A Multiple Comparative Study of Test-With Development Product Changes and their effects on some attributes of Team Speed and Product Quality

Steve Bannerman

December 27, 2010

Abstract

While studying Test-First Development (TFD), researchers have typically compared it to Test-Last Development (TLD), across groups or projects, and for relatively short durations. But, we have observed that teams testing fine-grained changes do not necessarily develop tests for those changes first. And, comparing across groups or projects is difficult, due to the "contextual variables" specific to each. Finally, many "real-life" projects have long durations.

Because of TFD's potential to increase team speed or product quality, and because of these previous limitations, we were motivated to design and execute our own related study. Our work on this study has resulted in three original contributions to knowledge: (1) our software change model; (2) our measurement method; and (3) our study results.

First, as part of our software model, we defined Test-With Development (TWD) as more general than TFD, and as a step that can be practised to a degree. Then, we defined team speed measures for a developer's product change: initial net size, discards over time, and net size over time. As well, we adopted product quality measures for a developer's product change: change in the number of potential bugs and change in the average method complexity.

Second, we defined a measurement method which allowed us to perform two studies, across six open-source projects and thousands of developer changes. With it, we performed an auxiliary study to calibrate our discards over time measure. Then, we used it to perform our main study to compare the effects of TWD product changes on some attributes of team speed and product quality.

Finally, the results from our main study indicate that there were significant differences between the effects of TWD product changes. We interpret these results to indicate that teams that practice TWD can increase their efficiency, by reducing their product's complexity faster.

Acknowledgements

First, I would like to thank my supervisor, and I hope friend, Dr. Andrew Martin, for his substantial guidance. In particular, his help on the software change model was invaluable. He has been both gracious and helpful—a combination I very much appreciate. Although I still have a lot to learn about what it means to research, much of what I have learned has been the result of our interactions.

Second, I would like to thank those from the Software Engineering Research Group, within the Computing Laboratory at the University of Oxford, for their suggestions, comments, and encouragement. In particular, I would like to thank Dr. Jim Davies and Dr. Steve McKeever.

Third, and for the same reasons, I would like to thank those that I have worked with and had fun with over the last nineteen years—I have learned so much from you and you have made my career in software development, and my life, rich. In particular, I would like to thank Dr. Scott Bublin, Dr. John and Michele Stutz, and Dr. Mark Wiegand for their help and inspiration, particularly in starting this endeavour. Also, I would like to thank Dr. Linda Rising and Dr. Mark Wiegand for their time and their comments as they have reviewed my work. Additionally, I would like to thank my closest friends for their encouragement: Jason Bernick and Wayne Wilkins.

Finally, I would like to thank those that I love most and who have been the most understanding and supportive while we have undertaken this substantial effort together: my family. Thank you Carol, Faith, Chloe, and Charlotte.

Dominus Illuminatio Mea

Contents

1 Introduction **14**

1.1 Test-First Development 15

1.2 Test-With Development 17

1.3 Multiple Comparative Study 18

1.3.1 Motivation 18

1.3.2 Assumptions 19

1.3.3 Contributions 20

2 Test-First Development **24**

2.1 Introduction 24

2.2 Extreme Programming 25

2.3 Test-First Development 27

2.4 Relevant Studies 28

2.4.1 Study Summaries 29

2.4.2 Study Descriptions 29

2.4.3 Study Limitations 36

2.5 Summary 37

3 Software Change Model **39**

3.1 Introduction 39

3.2 Model Objectives 40

3.3 Specific Changes Example 41

3.3.1 Product Changes 41
3.3.2 Degree of TWD 43
3.3.3 Team Speed 43
3.3.4 Product Quality 45
3.4 General Changes Model 45
3.4.1 Product Changes 46
3.4.2 Degree of TWD 48
3.4.3 Team Speed 48
3.4.4 Product Quality 51
3.5 Summary 52

4 Auxiliary Study **53**
4.1 Introduction 53
4.2 Study Design 54
4.2.1 Motivation 54
4.2.2 Plan 54
4.2.3 Protocol 55
4.3 Study Results 62
4.3.1 Analysis 63
4.3.2 Discussion 68
4.4 Summary 69

5 Study Design **70**
5.1 Introduction 70
5.2 Motivation 71
5.3 Study Plan 71
5.3.1 Objective 71
5.3.2 Research Question 72
5.3.3 Frame of Reference 72
5.3.4 Subject Definition and Selection Strategy 73
5.3.5 Data Collection Methods 74

