EXPLORATORY STUDY ON THE EFFECTIVENESS OF TYPE-LEVEL COMPLEXITY METRICS

by

Killian Smith

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

Spring 2018

ACKNOWLEDGEMENTS

Thoughout my time at Montana State University I have been greatly assited by professors, staff, and fellow students, but no one has helped me more than my advisor Dr. Clemente Izurieta. Without his guidance during my time as an undergraduate student, I may not have continued my education to graduate school. He has helped me understand what it means to be a researcher and a scientist in a field where these terms can, at times, loose rigor and meaning.

Funding Acknowledgment

The work presented in this thesis was made possible by the funding provided by Montana State University through graduate teaching assistant positions. Without this source of funding, the completion of this document would have been unlikely.

TABLE OF CONTENTS

1. INTRODUCTION 1

2. BACKGROUND 4

Background 4
Linear Model Metrics 6
Tree Model Metrics 6
Graph Model Metrics 9
Motivation 10

3. RESEARCH QUESTIONS 13

RQ0 14
RQ1 15
RQ2 17
RQ3 19
RQ4 21

4. EXPERIMENTAL DESIGN 23

Domain 23
Methods 25
Artifacts 28
ANT 28
Commons-Lang 28
Commons-CLI 29
Wicket 29
Commons-Compression 29

5. RESULTS 31

6. DISCUSSION 41

Research Question 0 41
Research Question 1 42
Research Question 2 42
Research Question 3 43
Research Question 4 43

TABLE OF CONTENTS – CONTINUED

7. THREATS TO VALIDITY 45

Internal Validity 45
External Validity 45
Construct Validity 46
Conclusion Validity 46

8. CONCLUSION 47

Future Research 48

APPENDIX: Artifact Profiles 52

APPENDIX: Algebraic Types 63

LIST OF TABLES

Table Page

1.1 Lehman's Laws of Software Evolution (1980) 2
2.1 Weyuker's Metric Properties 5
2.2 Logic and Program Correspondence 11
2.3 Type Level Metrics 12
3.1 Research Question Summary 13
3.2 Research Question 0 (Part A) 14
3.3 Research Question 0 (Part B) 15
3.4 Research Question 1 16
3.5 Research Question 2 18
3.6 Research Question 3 20
3.7 Research Question 4 22
4.1 Criterion 24
4.2 Metrics Collected for Java Projects 27
5.1 Metric Correlation to Faults Spearman Test Results $n = 60$ 39
5.2 Delta Metric Correlation to Faults Spearman Test Results $n = 55$ 40
B.1 Java and Algebraic Types 64
B.2 Java and Algebraic Types (cont) 65

LIST OF FIGURES

Figure Page

5.1 Metrics Trends (Part A) 32
5.2 Metrics Trends (Part B) 33
5.3 Classic Metric Results 34
5.4 Coupling Metric Results 35
5.5 Coupling Metric Results (cont) 36
5.6 Type Level Metric Results 37
5.7 Type Level Metric Results (cont) 38
A.1 ANT Profile 53
A.2 ANT Delta Profile 54
A.3 Commons-Lang Profile 55
A.4 Commons-Lang Delta Profile 56
A.5 Commons-CLI Profile 57
A.6 Commons-CLI Delta Profile 58
A.7 Wicket Profile 59
A.8 Wicket Delta Profile 60
A.9 Commons-Compress Profile 61
A.10 Commons-Compress Delta Profile 62

ABSTRACT

The research presented in this thesis analyzes the feasibility of using information collected at the type level of object oriented software systems as a metric for software complexity, using the number of recorded faults as the response variable. In other words, we ask the question: Do popular industrial language type systems encode enough of the model logic to provide useful information about software quality? A longitudinal case study was performed on five open source Java projects of varying sizes and domains to obtain empirical evidence supporting the proposed type level metrics. It is shown that the type level metrics *Unique Morphisms* and *Logic per Line of Code* are more strongly correlated to the number of reported faults than the popular metrics *Cyclomatic Complexity* and *Instability*, and performed comparably to *Afferent Coupling*, *Control per Line of Code*, and *Depth of Inheritance Tree*. However, the type level metrics did not perform as well as *Efferent Coupling*. In addition to looking at metrics at single points in time, successive changes in metrics between software versions was analyzed. There was insufficient evidence to suggest that the metrics reviewed in this case study provided predictive capabilities in regards to the number of faults in the system. This work is an exploratory study; reducing the threats to external validity requires further research on a wider variety of domains and languages.

NOMENCLATURE

LOC	Lines of Code
CYC	Cyclomatic Complexity
EFF	Efferent Coupling
AFF	Afferent Coupling
RMI	Instability
DIT	Depth Inheritance Tree
T	Types
M	Morphisms
T/M	Types per Morphism
L/LOC	Logic per Line of Code
C/LOC	Control per Line of Code

INTRODUCTION

Software systems are rapidly becoming integrated into everyday life in the modern world. Infrastructure, medicine, and communication have all become intertwined with cyber and cyber-physical systems. This inter-connectedness can provide enormous benefits, but at a cost: as software systems become increasingly prevalent in the management of real-world systems, their complexity also increases, leaving room for faulty programs or security flaws that can be exploited by a malicious agent. This complexity can come from a myriad of sources, including support for legacy features, adding new features to a system that may be at odds with the original design, or developer errors.

The realization that software complexity is harmful is not a new development. In his 1980 paper [1], Lehman analyzes the life cycles of various software systems and the effects of management when developing software. In this paper, Lehman observes that 70% of software's expenditure is due to maintenance, and that implementing parts of a software system *before* the complete, or even local, model of the system is verified can lead to an increase in the number of faults in the system. Furthermore, the more complex the system becomes, and the more it deviates from the initial model from superimposed modifications, the system becomes harder to change. From his findings, Lehman proposed five laws of software engineering (Table 1.1). Lehman repeated this work in his 1997 paper [2] and obtained similar results that support these laws of software evolution.

In both of these papers, Lehman uses *number of modules* as the sole software metric to indicate complexity. This is an obvious limitation when attempting to

Table 1.1: Lehman's Laws of Software Evolution (1980)

1	**Continuing Change** A program that is used and that as an implementation of its specification reflects some other reality, undergoes continual change or becomes progressively less useful. The change or decay process continues until it is judged more cost effective to replace the system with a recreated version.
2	**Increasing Complexity** As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.
3	**Fundamental Law of Software Evolution** Program evolution is subject to a dynamics which makes the programming process, and hence measures of global project and system attributes, self-regulating with statistically determinable trends and invariances.
4	**Invariant Work Rate** During the active life of a program the global activity rate in a programming project is statistically invariant.
5	**Perceived Complexity** During the active life of a program the release content (changes, additions, deletions) of the successive releases of an evolving program is statistically invariant.

determine the complexity of software, or identify potential aspects for improvement. Software complexity is not something that can be measured directly; however, there are certain aspects of a program, language, or model, that do indicate complexity. Developers are primarily concerned with producing software that is both correct to a specification and abstract enough to easily modify and reason about the consequent results of full or partial evaluation. When a piece of software becomes too complex for a developer to correctly understand the underlying semantics, there is a much higher likelihood that faults will be introduced. To this end, in this thesis we refer to *metrics* as program properties that the developer has control over, and will use faults as the response variable that determines the *fitness* of a given metric as a measure of overall software complexity.

BACKGROUND

Background

Software metrics have a long and varied history, ranging from the simple *Lines of Code* (LOC) of software written in machine code, to complex sets of metrics [3] for object oriented systems. The effectiveness, and in some cases validity, of metrics often comes into debate. Metrics such as Halstead's programming effort measure [4], have terms that are difficult (or sometimes impossible) to compute or are entirely implementation dependent. Others, like lines of code, have very little meaning when they are compared across projects or programming languages. Weyuker [4] proposed a set of properties to more formally address the validity of various metrics (Table 2.1).

The properties merited by Weyuker provide a good generalization of what should be expected, but should be taken with a grain of salt; in the presence of certain runtime environments, some of the warranted properties are non-existent. In a stateless environment, properties 6, 7, and 9 will not hold as a result of referential transparency, and in a total language property 4 will not hold as equivalent programs can be reduced to the same normal form. Failure to meet these properties in the specified environments should not be viewed as a detriment.

Metrics to measure software complexity roughly fall into three categories based on the level of abstraction of the model that they operate on. The simplest level of abstraction is isomorphic to a list, and treats a program as a 1-dimensional linear sequence. This linear sequence could be comprised of lines of code, modules, or any other atomic variable. Increasing the level of abstraction to 2-dimensions, the next level of metrics operate on tree representations of a program. These trees could be an abstract syntax tree (AST) of the program or, dependency trees. The final level of abstraction lifts the program to 3-dimensions, representing the program as a graph.

Table 2.1: Weyuker's Metric Properties

1	There are unique values of complexity for different programs : $\exists P, Q \,.\, c(P) \neq c(Q)$
2	The set $\exists k \geq 0 \,.\, \{x \mid c(P) == k\}$ is finite.
3	There are distinct programs that have the same complexity $\exists P, Q \,.\, c(P) = c(Q)$
4	Equivalent programs can have different complexities: $\exists P, Q \,.\, P = Q \wedge c(P) \neq c(Q)$
5	Metric addition is strictly increasing: $\forall P, Q \,.\, c(P) \leq c(PQ) \wedge c(Q) \leq c(PQ)$
6	Catenation of programs may not yield uniform behavior of complexity: $\exists P, Q, R \,.\, c(P) = c(Q) \wedge c(PR) \neq c(QR)$ or $\exists P, Q, R \,.\, c(P) = c(Q) \wedge c(RP) \neq c(RQ)$
7	Permutation of statements does not preserve complexity: $\exists P \,.\, c(P) \neq c(permute(P))$
8	Complexity is preserved across alpha equivalence: $\exists P, Q \,.\, P =_\alpha Q \implies c(P) = c(Q)$
9	Complexity can be introduced with catenation: $\exists P, Q \,.\, c(P) + c(Q) < c(PQ)$
10	Complexity is monotonically increasing: $\forall P, Q \,.\, c(P) + c(Q) \leq c(PQ)$

The most common family of metrics at this level operate on control flow graphs (CFG).

Linear Model Metrics

The simplest way to model a program is as a vector of characters or binary digits. This view of software contains very little information about the structure or functionality of the program, but due to this simplicity they are often easy to compute. Metrics falling in this category are typically on the *absolute* scale, and can be used to normalize other software metrics allowing metrics from differing sized programs to be compared.

- **Lines of Code**: The lines of code metric, or LOC, counts the number of non-empty lines in a program that are not comment lines. While extremely useful for characterizing complexity in low-level languages like assembly, it's effectiveness decreases for higher level languages as the amount of information per line of code increases.

- **Entropy Measure**: The amount of randomness, or entropy, of software can also be an indicator for the complexity of individual pieces of a system. Entropy measures can determine regions of code that are uncharacteristically complex, and target them for further analysis. This technique has seen widespread use in reverse engineering and malware analysis [5], [6], [7].

Tree Model Metrics

Programming languages are defined formally by grammars that describe how to build valid programs. Character streams are parsed into an abstract syntax tree (AST) based on this grammar [8]. Metrics operating on this model of a program contain information about the structure and function of the program, but requires the program to be parsed into an AST before any metrics can be calculated. More formally, metrics at this level capture the *static* properties of a program, and typically fall on the *ratio* scale. These metrics provide insight into the statics of a program, and can be used in static analysis and optimization.

- **Attribute Counts**: The simplest measurement that can be made on an AST is the raw count of attributes, in this case annotated tree nodes, that the language contains. Possible attributes of interest include statements, expressions, keywords, or rules used to generate the AST. Calculations of this type of metric are *catamorphic* procedures (meaning they are generalized folds), and take $O(n)$ time to compute after parsing has been completed, where n is the number of nodes in the AST.

- **Dominance**: Another useful measurement on an AST is *dominance*, or number of dependencies, of non-leaf nodes in the tree. Deeper ASTs will contain nodes with a large number of dependencies, and changes to these nodes can result in a "ripple effect" of changes on the nodes that it dominates. Ideally, AST dependancies should be kept shallow to minimize the effect of this "ripple" of changes. In their 1999 paper [9], Burd et al. discuss the viability of using two software metrics based upon function call dominance trees on a case study of GNUs C compiler, gcc. The paper provides a distinction between dependencies within a submodule and between submodules, referring to them as *direct dominance* and *strong dominance*, respectively. The first metric, $m1$, is defined as the ratio between *strong dominance nodes* to *direct dominance nodes*. The second metric, $m2$, is defined as the percentage of nodes that are unique to a dominance tree. Using data collected from modification logs each of the versions of gcc tested, it was found that decreases in $m1$ and $m2$ caused significantly more time to be spent on corrective maintenance; which can be interpreted as meaning these versions of the software were less maintainable.

- **Coupling**: The concept of *coupling* is similar to that of dominance, but abstracted to the level of modules. Coupling refers to the "interconnectedness" of between modules, including relations that a module is dependent on (called *efferent* coupling or EFF) and the relations that a module is a dependency for (called *afferent* coupling or AFF). The overall coupling between objects (called CBO) includes all non-inheritance related coupling. An additional measurement, *instability* or RMI, is defined as $RMI = \frac{EFF}{EFF+AFF}$ and is useful for describing how resilient to change a module is. Just as with dominance relations, changes to a module can "ripple" to other modules that are directly or indirectly dependent on that module. Software systems should be designed accordingly; by increasing the stability of modules that are not likely to be changed, and pushing instability to more dynamic modules. The usage of many of these metrics for object oriented software was analyzed by Martin in his 1994 paper [10]. These coupling metrics are often used in object oriented metric suites [11], [12], [13].

- **Depth of Inheritance Tree**: This OO specific metric, related to both dominance and coupling, is defined as the maximal height of the inheritance class hierarchy. This metric, along with CBO, is one of the metrics popularized by the CK suite of metrics [12]. Like both Dominance and Coupling, the notion behind this metric is to prevent excessive inter-dependencies that can cause "ripples" of changes upon a single modification.

