers.sevenz.CoderBase	0.003283159
org.apache.commons.compress.archivers.zip.AsiExtraField	0.003238954
org.apache.commons.compress.archivers.tar.TarArchiveInputStreamTest	0.003207873
org.apache.commons.compress.changes.ChangeSetPerformer	0.003176016
org.apache.commons.compress.archivers.zip.AbstractUnicodeExtraField	0.00315634

Table A-4: Apache Commons Compression Complete Results (cont.)

Class	Complexity
org.apache.commons.compress.compressors.lz4. BlockLZ4CompressorInputStream	0.003119839
org.apache.commons.compress.archivers.zip.ExtraFieldUtils	0.003113769
org.apache.commons.compress.archivers.zip. X5455_ExtendedTimestampTest	0.003085909
org.apache.commons.compress.utils.ArchiveUtils	0.003058745
org.apache.commons.compress.archivers.ar.ArArchiveOutputStream	0.003016018
org.apache.commons.compress.archivers.dump.TapeInputStream	0.00300001
org.apache.commons.compress.compressors.lz4. BlockLZ4CompressorRoundtripTest	0.002992771
org.apache.commons.compress.compressors.lz4. FramedLZ4CompressorRoundtripTest	0.002992771
org.apache.commons.compress.compressors.pack200. Pack200CompressorInputStream	0.002991196
org.apache.commons.compress.archivers.zip.GeneralPurposeBit	0.002753807
org.apache.commons.compress.archivers.zip.Simple8BitZipEncoding	0.002724805
org.apache.commons.compress.compressors.DetectCompressorTestCase	0.002702333
org.apache.commons.compress.utils.SeekableInMemoryByteChannel	0.002698618
org.apache.commons.compress.utils.ChecksumCalculatingInputStream	0.002685944
org.apache.commons.compress.compressors.GZipTestCase	0.002679434
org.apache.commons.compress.archivers.zip.X7875_NewUnix	0.002675644
org.apache.commons.compress.compressors.lz77support. AbstractLZ77CompressorInputStreamTest	0.002665464
org.apache.commons.compress.archivers.zip.ParallelScatterZipCreator	0.002659607
org.apache.commons.compress.archivers.zip.ZipArchiveInputStreamTest	0.002648736
org.apache.commons.compress.archivers.LongPathTest	0.002636317
org.apache.commons.compress.archivers.LongSymLinkTest	0.002636317
org.apache.commons.compress.archivers.ArTestCase	0.002567848
org.apache.commons.compress.compressors.Pack200TestCase	0.002537799
org.apache.commons.compress.archivers.TarTestCase	0.00251607
org.apache.commons.compress.utils.BitInputStreamTest	0.002513059
org.apache.commons.compress.archivers.ar.ArArchiveEntry	0.002484147
org.apache.commons.compress.IOMethodsTest	0.002457123
org.apache.commons.compress.archivers.zip.Lister	0.002450979
org.apache.commons.compress.archivers.sevenz.CLI	0.002442831
org.apache.commons.compress.compressors.gzip. GzipCompressorOutputStream	0.002424856
org.apache.commons.compress.archivers.zip.ZipEightByteInteger	0.002417323
org.apache.commons.compress.archivers.CpioTestCase	0.002415513
org.apache.commons.compress.utils.IOUtilsTest	0.002404912
org.apache.commons.compress.archivers.zip.ParallelScatterZipCreatorTest	0.002371846
org.apache.commons.compress.compressors.brotli.BrotliUtils	0.00236047
org.apache.commons.compress.archivers.zip. Zip64ExtendedInformationExtraFieldTest	0.002358287

Table A-4: Apache Commons Compression Complete Results (cont.)