5.4 Study Protocol 74
5.4.1 Data Collection Procedures 75
5.4.2 Analysis Procedures 81
5.5 Summary 82

6 Study Results **84**
6.1 Introduction 84
6.2 Results Summary 85
6.2.1 Initial Net Size 85
6.2.2 Discards Over Time 88
6.2.3 Net Size Over Time 89
6.2.4 Potential Bugs 91
6.2.5 Average Method Complexity 92
6.3 Results Discussion 96
6.3.1 Some TWD Results 96
6.3.2 All TWD Results 97
6.4 Threats to Validity 98
6.4.1 Compilation Instructions 98
6.4.2 Static TWD Analysis 99
6.4.3 Tool Bias 101
6.4.4 Potential Bugs 102
6.5 Summary 103

7 Conclusion **104**
7.1 Conclusions 105
7.2 Relation to Existing Studies 106
7.3 Implications and Limitations 107
7.4 Future Work 108
7.5 Summary 109

A External Validation **111**

A.1 Abstract 111

B Programming Advances **113**

B.1 Structured Programming 113

B.2 Object-Oriented Programming: Foundation 114

B.3 Object-Oriented Programming: Mainstream 115

C Product Reconstruction **117**

C.1 Revision Identification 117

C.2 Revision Assembly 118

C.3 Revision Build 118

C.4 Revision Storage 119

C.5 Studied Projects 120

D Measurement Details **122**

D.1 Degree of TWD 122

D.2 Initial Net Size 124

E Project Summary Results **126**

E.1 ant 126

E.2 cayenne 129

E.3 commons-codec 131

E.4 ehcache 132

E.5 hadoop-core 134

E.6 xstream 136

F Project Detailed Results **138**

F.1 ant 138

F.2 cayenne 143

F.3 commons-codec 148

F.4 ehcache 153

F.5 hadoop-core 158
F.6 xstream 163

G Electronic Resources **168**
G.1 Product Resources 168
G.2 Measurement Resources 169
G.3 Analysis Resources 169

Glossary **170**

Bibliography **172**

List of Tables

2.1 Effects of TFD–Students 29

2.2 Effects of/Correlations with TFD–Professionals 30

2.3 Studied Attributes of Speed and Quality 30

2.4 Development Durations for Study Subjects 31

4.1 Auxiliary Study Subject Projects 56

E.1 ant—Summary of descriptive statistics 127

E.2 ant—Summary of Welch Two Sample t-test 127

E.3 cayenne—Summary of descriptive statistics 130

E.4 cayenne—Summary of Welch Two Sample t-test 130

E.5 commons-codec—Summary of descriptive statistics 132

E.6 commons-codec—Summary of Welch Two Sample t-test 132

E.7 ehcache—Summary of descriptive statistics 133

E.8 ehcache—Summary of Welch Two Sample t-test 133

E.9 hadoop-core—Summary of descriptive statistics 135

E.10 hadoop-core—Summary of Welch Two Sample t-test 135

E.11 xstream—Summary of descriptive statistics 136

E.12 xstream—Summary of Welch Two Sample t-test 137

G.1 Scripts to build product revisions 168

G.2 Product revisions 168

G.3 Scripts to measure product revisions 169

G.4 Raw product revision measurements 169

G.5 Scripts to analyse product revision measurements 169

G.6 Correlated product revision analyses 169

List of Figures

4.1 ehcache—Preserved Changes over Time (Absolute) 63
4.2 ehcache—Preserved Changes over Time (Relative) 64
4.3 hibernate3—Preserved Changes over Time (Relative) 64
4.4 industry—Preserved Changes over Time (Relative) 65
4.5 tomcat-60X—Preserved Changes over Time (Relative) 65
4.6 xerces—Preserved Changes over Time (Relative) 66
4.7 xstream—Preserved Changes over Time (Relative) 66

6.1 Average Initial Net Size 86
6.2 Confidence Intervals for Initial Net Size 87
6.3 Percentage Differences for Average Initial Net Size 87
6.4 Average Discards Over Time 88
6.5 Confidence Intervals for Discards Over Time 89
6.6 Derived Net Size Over Time 90
6.7 Percentage Differences for Net Size Over Time 91
6.8 Average Change in Potential Bugs 92
6.9 Confidence Intervals for Change in Potential Bugs 93
6.10 Average Change in Average Method Complexity 94
6.11 Confidence Intervals for Change in Average Method Complexity 94
6.12 Percentage Differences for Change in Average Method Complexity 95