Graph Model Metrics

When executed as machine code, programs act as finite state machines. Given an initial state, an operating system will step though instructions as specified by the program, adjusting the system state for each step. However, for any given instruction step there can be many possible paths to take. A path could be chosen by the current system state, user input, or future computation. Paths are not necessarily unique either; multiple paths can lead back to a previous instruction location (in the case of a loop), and multiple paths can lead to the same instruction location. We can express *all* possible computation paths as a concrete instance of a directed graph, where nodes are unique program instructions and directed edges are an execution step. Formally, for a given program P, we call this representation the *control flow graph* (or CFG) of program P. Metrics at this level capture the *dynamics* of a program, and typically fall on a ratio scale.

- **Variable Definitions and Usage**: According to Moseley et al. [14], a majority of complexity introduced into a software system comes from the presence of state. Variables can be instantiated, mutated, and vary throughout the programs execution. Coupled with the fact that variable behavior will often vary based on paths through the CFG, it is easy to see why this claim was made. In [15], numerous software testing techniques are described for *covering* state in CFGs for use in testing. Taken in another context, this coverage is akin to a measurement of the complexity of state in a system. This state is described by the relations of unique variable definitions and usages, or better known as *def-use pairs*.

- **Unique Paths**: An intuitive measure for program complexity is to count the number of possible execution paths. If a developer was to *fully* test an application, they would have to write a text cases that covered *every* execution path through the graph. Every combination of branches and loops through the CFG would need to be address, which is exponential relative to the size of the graph. Clearly, more paths through the CFG would mean more complexity for the developer to worry about.

- **Cyclomatic Complexity**: A more abstract variant of the unique paths metric is Cyclomatic Complexity (or CYC), analyzed by McCabe in 1976 [16], [17]. Cyclomatic complexity is defined as $v(G) = e - n + p$, where n is the number of nodes in the CFG, e is the number of edges, and p is the number of connected components (typically 1). For those familiar with topology, this is equivalent of measuring the *first Betti number* of a program [18], [19], and captures the *shape* of the programs dynamic properties. In layman terms, it is the number of 1 dimensional "holes" in the graph. This metric is *significantly* easier to compute than the unique paths metric, but still captures aspects of the complexity of the paths by giving a count of all unique cycles within the graph.

Motivation

As an alternative to the classic style of metric that rely on concrete software implementations, we would like to have metrics that indicate the complexity of the software's solution to the given problem space. Ideally, before a software system is implemented, the problem is examined and a model of the requirements is created. Whether the model is a detailed UML diagram or a simple list of required features, the specifications defined describe a contract that the software needs to follow. These specifications are encoded into a languages type system when developers begin implementing the model, and the program itself becomes a *constructive proof* in the problem space. This observation is better known as the

Curry-Howard Correspondence [20], [21], or *programs as proofs*.

Table 2.2: Logic and Program Correspondence

Logic	Programming
Universal Quantification	Dependent Product (Π-Type)
Existential Quantification	Dependent Sum (Σ-Type)
Implication	Function Type
Conjunction	Product Type
Disjunction	Sum Type
True	Unit Type
False	Bottom Type

To this end, we propose a series of software metrics that capture the complexity at this abstracted level. These metrics are shown in Table 2.3.

Unfortunately, not all programming languages have type systems powerful enough to encode *all* specification logic. These shortcomings in the type system mean that some properties of the specification cannot be checked, and the developer must rely on their own diligence to adhere to the formal system design. These properties that are left unchecked by the compiler can lead to program faults. This leads us to the case study presented by this thesis: Do popular industrial language type systems encode enough of the model logic to provide useful information about software quality?

Table 2.3: Type Level Metrics

Metric		Description
Types	T	The number of unique types that a model defines. This includes *all* types: Simple, Sum, Product, and Inductive.
Morphisms	M	The number of unique morphisms, or function mappings, that a model defines.
Types / Morphisms	T/M	The ratio between the number of unique types and unique morphisms defined by a model.
Logic / Control	L/C	The ratio of lines related to model logic to the lines that define program control. In other words, this is the ratio of types + morphisms to the number of lines that define control operations such as assignment, variable mutation, or branching.
Logic / LOC	L/LOC	The ratio of lines related to model logic to the total number of non-comment lines in the software. In other words, the percentage of the project that encodes software specifications checkable by the compiler.
Control / LOC	C/LOC	The ratio of lines related to program control to the total number of non-comment lines in the software. In other words, the percentage of the project that possibly encodes specifications that are non-checkable by the compiler.

RESEARCH QUESTIONS

This thesis aims to better understand and quantify complexity at the type-level of a software system, and attempts to improve software quality in regards to the total number of reported faults. As a baseline, popular classical software complexity metrics will also be tested against; namely, LOC, CYC, AFF, EFF, RMI, and DIT. The type-level metrics that will be analyzed are T, M, L/C, L/LOC, and C/LOC.

In addition to raw metrics, we also analyze the effect that changes in metric values between successive versions of a software system have on faults. This analysis will provide a basis to whether changes in the specified metrics can predict the number of reported faults that will occur in a software system following the application of changes to the code base.

Table 3.1: Research Question Summary

0	Do the software metrics in this case study behave similarly over time, regardless of project?
1	Are there correlations between classic metrics and the number of recorded faults found in a given software system?
2	Are there correlations between type-level metrics and the number of recorded faults found in a given software system?
3	Are there correlations between the rates of change of classic metrics and the number of recorded faults found in a given software system?
4	Are there correlations between the rates of change of type level metrics and the number of recorded faults found in a given software system?

Specifically, we answer the following research questions:

RQ0

Do the software metrics in this case study behave similarly over time, regardless of project? Before running more interesting statistical tests, we must first observe the metrics over time to determine if the metrics follow similar trends across projects. If Lehmans Laws of Software Evolution hold, we should expect monotonic regressions (whether linear or curved) to adequately model the metrics over the software's lifetime. Furthermore, we expect the same relationship for the same metric across projects; i.e.- given that CYC is monotonically increasing for project A, we expect CYC for project B to be monotonically increasing as well.

Table 3.2: Research Question 0 (Part A)

Intuition	:	Assuming that Lehmans Laws of Software evolution hold for all software systems, we expect there to be monotonic changes in type level metrics over a systems lifetime.
$H_{0A,0}$	=	There are no monotonic relations in the metrics over the course of the software's lifetime. In other words, running an F-Test of the r^2 values for both linear and quadratic models results in p-values $p > \alpha = 0.05$.
$H_{0A,A1}$	=	There are metrics that can be modeled using monotonic regressions. More concretely, running an F-Test of the r^2 values for either linear or quadratic models result in a p-value $p \leq \alpha = 0.05$.

Table 3.3: Research Question 0 (Part B)

Intuition	:	We expect the same software metric to exhibit similar monotonic behavior over time, regardless of the software project.
$H_{0B,0}$	=	There *are no* metrics that exhibit the same monotonic behavior across projects.
$H_{0B,A1}$	=	There *are* metrics that exhibit the same monotonic behavior across projects.

RQ1

Are there correlations between classic metrics and the number of recorded faults found in a given software system? Answering this question will provide a baseline to compare the type level metrics with, as well as provide insight into the effectiveness of popular complexity metrics in regards to software quality and probability of faults in the system.

Table 3.4: Research Question 1

Intuition	:	These metrics are historically popular among software developers and in academic/commercial metric suites. As such, we should expect them to correlate highly with the number of faults.
$H_{1,0}$	=	None of the metrics are correlated with the number of recorded faults. More concretely, $\lvert\rho_{LOC}\rvert < \theta$, $\lvert\rho_{CYC}\rvert < \theta$, $\lvert\rho_{AFF}\rvert < \theta$, $\lvert\rho_{EFF}\rvert < \theta$, $\lvert\rho_{RMI}\rvert < \theta$, $\lvert\rho_{DIT}\rvert < \theta$,where θ is the statistically significant threshold of correlation for the sample size [22].
$H_{1,A1}$	=	LOC is correlated with the number of recorded faults; i.e., $\lvert\rho_{LOC}\rvert \geq \theta$.
$H_{1,A2}$	=	CYC is correlated with the number of recorded faults; i.e., $\lvert\rho_{CYC}\rvert \geq \theta$.
$H_{1,A3}$	=	AFF is correlated with the number of recorded faults; i.e., $\lvert\rho_{AFF}\rvert \geq \theta$.
$H_{1,A4}$	=	EFF is correlated with the number of recorded faults; i.e., $\lvert\rho_{EFF}\rvert \geq \theta$.
$H_{1,A5}$	=	RMI is correlated with the number of recorded faults; i.e., $\lvert\rho_{RMI}\rvert \geq \theta$.
$H_{1,A6}$	=	DIT is correlated with the number of recorded faults; i.e., $\lvert\rho_{DIT}\rvert \geq \theta$.

RQ2

Are there correlations between type-level metrics and the number of recorded faults found in a given software system? Metrics at the type level operate on the source program at a vastly more abstract level. If correlations are found to exist, these type-level metrics can be used as a cheaper alternative to classic metrics as the control portions of the program do not need to be parsed or go through static analysis. In addition, this family of metric is better suited to compare software across development teams or even across programing languages.

Table 3.5: Research Question 2

Intuition	:	Since the type level of a language encodes all of the compiler checkable model specifications, we expect there to be a strong negative correlation between these metrics and the number of faults. With more of the model being encoded into the type system, and therefore compiler checkable, there should be fewer program faults than in a system with less of the model encoded in the type system [22].
$H_{2,0}$	=	None of the metrics are correlated with the number of recorded faults. More concretely, $\lvert\rho_T\rvert < \theta$, $\lvert\rho_M\rvert < \theta$, $\lvert\rho_{T/M}\rvert < \theta$, $\lvert\rho_{L/C}\rvert < \theta$, $\lvert\rho_{L/Loc}\rvert < \theta$, $\lvert\rho_{C/LOC}\rvert < \theta$, where θ is the statistically significant threshold of correlation for the sample size.
$H_{2,A1}$	=	Type counts are correlated with the number of recorded faults; i.e., $\lvert\rho_T\rvert \geq \theta$.
$H_{2,A2}$	=	Morphism counts are correlated with the number of recorded faults; i.e., $\lvert\rho_M\rvert \geq \theta$.
$H_{2,A3}$	=	The ratio of Types to Morphisms is correlated with the number of recorded faults; i.e., $\lvert\rho_{T/M}\rvert \geq \theta$.
$H_{2,A4}$	=	The ratio of Logic to Control is correlated with the number of recorded faults; i.e., $\lvert\rho_{L/C}\rvert \geq \theta$.
$H_{2,A5}$	=	The ratio of Logic to LOC is correlated with the number of recorded faults; i.e., $\lvert\rho_{L/Loc}\rvert \geq \theta$.
$H_{2,A6}$	=	The ratio of Control to LOC is correlated with the number of recorded faults; i.e., $\lvert\rho_{C/Loc}\rvert \geq \theta$.

RQ3

Are there correlations between the rates of change of classic metrics and the number of recorded faults found in a given software system? This question aims to determine if is is possible to predict whether or not an update to a software system will produce an increase in the occurrence of reported faults.

Table 3.6: Research Question 3

Intuition	:	When writing software, developers must maintain a mental model of a program. Large changes between software versions means that developers must dramatically alter their mental image in a short period of time, leaving potential for more faults to be introduced. As such, we expect the size of the change between metrics to be directly proportional to the number of faults [22].
$H_{3,0}$	=	None of the Δ metrics are correlated with the number of recorded faults. More concretely, $\lvert\rho_{\Delta LOC}\rvert < \theta$, $\lvert\rho_{\Delta CYC}\rvert < \theta$, $\lvert\rho_{\Delta AFF}\rvert < \theta$, $\lvert\rho_{\Delta EFF}\rvert < \theta$, $\lvert\rho_{\Delta RMI}\rvert < \theta$, $\lvert\rho_{\Delta DIT}\rvert < \theta$, where θ is the statistically significant threshold of correlation for the sample size.
$H_{3,A1}$	=	Delta LOC is correlated with the number of recorded faults; i.e., $\lvert\rho_{\Delta LOC}\rvert \geq \theta$.
$H_{3,A2}$	=	Delta CYC is correlated with the number of recorded faults; i.e., $\lvert\rho_{\Delta CYC}\rvert \geq \theta$.
$H_{3,A3}$	=	Delta AFF is correlated with the number of recorded faults; i.e., $\lvert\rho_{\Delta AFF}\rvert \geq \theta$.
$H_{3,A4}$	=	Delta EFF is correlated with the number of recorded faults; i.e., $\lvert\rho_{\Delta EFF}\rvert \geq \theta$.
$H_{3,A5}$	=	Delta RMI is correlated with the number of recorded faults; i.e., $\lvert\rho_{\Delta RMI}\rvert \geq \theta$.
$H_{3,A6}$	=	Delta DIT is correlated with the number of recorded faults; i.e., $\lvert\rho_{\Delta DIT}\rvert \geq \theta$.

RQ4

Are there correlations between the rates of change of type level metrics and the number of recorded faults found in a given software system? Like RQ3, this question is aimed at predicting fault behavior based of the rate of change of metrics. Similarly to RQ2, if there does exist a correlation we can use the Δ type-level metric to provide useful metrics that are cheaper to calculate than the Δ classic metrics.