Class	Complexity
org.apache.commons.compress.compressors.bzip2.BlockSortTest	0.002354754
org.apache.commons.compress.archivers.zip.ZipUtilTest	0.002319878
org.apache.commons.compress.archivers.zip.X7875_NewUnixTest	0.002311516
org.apache.commons.compress.changes.ChangeSet	0.002303914
org.apache.commons.compress.utils.IOUtils	0.002300523
org.apache.commons.compress.archivers.zip.ZipLong	0.002285175
org.apache.commons.compress.archivers.zip.BinaryTree	0.002277339
org.apache.commons.compress.compressors.z.ZCompressorInputStream	0.002276254
org.apache.commons.compress.archivers.zip.ZipArchiveEntryTest	0.002272476
org.apache.commons.compress.compressors.lz4.XXHash32Test	0.002252872
org.apache.commons.compress.archivers.sevenz.LZMA2Decoder	0.002250121
org.apache.commons.compress.archivers.zip.ZipShort	0.002249842
org.apache.commons.compress.archivers.zip.ZipEncodingHelper	0.002244895
org.apache.commons.compress.utils.SeekableInMemoryByteChannelTest	0.002244539
org.apache.commons.compress.archivers.zip.UnshrinkingInputStream	0.002241647
org.apache.commons.compress.compressors.snappy. FramedSnappyCompressorInputStreamTest	0.002226633
org.apache.commons.compress.archivers.zip.ExplodingInputStream	0.002225647
org.apache.commons.compress.compressors.lzma.LZMAUtils	0.002217047
org.apache.commons.compress.compressors.xz.XZUtils	0.002217047
org.apache.commons.compress.archivers.sevenz.Folder	0.00220501
org.apache.commons.compress.archivers.sevenz.LZMADecoder	0.002202522
org.apache.commons.compress.utils.BitInputStream	0.002193281
org.apache.commons.compress.compressors.FileNameUtil	0.002188507
org.apache.commons.compress.DetectArchiverTestCase	0.002184996
org.apache.commons.compress.archivers.ArchiveOutputStreamTest	0.002181935
org.apache.commons.compress.compressors.FramedSnappyTestCase	0.002181554
org.apache.commons.compress.archivers.zip.ScatterZipOutputStream	0.002171416
org.apache.commons.compress.archivers.zip.ExtraFieldUtilsTest	0.002168833
org.apache.commons.compress.compressors.xz.XZCompressorInputStream	0.002164308
org.apache.commons.compress.archivers.arj.ArjArchiveEntry	0.002158017
org.apache.commons.compress.archivers.DumpTestCase	0.002151413
org.apache.commons.compress.archivers.tar.TarLister	0.002150649
org.apache.commons.compress.archivers.tar.TarArchiveEntryTest	0.002146083
org.apache.commons.compress.archivers.zip.UnrecognizedExtraField	0.002141361
org.apache.commons.compress.ArchiveReadTest	0.002139738
org.apache.commons.compress.compressors.lz4.FactoryTest	0.002136096
org.apache.commons.compress.compressors.pack200. Pack200CompressorOutputStream	0.00213037
org.apache.commons.compress.archivers.SevenZTestCase	0.002124195

Table A-4: Apache Commons Compression Complete Results (cont.)