F.1 ant—Histogram and boxplot for initial net size 139

F.2 ant—Histogram and boxplot for discards over time 140
F.3 ant—Histogram and boxplot for changes in potential bugs 141
F.4 ant—Histogram and boxplot for changes in average method complexity 142
F.5 cayenne—Histogram and boxplot for initial net size 144
F.6 cayenne—Histogram and boxplot for discards over time 145
F.7 cayenne—Histogram and boxplot for changes in potential bugs . 146
F.8 cayenne—Histogram and boxplot for changes in average method complexity 147
F.9 commons-codec—Histogram and boxplot for initial net size . . . 149
F.10 commons-codec—Histogram and boxplot for discards over time . 150
F.11 commons-codec—Histogram and boxplot for changes in potential bugs 151
F.12 commons-codec—Histogram and boxplot for changes in average method complexity 152
F.13 ehcache—Histogram and boxplot for initial net size 154
F.14 ehcache—Histogram and boxplot for discards over time 155
F.15 ehcache—Histogram and boxplot for changes in potential bugs . 156
F.16 ehcache—Histogram and boxplot for changes in average method complexity 157
F.17 hadoop-core—Histogram and boxplot for initial net size 159
F.18 hadoop-core—Histogram and boxplot for discards over time . . . 160
F.19 hadoop-core—Histogram and boxplot for changes in potential bugs161
F.20 hadoop-core—Histogram and boxplot for changes in average method complexity 162
F.21 xstream—Histogram and boxplot for initial net size 164
F.22 xstream—Histogram and boxplot for discards over time 165
F.23 xstream—Histogram and boxplot for changes in potential bugs . 166
F.24 xstream—Histogram and boxplot for changes in average method complexity 167

Chapter 1

Introduction

Over the past century, many researchers and practitioners have worked to help teams increase their efficiency, by increasing their speed and the quality of their products (Witzel 2002). Taylor, Frank and Lillian Gilbreth, and Emerson lead the development of Scientific Management, to analyse and refine motions and processes so that individuals could execute their tasks more quickly and with less waste (Taylor 1967, Nelson 1992, Emerson 2003, Wood & Wood 2003). Shewhart and Deming spearheaded the introduction of Quality Management, to help analyse and increase the quality of products so that customers would receive greater value and companies would incur less rework costs (Shewhart 1980, Shewhart & Deming 1986, Deming 1986, Gabor 1992). More recently, Hammer and Davenport and Short ushered in Re-engineering Management, identifying the needs to use technology and to re-organise companies and processes so that teams could complete their tasks more quickly and at a higher level of quality (Hammer 1990, Davenport & Short 1990).

In the last half-century, software engineering researchers and practitioners have parallelled that general effort, to increase the efficiency of teams developing software. The attendees at the 1968 NATO conference on software engineering set many of the directions for this effort by proposing tools and techniques to address the societal impacts of software teams taking too long or producing low

quality products (Naur & Randell 1969, Randell 1979, Shaw 1989). Since then, a significant part of this community's effort has been evolving and introducing Software Process Models, to direct teams on *what* they should do and to help them structure and organise their work (Boehm 1986).

The community has evolved and introduced four influential Software Process Models to help software development teams increase their efficiency. Benington presented the "Stagewise" Model, which introduced stages before and after implementation (1983, 1987). Royce refined that model by introducing the "Waterfall" Model, which introduced a limited concept of iteration (1970, 1987). Many have since contributed to its refinement, in the form of an "Iterative and Incremental" Model (Zurcher & Randell 1968, Basili & Turner 1975, Mills 1976, O'Neill 1980, McCracken & Jackson 1982, Gilb 1985, Boehm 1986, Parnas & Clements 1986, Boehm 1988, Cobb & Mills 1990, Degrace & Stahl 1990, Larman & Basili 2003), which introduced ideas related to iteratively delivering a system in increments. Finally, several have refined that model, in the form of an "Agile" Model (Beck 1999*b*, Beck et al. 2001, Schwaber & Beedle 2001, Ambler 2002), which de-emphasises the "stages" before and after implementation and emphasises "customer involvement."

1.1 Test-First Development

Software development teams taking "best practises" to "extreme" levels might be able to develop more efficiently. Kent Beck and his colleagues have pioneered the Extreme Programming (XP) Method (Beck 1999*b*), which is an Agile Method within the Agile Model family, over the course of the last decade. An XP team *listens* to a customer story, writes a set of *automated tests* for that story, evolves the system's *design* and *code* until all of the tests are working, and then *integrates* those changes (Beck 1999*a*). By doing so, they might be able to deliver their product to their customers more quickly and at higher levels of quality.