Table 3.7: Research Question 4

Intuition	:	By the same rationale as RQ3, we expect the size of the change between metrics to be directly proportional to the number of faults.
$H_{4,0}$	=	None of the Δ metrics are correlated with the number of recorded faults. More concretely, $\lvert\rho_{\Delta T}\rvert < \theta$, $\lvert\rho_{\Delta M}\rvert < \theta$, $\lvert\rho_{\Delta T/M}\rvert < \theta$, $\lvert\rho_{\Delta L/C}\rvert < \theta$, $\lvert\rho_{\Delta L/Loc}\rvert < \theta$, $\lvert\rho_{\Delta C/LOC}\rvert < \theta$, where θ is the statistically significant threshold of correlation for the sample size [22].
$H_{4,A1}$	=	Delta Type counts are correlated with the number of recorded faults; i.e., $\lvert\rho_{\Delta T}\rvert \geq \theta$.
$H_{4,A2}$	=	Delta Morphisim counts are correlated with the number of recorded faults; i.e., $\lvert\rho_{\Delta M}\rvert \geq \theta$.
$H_{4,A3}$	=	The Delta ratio of Types to Morphisims is correlated with the number of recorded faults; i.e., $\lvert\rho_{\Delta T/M}\rvert \geq \theta$.
$H_{4,A4}$	=	The Delta ratio of Logic to Control is correlated with the number of recorded faults; i.e., $\lvert\rho_{\Delta L/C}\rvert \geq \theta$.
$H_{4,A5}$	=	The Delta ratio of Logic to LOC is correlated with the number of recorded faults; i.e., $\lvert\rho_{\Delta L/Loc}\rvert \geq \theta$.
$H_{4,A6}$	=	The Delta ratio of Control to LOC is correlated with the number of recorded faults; i.e., $\lvert\rho_{\Delta C/Loc}\rvert \geq \theta$.

EXPERIMENTAL DESIGN

Domain

Before initializing the case study, there are a number of criterion that need to be considered in the design phase. The two primary requirements for this study are the provision of *open source* projects and that the development logs and/or bug reports for these projects are publicly available. This will allow us to take measurements from the project as well as associate the measurements with a relative view of software quality.

Secondary considerations include the lifetime of a project, the properties of a languages type system and operational calculi, and the number of programming languages present in a project. We would like to check for Lehman's Laws of software evolution in the type level metrics studied, as well as determine if any trends exist to predict software quality in regards to the number of reported faults, which requires many previous versions of a software system to be available. To make the study easier to conduct, as well as less error prone, the target language should be statically typed and have concrete structures for declaring data types and the morphisms between these types. To eliminate any unnecessary complexity, the projects chosen should be written primarily in a single language. For reference, all criterion are shown in Table 4.1.

To satisfy these critereon, Java was chosen as the target language of the study, and a subset of projects available from the *Apache Software Foundation* was used as the domain. The language Java was chosen primarily for two reasons:

1. Java is an extremely popular language, ranking near the top of many popular benchmarks [23], [24], [25], along with C/C++. This popularity should make

Table 4.1: Criterion

1	The software needs to be open-source. Without access to the source code, it would be impossible to perform the required static analysis.
2	The software should be written primarily in a single language. Projects written in multiple languages add unnecessary complexity to our analysis. Additionally, some aspects of the softwares complexity could be hidden by biases within the programming languages themselves (differing levels of abstraction, maturity of libraries, etc).
3	Since our research questions focus on information at the type-level, the software should be written in a strong, statically typed language. Ideally this language would also have a uniform way of describing morphisms between the types.
4	The software needs to have many previous versions available. Without this, we would not be able to observe how the evolution of type-level metrics compares to classic metrics.
5	The project needs to have a log of reported faults available for each of the versions over the softwares lifetime.
6	The language should be at least relatively popular, making it easier to find software projects to test. Additionally, having the study done in a popular language would improve the relevancy of the study to a wider audience.

it much easier to find long-term projects with open source code, and help to satisfy the criterion 1, 4, and 6.

2. Java is statically typed and defines types and morphisms in a uniform way. Types are constructed as *Class Constructors*, with no argument constructors being the *unit type*, multi-argument constructors being *product types*, and multi-constructor classes being *sum types*. All morphisms within Java are defined by *Class Methods*, with an implicit first argument of the same type as that of the *Object* defined by the class. A more formal explanation can be seen in Appendix B. This satisfies criterion 3.

Criterion 1, 2 and 5, can be addressed by carefully choosing projects from the *Apache Software Foundation*. Projects included in this case study were chosen based on the number of previous versions with source code freely available, as well as on the quality of logging provided on each version of the project. High quality logs will track issues, notes, or faults for each version of the software, which we will use as the response variable in this case study.

Methods

After project selection, we began data collection. For each project selected, the following steps were executed:

1. Query the software logs for the project, and record the number of reported faults within each version of the project.

2. Download the source code of each software version that has corresponding fault data.

3. Load each of the collected software versions into the *Eclipse* IDE, and compile the project.

4. Once compiled, we can obtain software metric data from the *Eclipse Metric Plugin*, available through the *Eclipse Marketplace* [1]. The resulting data can be exported as an XML.

5. Query each of the XML metric data files for the relevant fields, listed in Table 4.2. This was automated through the use of a *Python* script, which output the data into a format readable by the statistical language *R* (which will be used in analysis).

6. Calculate the non-primitive type level metrics, T/M, L/C, L/LOC, and C/LOC.

7. Calculate the *Delta* (Δ) *metrics.* These metrics are obtained by subtracting every reading for a given metric at version i from version $i + 1$. In other words, the Δ-metric is the *change* in the metric at any given point in time. Repeat this calculation for every metric in the study.

After data collection, we began preliminary analysis. Before conducting any statistical tests, we first get a sense of the data by creating visual aids of the metrics for each project over time. These descriptive views detail how metrics evolve, and are helpful in addressing research question RQ0.

In addition to the per project views, we generate visuals for each metric *combined across projects*, plotted against the number of reported faults for each metric reading. Before combining, the metrics are normalized by LOC where applicable. For any metrics that exhibit monotonic behavior, we will also check for normality using a *QQ-Plot.* This provides crucial information on which statistical tests may be performed.

[1] Available from : https://marketplace.eclipse.org/content/eclipse-metrics

Table 4.2: Metrics Collected for Java Projects

LOC	The total number of lines of code in the software system.
CYC	The average McCabe cyclomatic score of methods in the project.
EFF	The average efferent coupling of modules.
AFF	The average afferent coupling of modules.
RMI	The average instability of modules.
DIT	The average depth of inheritance tree of modules.
T	The number of types that the system defines. In Java, this is equivalent to the number of classes.
M	The number of morphisms that the system defines. In Java, this is equivalent to the number of methods defined.
C	The number of lines that refer to program control only. In Java, this is equivalent to the number of lines of code contained in method bodies.

To answer research questions RQ1 and RQ2, we execute a correlation test on each of the collected metrics and their respected reported fault counts. The test that should be used is dependent on the results from the QQ-Plots that have already been generated. If the data is normally distributed, we will use a parametric *Pearson Test*; otherwise, a non-parametric *Spearman* test will be used.

The final research questions, RQ3 and RQ4, are addressed using the same techniques as the previous research questions, with the caveat that the *differences* between sequential metric and fault data points, rather that the data points themselves, serve as the datum.

Artifacts

This case study was performed on five separate software projects of varying sizes and domains. Each of the projects is listed below, along with a short description of the software provided by their respective development teams.

ANT

> *Apache Ant is a Java library and command-line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other. The main known usage of Ant is the build of Java applications. Ant supplies a number of built-in tasks allowing to compile, assemble, test and run Java applications. Ant can also be used effectively to build non Java applications, for instance C or C++ applications. More generally, Ant can be used to pilot any type of process which can be described in terms of targets and tasks.* –`http://ant.apache.org`

Commons-Lang

The standard Java libraries fail to provide enough methods for manipulation of its core classes. Apache Commons Lang provides these extra methods. Lang provides a host of helper utilities for the java.lang API, notably String manipulation methods, basic numerical methods, object reflection, concurrency, creation and serialization and System properties. Additionally it contains basic enhancements to java.util.Date and a series of utilities dedicated to help with building methods, such as hashCode, toString and equals. –`https://commons.apache.org/proper/commons-lang`

Commons-CLI

The Apache Commons CLI library provides an API for parsing command line options passed to programs. It's also able to print help messages detailing the options available for a command line tool. –`https://commons.apache.org/proper/commons-cli`

Wicket

Invented in 2004, Wicket is one of the few survivors of the Java serverside web framework wars of the mid 2000's. Wicket is an open source, component oriented, serverside, Java web application framework. With a history of over a decade, it is still going strong and has a solid future ahead. Learn why you should consider Wicket for your next web application. –`https://wicket.apache.org`

Commons-Compression

The Apache Commons Compress library defines an API for working with ar, cpio, Unix dump, tar, zip, gzip, XZ, Pack200, bzip2, 7z, arj, lzma, *snappy, DEFLATE, lz4, Brotli, Zstandard, DEFLATE64 and Z files.* – ***https://commons.apache.org/proper/commons-compress***

RESULTS

The results for this case study were produced in three phases. The first phase, preliminary analysis of metrics and their evolution, can be seen in Appendix A. These charts show the evolution of each metric of interest by individual project. Additionally, charts of the change between successive metric readings were created. In Appendix A, these charts are referred to as *Delta* (Δ) metrics. Statistics on the raw metrics collected on linear and curved fits can be seen in Figure 5.1 and Figure 5.2

The second phase of results was obtained by combining the data from each project, normalizing metrics by LOC where applicable. The delta-metric data from each project produced in the first phase are also combined by metric type. The resulting charts of this condensed data can be seen in Figures 5.3-5.7. This data will be used in statistical tests, where metrics will be tested for correlation with the number of faults in the system.

The third and final phase of case study results is to execute the correlation tests on the metric and delta metric datum processed in the second phase of results. Before choosing the appropriate correlation test, note the following:

1. Both the domain and the codomain of the datum are continuous, and reside on the ratio scale.

2. All visualizations of the metric datum appear to be roughly monotonic.

3. The metric datum fails normality tests, producing highly skewed QQ-plots.

As such, we will choose to use the non-parametric Spearman correlation test for our statistical analysis. The results of the correlation tests for each metric and delta metric can be seen in Table 5.1 and Table 5.2.

(a) Metric LOC Trends

Project	Relation	P-Values	
		Linear	Quadratic
ANT	+	0	8e-08
Wicket	+	0	3e-08
Compress	+	0	0
Lang	+	3e-04	0.0007
CLI	+	0.0485	0.0964

(b) Metric CYC Trends

Project	Relation	P-Values	
		Linear	Quadratic
ANT	-	0.0133	0.0155
Wicket	-	0	0
Compress	-	0	2.4e-07
Lang	-	0.1466	0.1512
CLI	-	0.1688	0.0977

(c) Metric EFF Trends

Project	Relation	P-Values	
		Linear	Quadratic
ANT	+	0.2116	0.2198
Wicket	+	0	0.0015
Compress	+	7e-04	0.0002
Lang	+	0.0889	0.0497
CLI	+	0.1817	0.1181

(d) Metric AFF Trends

Project	Relation	P-Values	
		Linear	Quadratic
ANT	+	0.0214	0.0252
Wicket	+	0	0
Compress	+	0	1e-06
Lang	+	0.0196	0.0140
CLI	NA	NA	NA

Figure 5.1: Metrics Trends (Part A)

(a) Metric RMI Trends

Project	Relation	P-Values	
		Linear	Quadratic
ANT	+	0.2116	0.2198
Wicket	+	0	0
Compress	+	7e-04	0.0002
Lang	+	0.0889	0.0497
CLI	+	0.1817	0.1181

(b) Metric T Trends

Project	Relation	P-Values	
		Linear	Quadratic
ANT	+	0	1.87e-06
Wicket	+	0	2e-08
Compress	+	0	0
Lang	+	4e-04	0.0008
CLI	+	0.0222	0.0679

(c) Metric M Trends

Project	Relation	P-Values	
		Linear	Quadratic
ANT	+	0	2.1e-07
Wicket	+	0	2e-08
Compress	+	0	0
Lang	+	0.0017	0.0030
CLI	+	0.0336	0.0532

(d) Metric C Trends

Project	Relation	P-Values	
		Linear	Quadratic
ANT	+	0	3.2e-07
Wicket	+	0	7e-08
Compress	+	0	0
Lang	+	0.0021	0.0017
CLI	+	0.0559	0.1169

Figure 5.2: Metrics Trends (Part B)