Class	Complexity
org.apache.commons.compress.compressors.bzip2. PythonTruncatedBzip2Test	0.002116245
org.apache.commons.compress.compressors.lzma. LZMACompressorInputStream	0.002115649
org.apache.commons.compress.MemoryLimitException	0.002115276
org.apache.commons.compress.archivers.zip.X0017_StrongEncryptionHeader	0.002113982
org.apache.commons.compress.compressors.gzip.GzipParameters	0.002110417
org.apache.commons.compress.compressors.ZTestCase	0.002108876
org.apache.commons.compress.archivers.zip.ScatterZipOutputStreamTest	0.00210526
org.apache.commons.compress.archivers.tar.BigFilesIT	0.002104897
org.apache.commons.compress.archivers.zip.JarMarker	0.002104694
org.apache.commons.compress.archivers.sevenz.DeltaDecoder	0.002093629
org.apache.commons.compress.utils.ChecksumVerifyingInputStream	0.002092181
org.apache.commons.compress.compressors.DeflateTestCase	0.002091647
org.apache.commons.compress.archivers.zip.ExplodeSupportTest	0.002091318
org.apache.commons.compress.archivers.sevenz. SevenZMethodConfigurationTest	0.002089295
org.apache.commons.compress.compressors.xz.XZCompressorOutputStream	0.002086146
org.apache.commons.compress.archivers.dump.DumpArchiveUtil	0.002079913
org.apache.commons.compress.compressors.lzma. LZMACompressorOutputStream	0.002078299
org.apache.commons.compress.compressors.deflate. DeflateCompressorInputStream	0.002077119
org.apache.commons.compress.archivers.sevenz. BoundedSeekableByteChannelInputStream	0.002074737
org.apache.commons.compress.compressors. CompressorStreamFactoryRoundtripTest	0.002074636
org.apache.commons.compress.compressors.lz4. BlockLZ4CompressorInputStreamTest	0.002074136
org.apache.commons.compress.compressors.BZip2TestCase	0.002074049
org.apache.commons.compress.ArchiveUtilsTest	0.002073654
org.apache.commons.compress.archivers.dump. DumpArchiveInputStreamTest	0.002072029
org.apache.commons.compress.archivers.JarTestCase	0.002072029
org.apache.commons.compress.archivers.zip.UnparseableExtraFieldData	0.002070789
org.apache.commons.compress.utils.CountingInputStream	0.002069465
org.apache.commons.compress.archivers.sevenz.AES256SHA256Decoder	0.002068919
org.apache.commons.compress.archivers.jar.JarArchiveInputStream	0.002066671
org.apache.commons.compress.archivers.sevenz.SevenZNativeHeapTest	0.002066019
org.apache.commons.compress.compressors.LZMATestCase	0.002064466
org.apache.commons.compress.compressors.deflate. DeflateCompressorOutputStream	0.002063795
org.apache.commons.compress.utils.ServiceLoaderIterator	0.00206107
org.apache.commons.compress.archivers.zip.ScatterSampleTest	0.00205919
org.apache.commons.compress.archivers.zip.GeneralPurposeBitTest	0.00205799
org.apache.commons.compress.utils.CountingOutputStream	0.002057625
org.apache.commons.compress.changes.Change	0.002055329
org.apache.commons.compress.compressors.bzip2.CRC	0.002054163

Table A-4: Apache Commons Compression Complete Results (cont.)

Class	Complexity
org.apache.commons.compress.archivers.TestArchiveStreamProvider	0.002053476
org.apache.commons.compress.compressors. TestCompressorStreamProvider	0.002053476
org.apache.commons.compress.archivers.zip. UnsupportedZipFeatureException	0.00205299
org.apache.commons.compress.archivers.zip.ZipExtraField	0.002052501
org.apache.commons.compress.compressors.xz.XZUtilsTestCase	0.002052501
org.apache.commons.compress.compressors.lzma.LZMAUtilsTestCase	0.002052501
org.apache.commons.compress.compressors.snappy.PureJavaCrc32C	0.002052382
org.apache.commons.compress.archivers.zip.ZipEightByteIntegerTest	0.002049142
org.apache.commons.compress.archivers.zip.ZipLongTest	0.002049142
org.apache.commons.compress.archivers.zip.ZipShortTest	0.002049142
org.apache.commons.compress.compressors.XZTestCase	0.002048971
org.apache.commons.compress.archivers.zip.ZipEncodingTest	0.002047536
org.apache.commons.compress.archivers.zip.NioZipEncoding	0.00204666
org.apache.commons.compress.utils.CharsetsTest	0.002046262
org.apache.commons.compress.archivers.jar.JarArchiveEntry	0.002045791
org.apache.commons.compress.changes.ChangeSetResults	0.002045417
org.apache.commons.compress.archivers.dump.DumpArchiveConstants	0.002045146
org.apache.commons.compress.compressors.pack200.Pack200UtilsTest	0.002043381
org.apache.commons.compress.utils.BoundedInputStream	0.002041561
org.apache.commons.compress.archivers.ExceptionMessageTest	0.002040596
org.apache.commons.compress.archivers.zip.AsiExtraFieldTest	0.00203956
org.apache.commons.compress.archivers.zip.FallbackZipEncoding	0.002038902
org.apache.commons.compress.parallel.FileBasedScatterGatherBackingStore	0.002038902
org.apache.commons.compress.archivers.zip.Maven221MultiVolumeTest	0.002038694
org.apache.commons.compress.archivers.memory. MemoryArchiveInputStream	0.00203658
org.apache.commons.compress.compressors.pack200.StreamBridge	0.002035194
org.apache.commons.compress.archivers.cpio.CpioUtil	0.002034169
org.apache.commons.compress.archivers.sevenz.Archive	0.002032652
org.apache.commons.compress.archivers.zip.ScatterSample	0.002031917
org.apache.commons.compress.archivers.zip.BitStreamTest	0.002031126
org.apache.commons.compress.archivers.zip.EncryptedArchiveTest	0.002030317
org.apache.commons.compress.compressors.pack200.Pack200Utils	0.002029796
org.apache.commons.compress.archivers.zip.CircularBuffer	0.002029709
org.apache.commons.compress.archivers.Lister	0.002028444
org.apache.commons.compress.archivers.zip.ZipArchiveEntryRequest	0.002027982
org.apache.commons.compress.archivers.dump.Dirent	0.002027757
org.apache.commons.compress.ChainingTestCase	0.002027702
org.apache.commons.compress.archivers.jar.JarArchiveOutputStreamTest	0.002022343
org.apache.commons.compress.archivers.ar.ArArchiveInputStreamTest	0.00202156