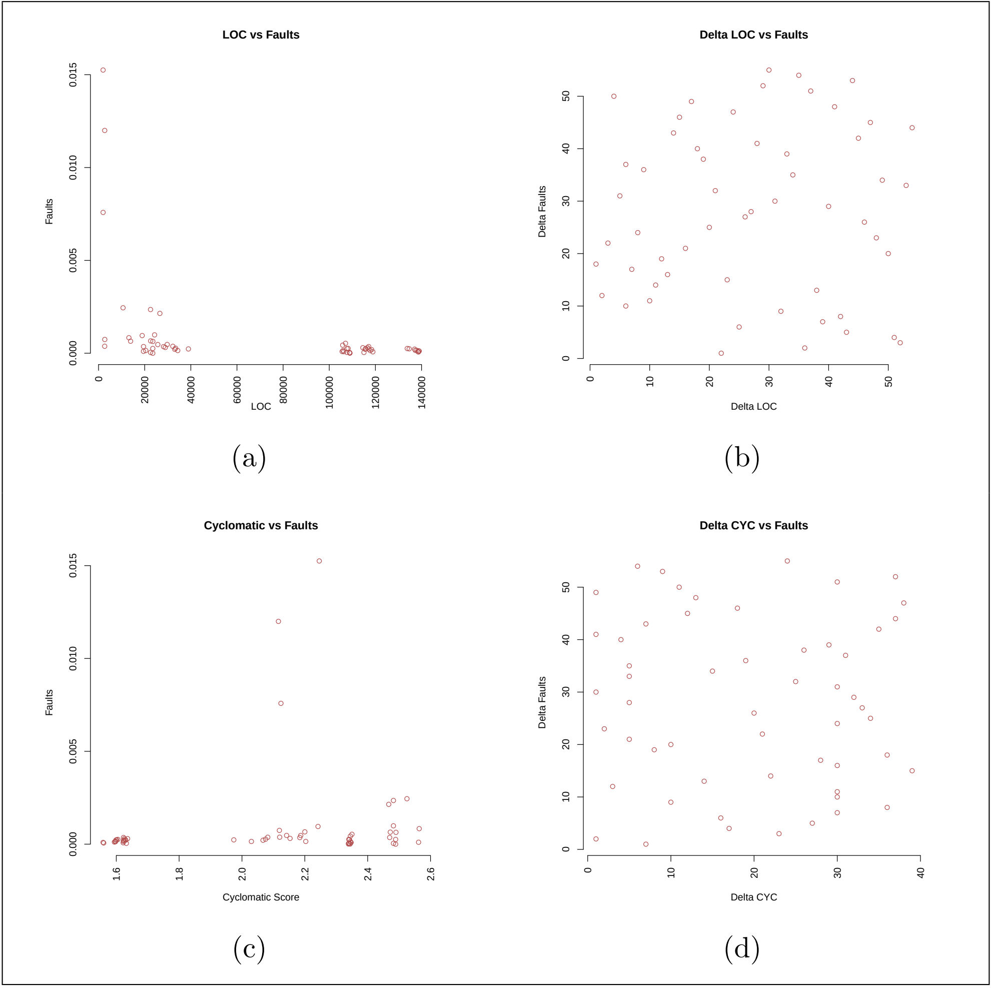


Figure 5.3: Classic Metric Results

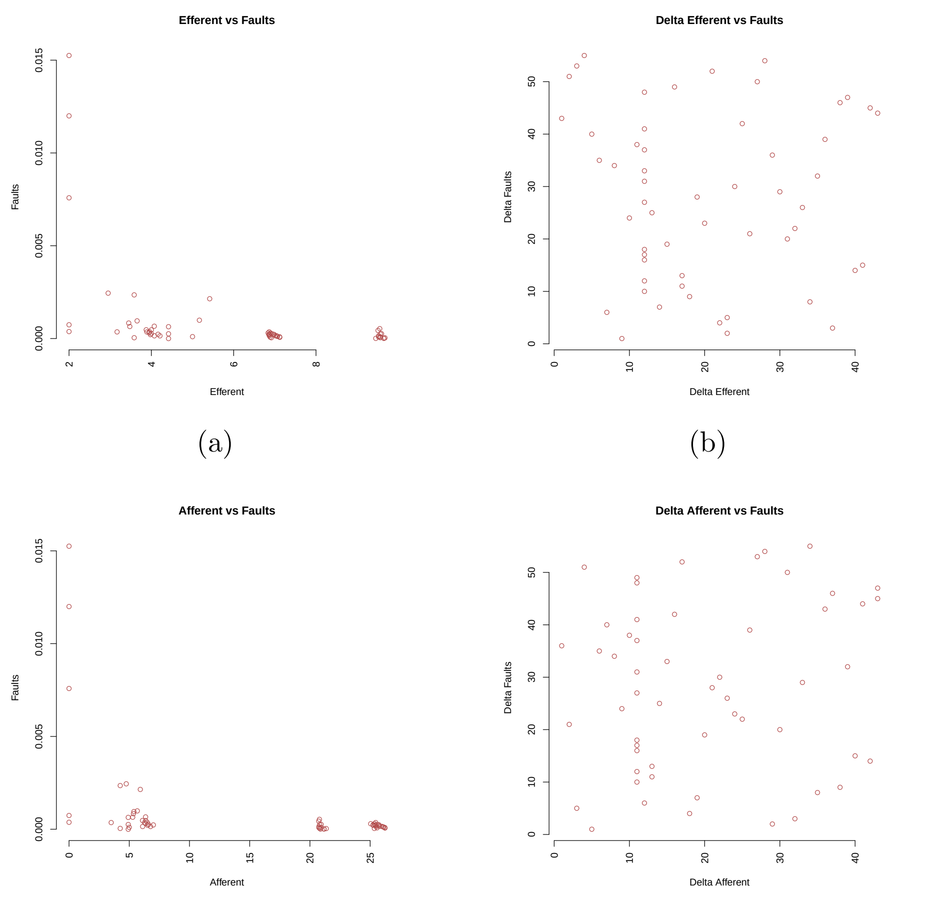


Figure 5.4: Coupling Metric Results

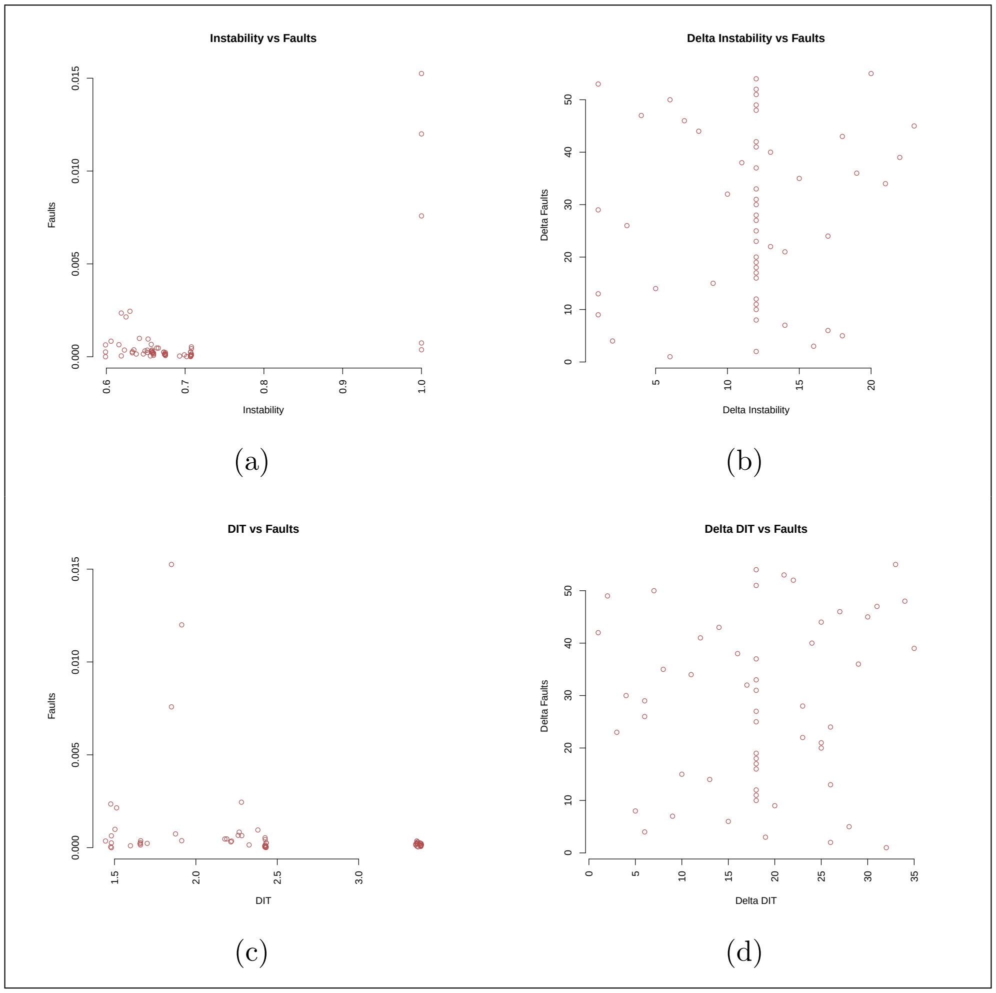


Figure 5.5: Coupling Metric Results (cont)

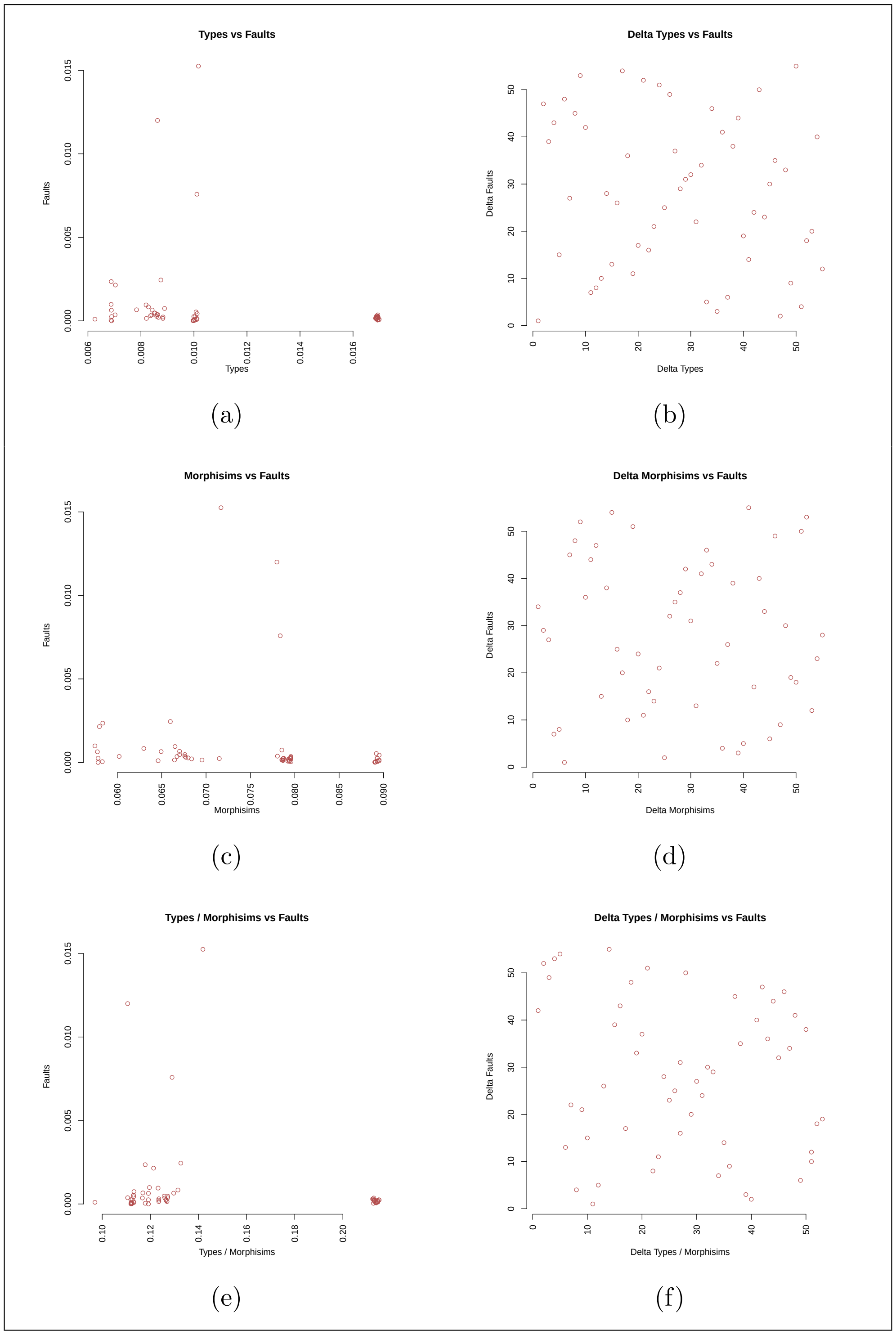
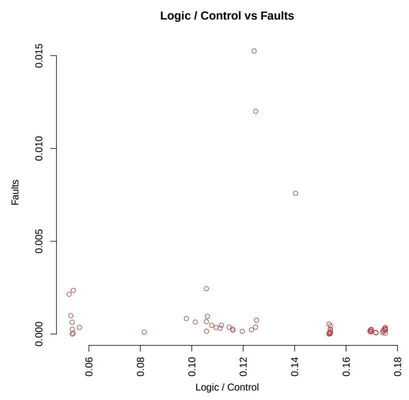
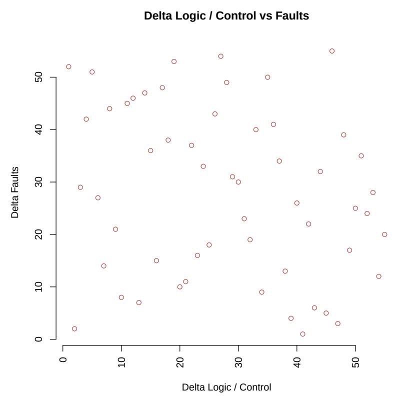


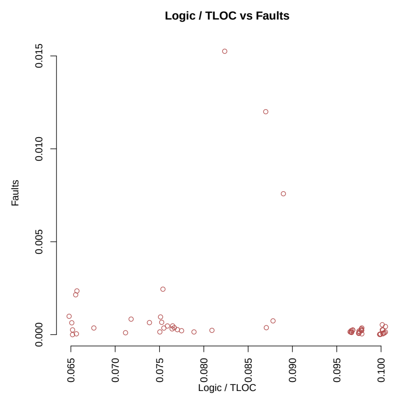
Figure 5.6: Type Level Metric Results



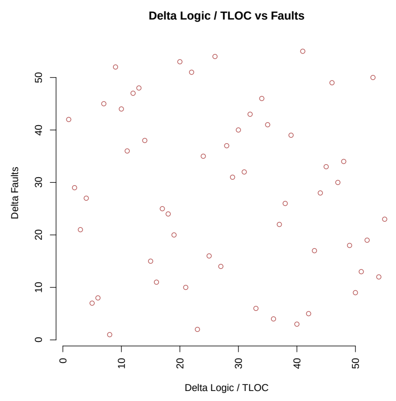
(a)



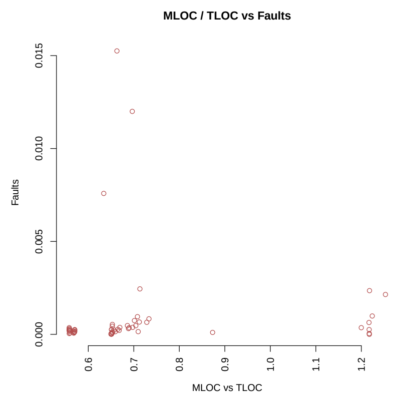
(b)



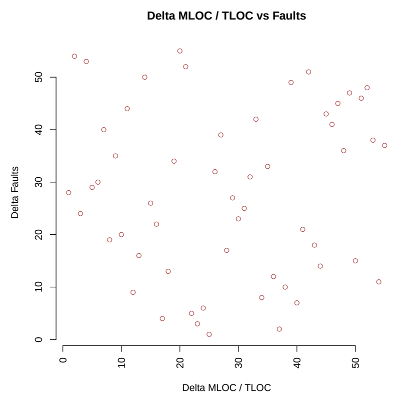
(c)



(d)



(e)



(f)

Figure 5.7: Type Level Metric Results (cont)

Table 5.1: Metric Correlation to Faults
Spearman Test Results $n = 60$

Metric	Results		
	ρ	p-value	correlation
LOC	-0.5886	7.5483e-07	moderate
Cyclomatic	0.2334	0.0727	—
Afferent	-0.5722	1.7883e-06	moderate
Efferent	-0.6571	1.1847e-08	strong
Instability	-0.1575	0.2295	—
DIT	-0.4053	0.0014	moderate
Types	**-0.3272**	**0.0107**	weak
Morphisms	-0.4464	0.0003	moderate
Types/Morphisms	-0.0242	0.8545	—
Logic/Control	-0.3941	0.0018	weak
Logic/LOC	-0.4412	0.0004	moderate
Control/LOC	0.4222	0.0008	moderate

significant at $p < \alpha = 0.05$
significant at Bonferroni corrected $\alpha = 0.05$

Table 5.2: Delta Metric Correlation to Faults
Spearman Test Results $n = 55$

Metric	Results		
	ρ	p-value	correlation
LOC	0.0437	0.7515	—
Cyclomatic	-0.0294	0.8312	—
Afferent	0.0998	0.4687	—
Efferent	-0.0780	0.5715	—
Instability	0.0443	0.7480	—
DIT	0.0880	0.5228	—
Types	-0.1422	0.2994	—
Morphisms	-0.0374	0.7856	—
Types/Morphisms	-0.1544	0.2602	—
Logic/Control	-0.2251	0.0985	—
Logic/LOC	-0.0923	0.5017	—
Control/LOC	0.0280	0.8389	—

significant at $p < \alpha = 0.05$
significant at Bonferroni corrected $\alpha = 0.05$

DISCUSSION

In the following sections, we analyze the results of our case study and their ramifications regarding the outlined research questions.

Research Question 0

In RQ0, we ask two preliminary questions: (a) whether the metrics used in this case study exhibit monotonic patterns over a softwares lifetime, and (b) whether metrics maintain the same monotonic relation across software projects. The results of regression fitting for each metric and project can be seen in Figure 5.1 and Figure 5.2. In addition to the p-values of the F-statistic, these tables also record the direction of the potential monotonic relation; $+$ for strictly increasing, and $-$ for strictly decreasing.

The raw metrics LOC, AFF, T, M, and C, showed strong evidence to reject the null hypotheses $H_{0A,0}$ and $H_{0B,0}$. For all five projects, the linear model for metrics was statistically significant *and* the monotonic relations for each metric remains constant across projects. This provides strong evidence supporting that LOC, AFF, T, M, and C, all follow Lehmans Laws of Software Evolution.

The remaining metrics, CYC, EFF, and RMI, still showed some evidence to reject $H_{0A,0}$ and $H_{0B,0}$, but this evidence was much weaker. Although all metrics maintained their monotonic relations across projects, not all of the projects could be fit to a linear or quadratic model with statistical significance. For CYC and Eff, only 3 out of the five projects could be fit to a model, and for RMI, only 2 of the projects could be fit. From the current evidence, we can neither accept or reject $H_{0A,0}$ and $H_{0B,0}$ for the metrics CYC, EFF, and RMI.

Research Question 1

In RQ1, we ask whether or not classically used metrics (LOC, CYC, EFF, AFF, RMI, or DIT) are correlated with the number of faults found in the software. Metrics that are correlated can be said to be a good indicator of software quality and complexity. The results of the Spearman test on the classic metrics can be seen in Table 5.1.

From this table, we can see that EFF is strongly correlated with the number of faults, and the metrics LOC, AFF, and DIT have a moderate correlation. Interestingly, CYC and RMI do not appear to be correlated to the fault count. More formally, we reject the null hypothesis $H_{1,0}$, and accept the alternative hypotheses $H_{1,A1}$, $H_{1,A3}$, $H_{1,A4}$, and $H_{1,A6}$. In practice, these results indicate that using EFF, AFF, and DIT metrics in Java projects provides a good method of evaluating software quality trends. Alternatively, these results indicate that CYC and RMI *do not* provide a good indication of software quality.

Contrary from what our results indicate, it is not likely a good idea to use LOC as a measure of software quality. More lines of code obviously increases psychological complexity for a developer. Intuitively, the positive correlation between LOC and faults is likely the result of applying software patches to a system. In this case, more lines of code are added to the software solely for the purpose of eliminating a fault(s).

Research Question 2

In RQ2, we ask whether or not type level metrics (T, M, T/M, L/C, L/LOC, C/LOC) are correlated with the number of faults found in the software. Metrics that are correlated can be said to be a good indicator of software quality and complexity. The results of the Spearman test on the classic metrics can be seen in Table 5.1.

From this table, we can see that M, L/LOC, and C/LOC are moderately correlated with the number of faults in the system. Additionally, the metrics T and L/C have a weak correlation, and T/M is not correlated. More concretely, we reject the null hypothesis $H_{2,0}$ and accept the alternative hypotheses $H_{2,A2}$, $H_{2,A5}$, and $H_{2,A6}$. The weakly correlated alternatives, $H_{2,A2}$ and $H_{2,A4}$, are neither rejected or accepted. We can however reject the alternative $H_{2,A3}$, as no significant correlation was found.

From these results, we can say with confidence that there do exist type level metrics that correlate with software quality in regards faults; namely, the number of unique morphisms present in a software system (M), the percentage of the code that describes logic (L/LOC), and the percentage of the code that explicitly handles control (C/LOC).

Research Question 3

In RQ3, we ask whether or not the differences between successive readings of the classically used metrics (Δ-LOC, Δ-CYC, Δ-EFF, Δ-AFF, Δ-RMI, or Δ-DIT) are correlated with the number of faults found in the software. Metrics that are correlated can be said to be a good indicator of software quality and complexity. The results of the Spearman test on these Δ-metrics can be seen in Table 5.2.

From the results of the Spearman test, we can see that there is *no* indication that there exists a correlation between the classic Δ-metrics and the number of faults. This implies that the difference in successive metric readings *does not* predict software quality in regards to number of faults present.

Research Question 4

In RQ4, we ask whether or not the differences between successive readings of the type level metrics (Δ-T, Δ-M, Δ-T/M, Δ-L/C, Δ-L/LOC, or Δ-C/LOC) are correlated with the number of faults found in the software. Metrics that are correlated can be said to be a good indicator of software quality and complexity. The results of the Spearman test on these Δ-metrics can be seen in Table 5.2.

From the results of the Spearman test, we can see that there is *no* correlation between the type level Δ-metrics and the number of faults. This implies that the difference in successive metric readings *does not* predict software quality in regards to number of faults present.

THREATS TO VALIDITY

Any case study or experiment performed will always posses inherent biases. Studies in the field of computer science or software engineering are no exception. In their book [26], Wohlin et al. detail four classes of threats to validity. Following their recommendations, we consider each of these threats and how to mitigate them in our particular case study.

Internal Validity

This form of threat refers to confounding variables in statistical tests, and considers potential factors that we have no control over or were omitted from the study. In our case, the primary internal threat to validity is the assumption that the software metrics that we chose to collect are orthogonal; meaning that the metrics are entirely independent. Because of this design choice, there is the possibility that some interactions between metrics was missed. In addition, we have assumed that all faults are found in each version of software, and that there are no “missed” faults that percolate through the study.

External Validity

This form of threat refers to the ability of an experiment to be generalized, allowing for the conclusion to be applied to *all* instances of a domain rather than just the subset that the experiment was performed over. In our case study, the threat to external validity is the limited number of projects and software versions that were operated over. To mitigate this threat, we chose projects from varying domains. An additional external threat is the fact that a single programming language was used in the case study. Future research on other software projects and programming languages can reduce the external threat to validity.

Construct Validity

This form of threat refers to experimental design choices that may have influence the experiment or case study. In our case study there are two main construct threats: the software that was used to obtain the metrics, and that some of the metrics used were metrics averages across modules in the software. In [27], Lincke et al. found that many software metric tools produce differing results. This is a significant construct threat, as our results are directly dependent on the data collected by these tools. To mitigate this threat, we chose to use a metric tool provided by a well known software company and had many users.

Conclusion Validity

This form of threat refers to the confidence of the conclusion(s) drawn from the experimental results. In the case of our case study, the primary concern is if the correlations observed indicate causality. Again, this threat can be mitigated by reproducing this case study on additional software projects.

CONCLUSION

From our case study, we can make several recommendations to Java developers to reduce the number of faults in software. When working on projects, we advise developers to put priority on monitoring the following software properties: efferent coupling (EFF), afferent coupling (AFF), morphisms (M), percentage of code that contains logic (L/LOC), percentage of code that contains control (C/LOC), and the depth of inheritance tree (DIT). The other metrics analyzed in this case study did not produce adequate results to justify their usage in gauging software faults.

From the results presented for the Δ-metrics, we have found insufficient evidence to suggest that there is any correlation between changes in successive metric readings and the number of faults present in the software.

While never explicitly defined as a research question, one of the most consequential result of the this case study can be seen in the signs of the correlations of the metrics used. The metrics that relate directly to encoded logic of the software model *all* have a negative correlation coefficient. Conversely, the metric that captures the control portion of the software (C/LOC and CYC) have a positive coefficient. In other words, increasing the amount of the model represented in the type level of a software project seems to result in fewer faults in the system. Intuitively, this is a logical statement as any information present at the type level of software can be compiler verified. With more of the specifications occurring at the type level, and therefore being verifiable, more faults will be caught *before* they can occur in the software.

Related are the coupling metrics in object oriented systems. Since the notion of an “object” and a “type” are isomorphic, coupling metrics can be thought of as measuring type level dependencies. Just like the other type level metrics in the study, coupling metrics are negatively correlated.

The case study presented in this thesis provides strong support for software development techniques and languages that emphasize typing and minimizes program control. In addition to making software development recommendations, this thesis should serve as an indication that language research in the field of type theory can be greatly beneficial to developers and improve software quality.

Future Research

The research presented in this thesis provides a foundation for promoting the usage of type level metrics; however, the small sample size is concerning in regards to external validity. Future research replicating this case study would be extremely beneficial in mitigating this threat.

An additional area of future research is the consideration of other object oriented languages. The methods used in this case study should be easily extensible to most statically typed languages that have uniform ways of declaring types and mappings between types.

Lastly, the Δ-metrics that were presented in this case study only considered successive versions. By extending the window of time, looking at many previous versions of the software, additional patterns regarding software quality in terms of faults may be found.

Bibliography

[1] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the ieee*, vol. 68, no. 9, pp. 1060–1076, 1980.

[2] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski, "Metrics and laws of software evolution-the nineties view," in *Software metrics symposium, 1997. proceedings., fourth international*, IEEE, 1997, pp. 20–32.

[3] J. C. Munson and D. S. Werries, "Measuring software evolution," in *Software metrics symposium, 1996., proceedings of the 3rd international*, IEEE, 1996, pp. 41–51.

[4] E. J. Weyuker, "Evaluating software complexity measures," *Ieee transactions on software engineering*, vol. 14, no. 9, pp. 1357–1365, 1988.

[5] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," *Ieee security & privacy*, vol. 5, no. 2, 2007.

[6] CodiSec, *Veles*, 2017. [Online]. Available: `https://github.com/wapiflapi/veles`.

[7] B. M. Institute, *Cantor dust*, 2012. [Online]. Available: `https://sites.google.com/site/xxcantorxdustxx/`.

[8] R. Harper, *Practical foundations for programming languages.* Cambridge University Press, 2016.

[9] E. Burd and M. Munro, "An initial approach towards measuring and characterising software evolution," in *Reverse engineering, 1999. proceedings. sixth working conference on*, IEEE, 1999, pp. 168–174.

[10] R. Martin, "Oo design quality metrics," *An analysis of dependencies*, vol. 12, pp. 151–170, 1994.

[11] A. Nicolaescu, H. Lichter, and Y. Xu, “Evolution of object oriented coupling metrics: A sampling of 25 years of research,” in *Proceedings of the second international workshop on software architecture and metrics*, IEEE Press, 2015, pp. 48–54.

[12] R. Subramanyam and M. S. Krishnan, “Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects,” *Ieee transactions on software engineering*, vol. 29, no. 4, pp. 297–310, 2003.

[13] H. M. Olague, L. H. Etzkorn, S. Gholston, and S. Quattlebaum, “Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes,” *Ieee transactions on software engineering*, vol. 33, no. 6, pp. 402–419, 2007.

[14] B. Moseley and P. Marks, “Out of the tar pit,” *Software practice advancement (spa)*, 2006.

[15] P. Ammann and J. Offutt, *Introduction to software testing.* Cambridge University Press, 2016.

[16] T. J. McCabe, “A complexity measure,” *Ieee transactions on software engineering*, no. 4, pp. 308–320, 1976.

[17] T. J. McCabe and C. W. Butler, “Design complexity measurement and testing,” *Communications of the acm*, vol. 32, no. 12, pp. 1415–1425, 1989.

[18] V. Robins, J. D. Meiss, and E. Bradley, “Computational topology at multiple resolutions: Foundations and applications to fractals and dynamics,” PhD thesis, University of Colorado, 2000.

[19] G. Rote and G. Vegter, “Computational topology: An introduction,” in *Effective computational geometry for curves and surfaces*, Springer, 2006, pp. 277–312.