Table A-4: Apache Commons Compression Complete Results (cont.)

Class	Complexity
org.apache.commons.compress.archivers.zip. X0016_CertificateIdForCentralDirectory	0.002020823
org.apache.commons.compress.archivers.zip.X0015_CertificateIdForFile	0.002020823
org.apache.commons.compress.archivers.arj.ArjArchiveInputStreamTest	0.002019733
org.apache.commons.compress.archivers.tar.SparseFilesTest	0.002018888
org.apache.commons.compress.archivers.memory.MemoryArchiveEntry	0.002018849
org.apache.commons.compress.archivers.jar.JarArchiveOutputStream	0.002018419
org.apache.commons.compress.MockEvilInputStream	0.002016637
org.apache.commons.compress.archivers.cpio.CpioArchiveOutputStreamTest	0.002015645
org.apache.commons.compress.archivers.ar.ArArchiveOutputStreamTest	0.002014147
org.apache.commons.compress.archivers.sevenz.SevenZMethodConfiguration	0.002012766
org.apache.commons.compress.archivers.zip.ZipMethod	0.002012766
org.apache.commons.compress.archivers.zip.UnicodePathExtraField	0.002012746
org.apache.commons.compress.archivers.zip.UnicodeCommentExtraField	0.002012746
org.apache.commons.compress.archivers.cpio.CpioArchiveInputStreamTest	0.00201232
org.apache.commons.compress.archivers.zip.CircularBufferTest	0.002010854
org.apache.commons.compress.archivers.sevenz.SevenZMethod	0.002008801
org.apache.commons.compress.compressors.gzip.GzipUtils	0.002007689
org.apache.commons.compress.compressors.bzip2.BZip2Utils	0.002007689
org.apache.commons.compress.compressors.deflate.DeflateParameters	0.002007457
org.apache.commons.compress.archivers.cpio.CpioUtilTest	0.002007256
org.apache.commons.compress.compressors.CompressorStreamProvider	0.002007256
org.apache.commons.compress.archivers.ArchiveStreamProvider	0.002007256
org.apache.commons.compress.compressors.deflate. DeflateCompressorInputStreamTest	0.002007256
org.apache.commons.compress.utils.CloseShieldFilterInputStream	0.002004387
org.apache.commons.compress.archivers.EntryStreamOffsets	0.002004387
org.apache.commons.compress.compressors.bzip2. BZip2CompressorInputStreamTest	0.002001228
org.apache.commons.compress.archivers.zip.BitStream	0.002001017
org.apache.commons.compress.compressors.CompressorServiceLoaderTest	0.001999288
org.apache.commons.compress.archivers.ArchiveServiceLoaderTest	0.001999288
org.apache.commons.compress.utils.CountingStreamTest	0.001999288
org.apache.commons.compress.compressors.pack200.Pack200Strategy	0.00199921
org.apache.commons.compress.archivers.ArchiveEntry	0.001995006
org.apache.commons.compress.compressors.deflate.DeflateParametersTest	0.00199372
org.apache.commons.compress.archivers.dump.DumpArchiveUtilTest	0.00199372
org.apache.commons.compress.archivers.sevenz.SevenZArchiveEntryTest	0.00199372
org.apache.commons.compress.archivers.zip.ScatterStatistics	0.00199288
org.apache.commons.compress.archivers.zip.StreamCompressorTest	0.001989857
org.apache.commons.compress.compressors.BZip2UtilsTestCase	0.001987481

Table A-4: Apache Commons Compression Complete Results (cont.)