[20] A. Abel, “Natural deduction and the curry-howard-isomorphism,” 2016.

[21] M. H. Sørensen and P. Urzyczyn, *Lectures on the curry-howard isomorphism*. Elsevier, 2006, vol. 149.

[22] D. Wackerly, W. Mendenhall, and R. Scheaffer, *Mathematical statistics with applications*. Nelson Education, 2007.

[23] P. Carbonnelle, *Pypl popularity of programming language*, 2018. [Online]. Available: `http://pypl.github.io/PYPL.html`.

[24] B. Putano, *Most popular and influential programming languages of 2018*, 2018. [Online]. Available: `https://stackify.com/popular-programming-languages-2018/`.

[25] TIOBE, *Tiobe index for march 2018*, 2018. [Online]. Available: `https://www.tiobe.com/tiobe-index`.

[26] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[27] R. Lincke, J. Lundberg, and W. Löwe, “Comparing software metrics tools,” in *Proceedings of the 2008 international symposium on software testing and analysis*, ACM, 2008, pp. 131–142.

[28] B. C. Pierce, *Types and programming languages*. MIT press, 2002.

APPENDIX: ARTIFACT PROFILES

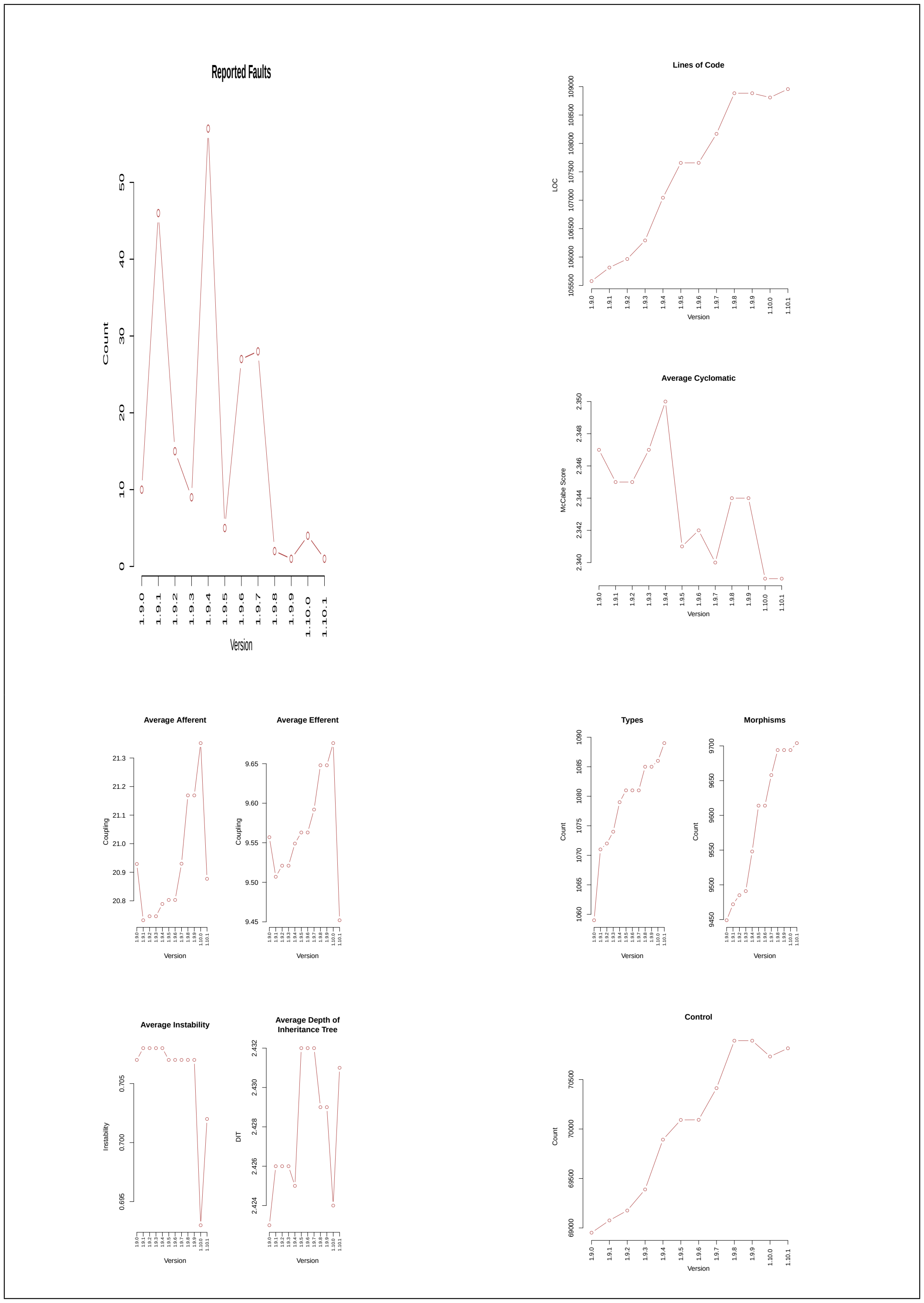


Figure A.1: ANT Profile

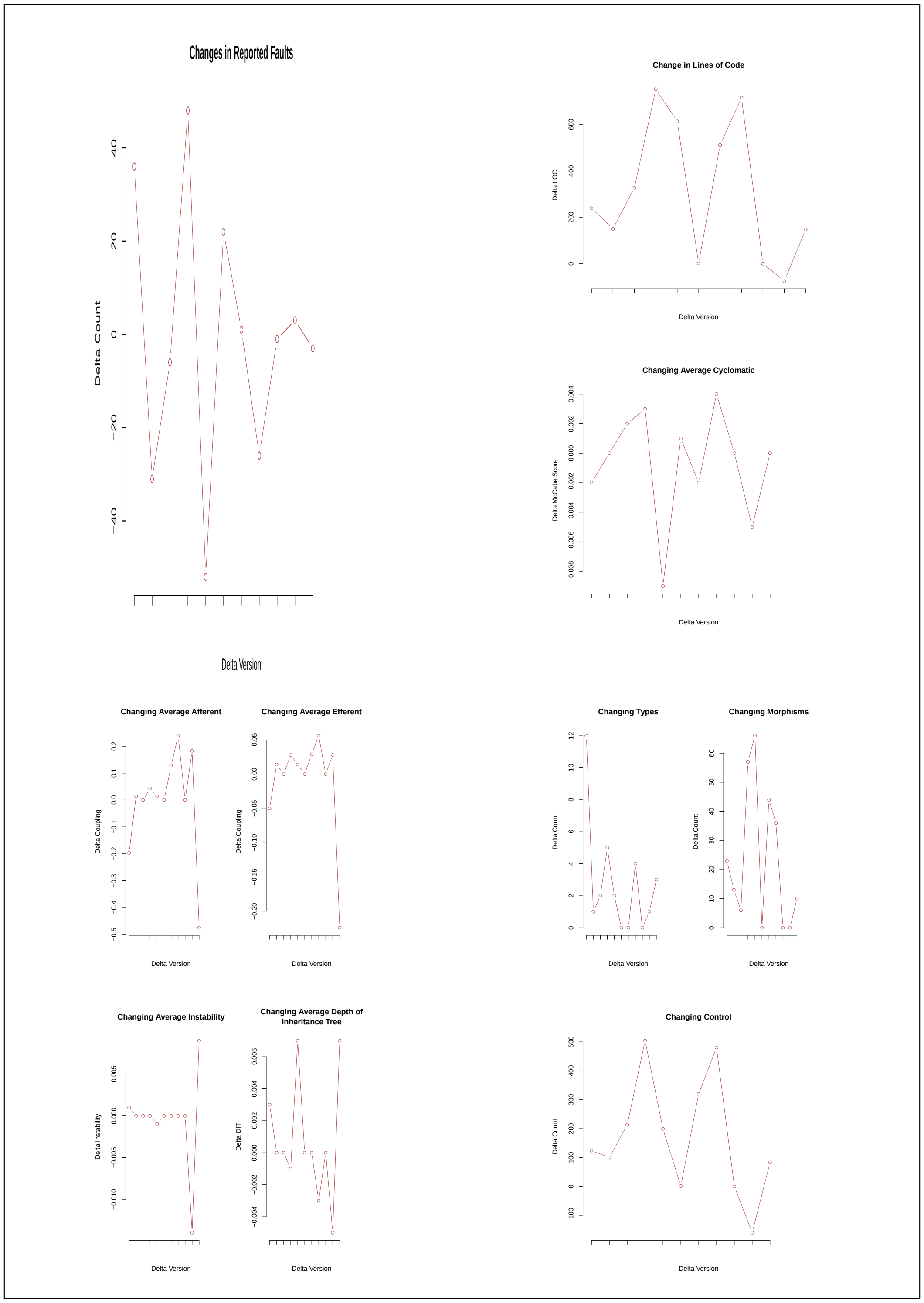


Figure A.2: ANT Delta Profile

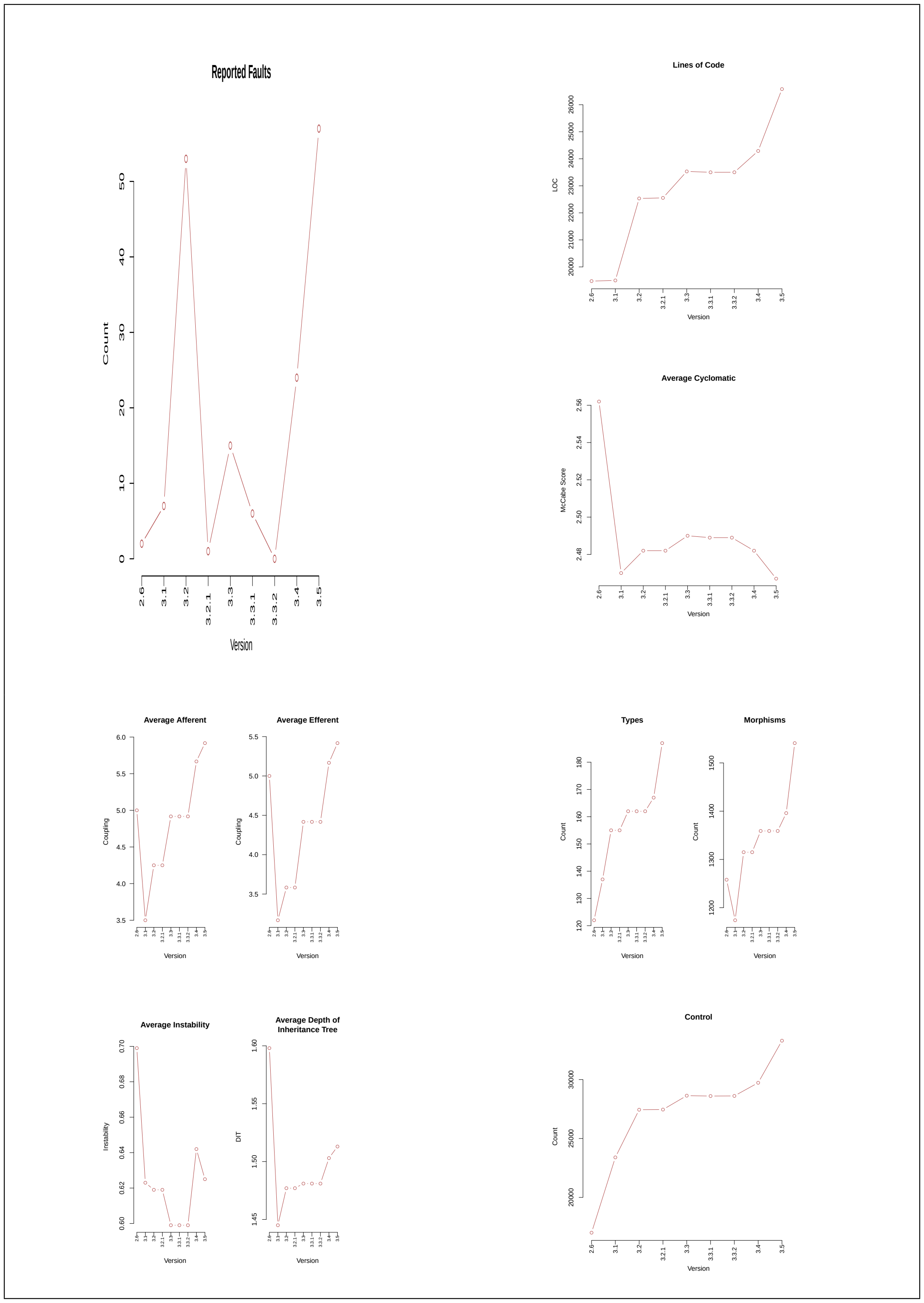


Figure A.3: Commons-Lang Profile

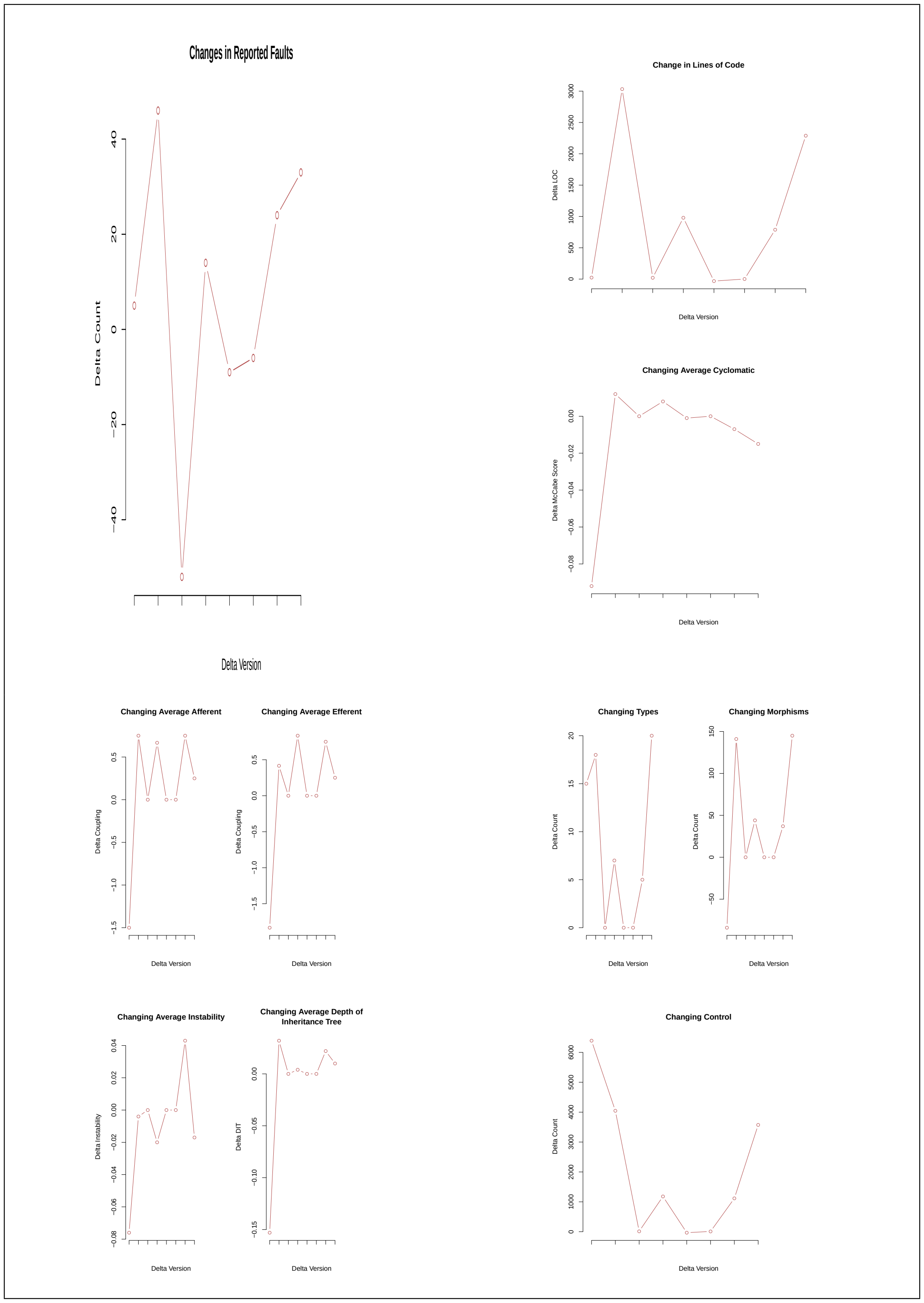


Figure A.4: Commons-Lang Delta Profile

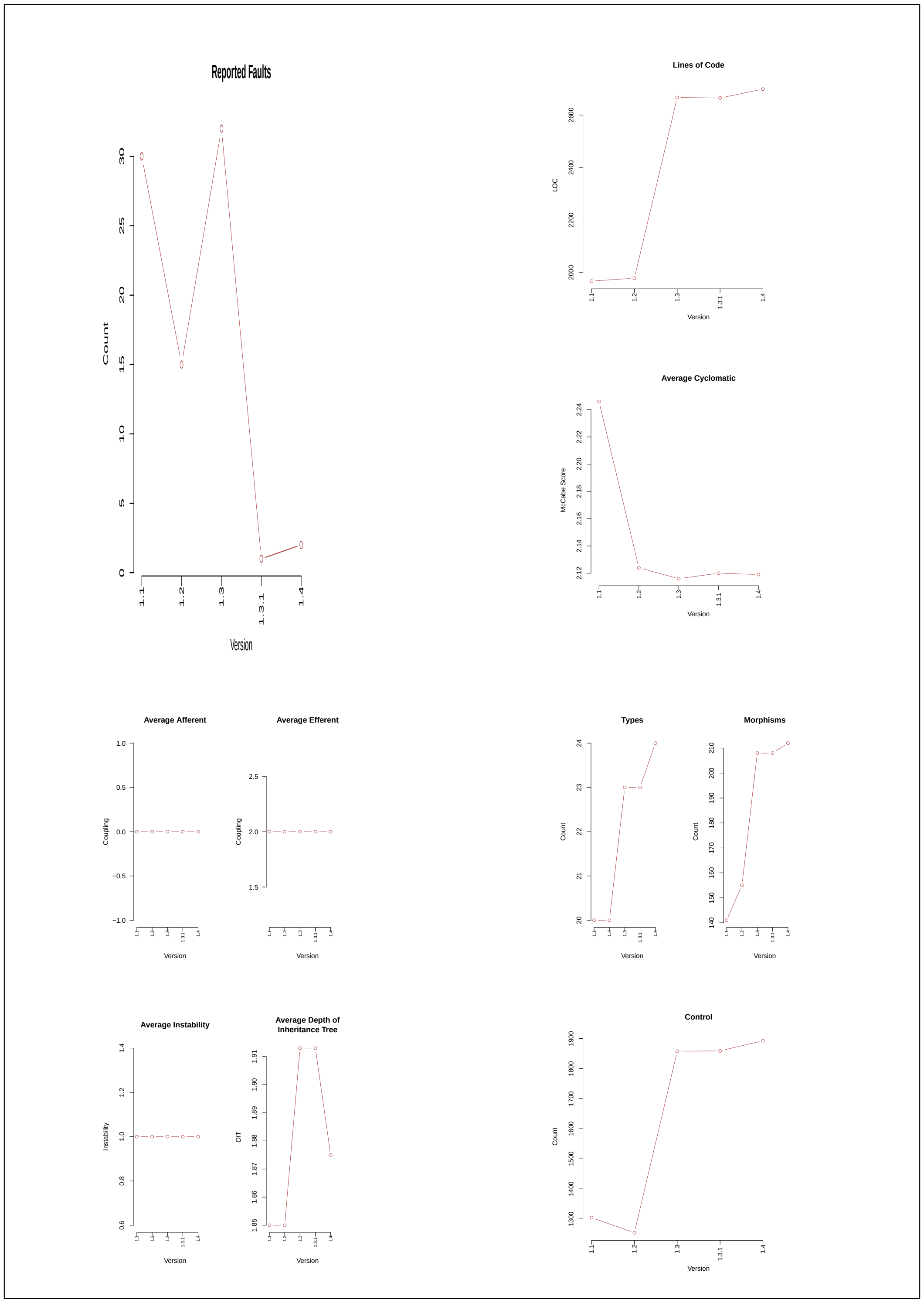


Figure A.5: Commons-CLI Profile

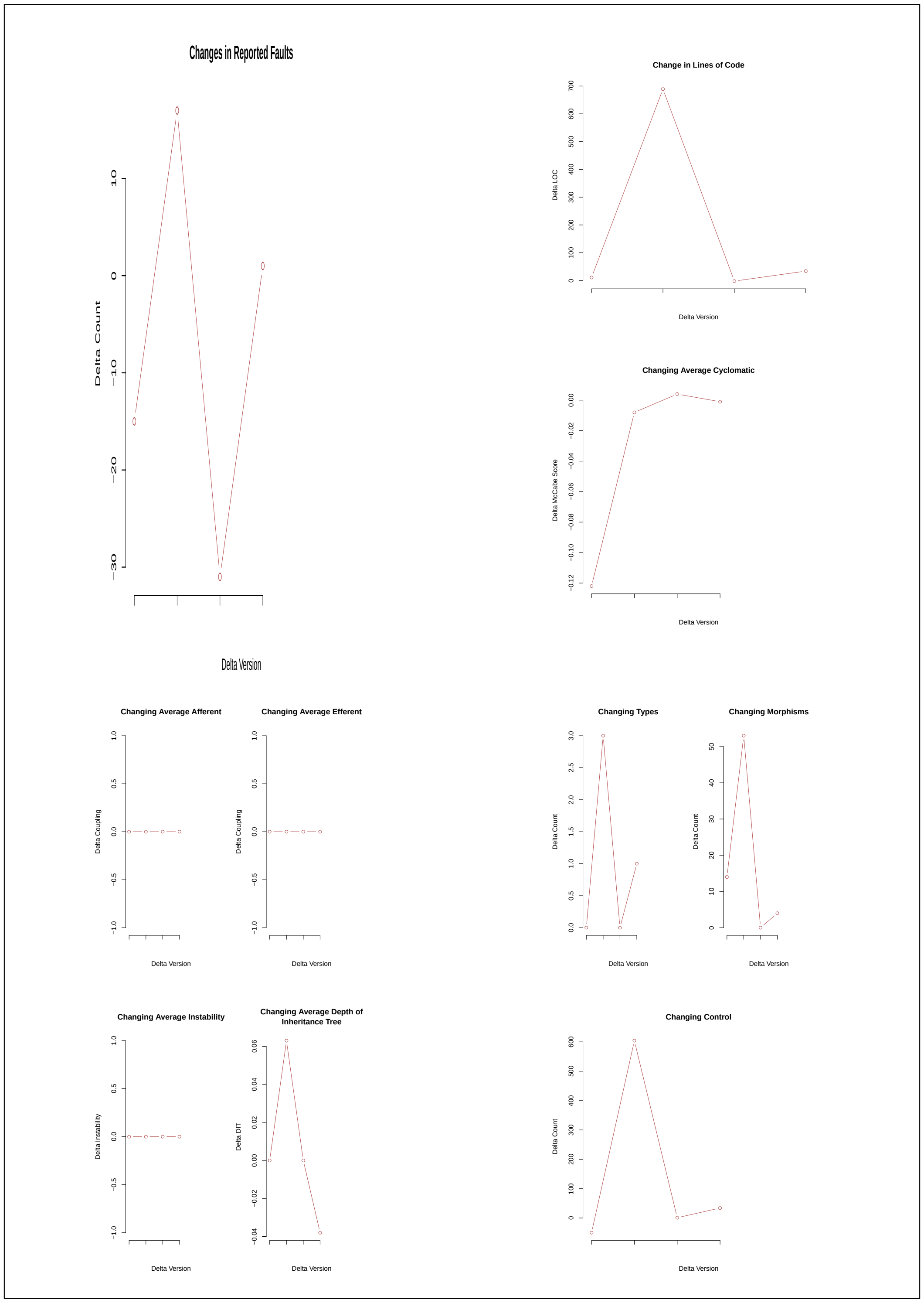


Figure A.6: Commons-CLI Delta Profile