Class	Complexity
org.apache.commons.compress.compressors.GzipUtilsTestCase	0.001987481
org.apache.commons.compress.archivers.zip.ZipEncoding	0.001987481
org.apache.commons.compress.parallel.ScatterGatherBackingStore	0.001987481
org.apache.commons.compress.utils.Lists	0.001987481
org.apache.commons.compress.archivers.dump.InvalidFormatException	0.001986746
org.apache.commons.compress.compressors.pack200. TempFileCachingStreamBridge	0.001986725
org.apache.commons.compress.archivers.zip.BinaryTreeTest	0.001984982
org.apache.commons.compress.archivers.dump.DumpArchiveEntryTest	0.001984367
org.apache.commons.compress.archivers.tar.TarArchiveSparseEntry	0.001984014
org.apache.commons.compress.compressors.snappy.FramedSnappyDialect	0.001980381
org.apache.commons.compress.utils.FlushShieldFilterOutputStream	0.001980235
org.apache.commons.compress.utils.Iterators	0.001980068
org.apache.commons.compress.compressors.pack200. InMemoryCachingStreamBridge	0.0019789
org.apache.commons.compress.archivers.dump. UnsupportedCompressionAlgorithmException	0.001978128
org.apache.commons.compress.archivers.zip.Zip64RequiredException	0.001978128
org.apache.commons.compress.utils.CRC32VerifyingInputStream	0.001978128
org.apache.commons.compress.utils.Sets	0.001978128
org.apache.commons.compress.archivers.ArchiveException	0.001978064
org.apache.commons.compress.archivers.StreamingNotSupportedException	0.001977775
org.apache.commons.compress.utils.Charsets	0.001976352
org.apache.commons.compress.archivers.memory. MemoryArchiveTestCase	0.001975551
org.apache.commons.compress.archivers.zip.X000A_NTFSTest	0.001975551
org.apache.commons.compress.compressors.CompressorException	0.001972117
org.apache.commons.compress.compressors.deflate. DeflateCompressorOutputStreamTest	0.001970061
org.apache.commons.compress.archivers.zip. X0019_EncryptionRecipientCertificateList	0.001970061
org.apache.commons.compress.compressors.xz. XZCompressorInputStreamTest	0.001970061
org.apache.commons.compress.archivers.sevenz.StreamMap	0.001970061
org.apache.commons.compress.archivers.zip.X0014_X509Certificates	0.001970061
org.apache.commons.compress.archivers.dump. UnrecognizedFormatException	0.001963822
org.apache.commons.compress.compressors.bzip2.Rand	0.001963822
org.apache.commons.compress.archivers.dump.ShortFileException	0.001963822
org.apache.commons.compress.archivers.arj.MainHeader	0.001963822
org.apache.commons.compress.parallel. ScatterGatherBackingStoreSupplier	0.001963822
org.apache.commons.compress.archivers.zip. ZipArchiveEntryRequestSupplier	0.001963822
org.apache.commons.compress.archivers.arj.LocalFileHeader	0.001963822
org.apache.commons.compress.archivers.zip.ZipArchiveEntryPredicate	0.001963822
org.apache.commons.compress.parallel.InputStreamSupplier	0.001957811

Table A-4: Apache Commons Compression Complete Results (cont.)

Class	Complexity
org.apache.commons.compress.archivers.zip.ZipConstants	0.001957811
org.apache.commons.compress.archivers.sevenz.BindPair	0.001957811
org.apache.commons.compress.PasswordRequiredException	0.001957811
org.apache.commons.compress.archivers.sevenz.Coder	0.001947147
org.apache.commons.compress.archivers.sevenz.SubStreamsInfo	0.001947147
org.apache.commons.compress.archivers.sevenz.NID	0.001941136
org.apache.commons.compress.archivers.zip.Zip64Mode	0.001941136
org.apache.commons.compress.archivers.tar.TarConstants	0.001941136
org.apache.commons.compress.archivers.zip.UnixStat	0.001941136
org.apache.commons.compress.archivers.cpio.CpioConstants	0.001941136
org.apache.commons.compress.utils.CharsetNames	0.001941136
org.apache.commons.compress.archivers.sevenz.StartHeader	0.001941136
org.apache.commons.compress.compressors.bzip2.BZip2Constants	0.001941136