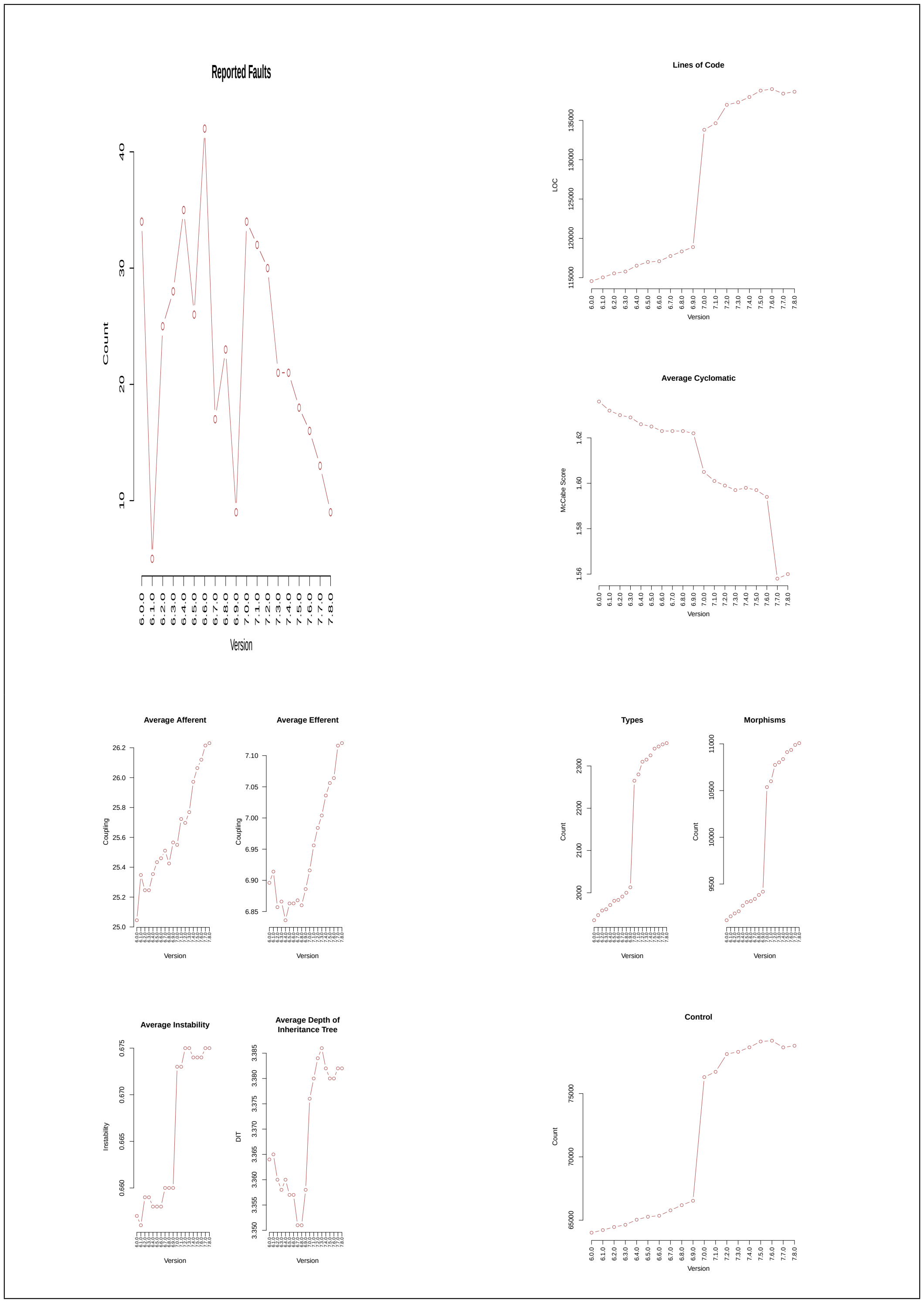


Figure A.7: Wicket Profile

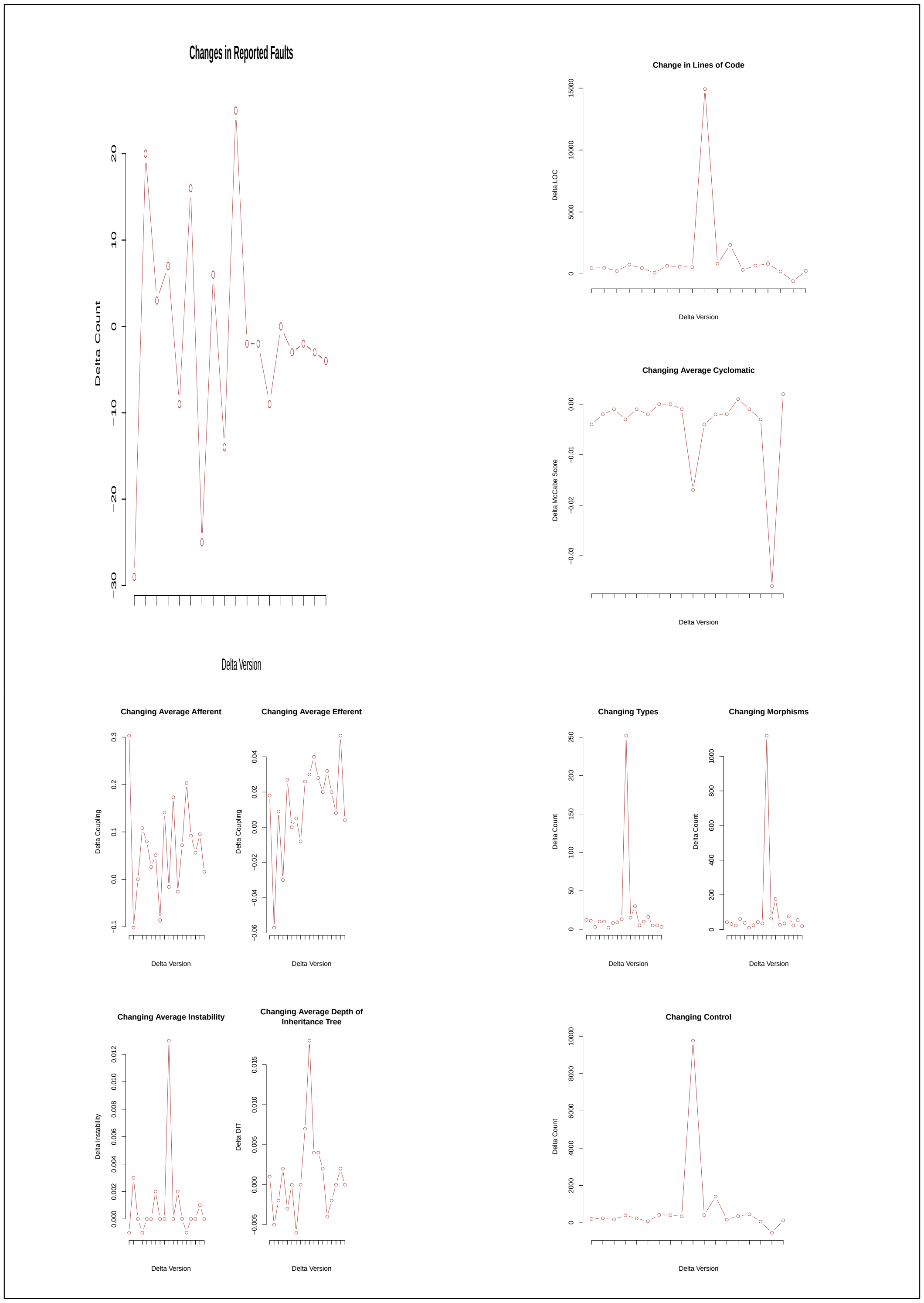


Figure A.8: Wicket Delta Profile

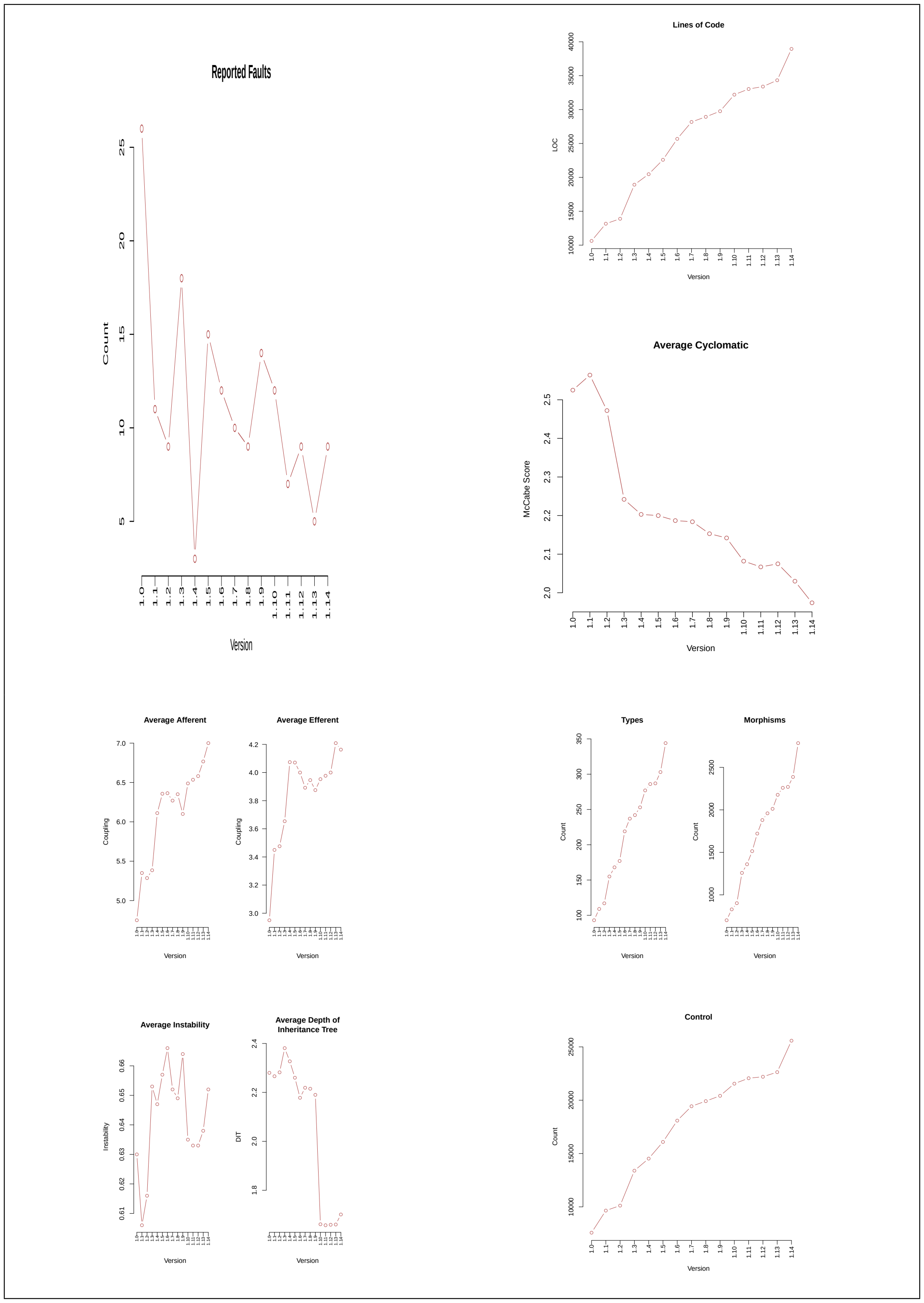


Figure A.9: Commons-Compress Profile

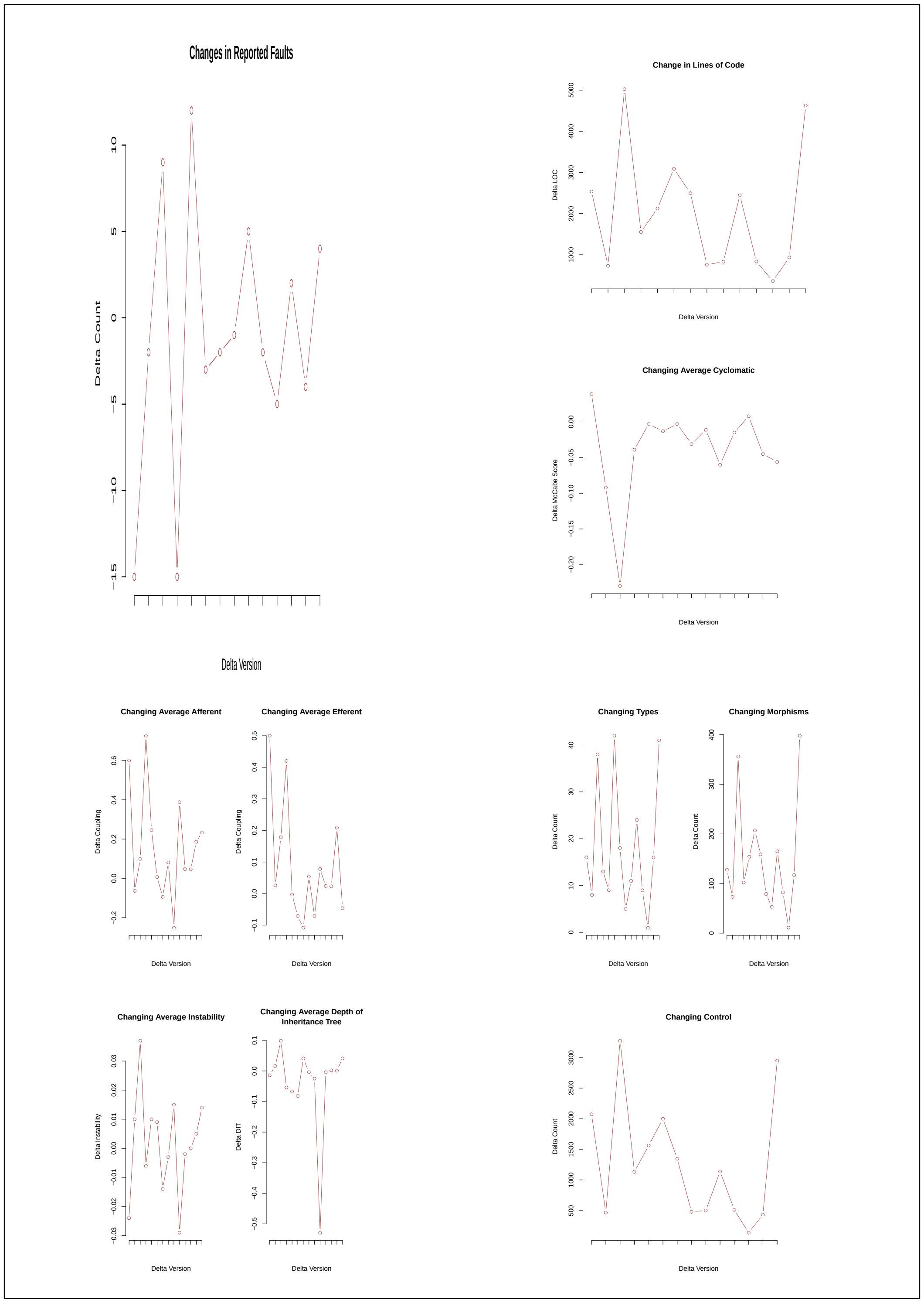


Figure A.10: Commons-Compress Delta Profile

APPENDIX: ALGEBRAIC TYPES

The algebra of data types is a rich and consequential field of study. This thesis only requires a basic understanding of the subject; enough to understand the primitive properties of typing and how they relate to concrete instances of Java code. For more information on this subject, refer to [28] or [8]. Table B.1 shows the relevant type related definitions and how they relate to Java.

Table B.1: Java and Algebraic Types

Void Type	This is a type space with no inhabitants. In Java, this would be a *void* type.
Unit Type	This is a type space with a single inhabitant. This would be a Java constructor that only serves as a label. For example, *True* for a Boolean or *Nil* for a Linked List.
Product Type	This is a type space that inhabits the cross product of types. In Java, these would be constructors that take more than one parameter. For example, consider a constructor that creates a point on a Cartesian plane, taking in an $Int\ x$ and an $Int\ y$. We can clearly see that the constructor does indeed create a type in the domain of $x \times y$. The number of inhabitants of this type space is $x \times y$, and logically these types can be thought of as a *logical and*.
Sum Type	This is a type space that inhabits the sum of types. In Java, sum types are realized by objects that have multiple constructors. For example, consider the sum type *Bool* comprised of the unit types *True* and *False*. The total number of inhabitants is $1 + 1 = 2$. Logically, this type can be thought of as *logical or*.
Polymorphic Type	These type spaces can be thought like variables in traditional algebra. Algebraically, we could write a linked list of type a as $L_a = 1 + a \times L_a$. Java realizes this concept using generics.

Table B.2: Java and Algebraic Types (cont)

Morphism	A morphism describes a mapping between an element in the domain into an element in the codomain (sometimes referred to as an *arrow*). For our purposes, we will also refer to more complex relations, like $f : A \to B \to C$ simply as a morphism. In Java, morphisms are created using *methods*, whose signature S would consist of: *parent object* $\to$ *cross product of arguments* $\to$ *return type*.
Logic Level	In this thesis, we refer to anything that is directly representable in *Logic*, meaning as type signatures, as residing at the logic level. This would include entities like class declarations, constructor signatures, and method signatures.
Control Level	In this thesis, we refer to anything that is directly related to flow control as residing at the control level. This would include constructor bodies and method bodies.