Test-Driven Development (TDD) is one of XP's "best practises," and possibly *the* key practice (Beck 1999*a*, Jeffries et al. 2001, Auer & Miller 2002); practising TDD might allow a team to develop more efficiently. When a team practises TDD, they write automated tests *before* they write or change their code, to ensure that their changes cause the new tests to move from *failing* to *passing* and to add to the team's test assets. Because they write their product tests before they change their product, they practice the Test-First Development (TFD) step. Teams executing TDD and TFD might be able to develop their product more quickly and at a higher level of quality—it could focus them, removing excess from their activities, and help them use their test assets to improve the internal and external quality of their product.

This hypothesis, that TFD might increase a team's efficiency, has been the subject of some research, and the results have been mixed. One subset of the researchers has studied students to determine the effects of TFD. Typically, they have conducted controlled experiments and overall they have found that practising TFD increased speed a little but had no effect on quality. Another subset of these researchers has studied professionals by conducting controlled experiments and case-studies. Usually, they have studied real-life projects, albeit over relatively short durations, and overall have found that practising TFD was the cause of, or correlated with, slightly decreased speed and slightly or moderately increased quality.

In addition to these mixed results, these studies have had some limitations. To date, the relevant research has assumed that a team either practises TFD or they do not, rather than considering that a team may practice it to a degree. As well, the studies have had relatively short durations, when compared to many "real-life" software development projects. Finally, the studies have compared across projects or products, rather than within them, and have thus been subject to differences due to "contextual variables" (Basili et al. 1986, 1999).

We present an overview of XP, TDD, TFD, the research on the effects of TFD, and some of the limitations of those studies in Chapter 2.

1.2 Test-With Development

When practising TFD, developers evolve their automated product tests before making their fine-grained product changes (Beck 2003), in contrast to practising Test-Last Development (TLD). In TFD, developers add or change their automated product tests first. And, each of their product changes is typically fine-grained and corresponds to a particular feature. But, when practising TLD, developers evolve most or all of their product tests after making a series of product changes, if they evolve their product tests at all. Usually, the series of product changes is large-grained and corresponds to many features. And, in another more traditional variant of TLD, testing specialists evolve manual or automated product tests and then execute them after the development team has developed the features. Thus, there are several potential differences between TFD and TLD: *when* the product changes are made, relative to the product test changes; *how many* features are developed; *who* develops the product tests, the developers themselves or testing specialists; and *what* type of product tests are developed, manual or automated.

Further, we have observed that teams and developers sometimes practice variants of TFD. Sometimes, they evolve their automated product tests and make their product changes at the same time, interleaving them, in support of a particular fine-grained feature. Alternatively, they might make their product changes first and then evolve their automated product tests, also in support of a particular fine-grained feature—some of those performing related studies have differentiated this approach from TFD by referring to it as "iterative test-last" (George & Williams 2003, Pancur et al. 2003, Erdogmus et al. 2005, Janzen & Saiedian 2006, Sanchez et al. 2007). Thus, developers sometimes vary when they change the product and when they change the automated product tests, even if they are making fine-grained changes for a particular feature.

Hence, we define Test-With Development (TWD) as either of these alternative approaches, in addition to the TFD approach. That is, TWD is defined

by the co-evolution of automated product tests and product changes, by the developers themselves, in support of a particular fine-grained feature and independent of whether the developer makes the product changes first or last or in an interleaved manner. We choose the "with" modifier because the changes to the automated product tests are integrated and delivered before or with the related changes to the product, to the rest of the team, usually via a configuration management system.

Finally, we also recognise that developers may practice TWD to a degree. That is, a developer might only develop automated tests for some of the elements of the product changes that they make, instead of developing automated tests for all of them. For example, a Java developer might only develop automated tests for half of the new and changed product methods that they develop when adding a new feature.

1.3 Multiple Comparative Study

1.3.1 Motivation

Therefore, given the potential benefits of TWD, assumed from the potential benefits of TFD, and the potential to improve upon some of the limitations of the existing TFD studies, we were motivated to begin to answer this question:

> *What are the effects of practising TWD, to varying degrees within a project, on team speed and product quality, when those practising it are real-life software development teams, developing their product over the course of many years*?

If the effects on software development team speed and product quality are significantly different, depending on the degree to which a team has practised TWD, then future teams and our society may benefit from this information. Ultimately, knowledge of the differences could allow these teams to work more quickly, or to deliver products of higher quality, or both. And, given the re-

sources that are allocated to buying and developing software, and the importance of reasonably priced high quality software products, this knowledge could have far-reaching effects on teams developing software in particular, and be very valuable to society in general.

1.3.2 Assumptions

Along with the motivation for our study, we had several core assumptions.

First, we assumed that some software developers and teams practice TWD, and that those who do practice it, practice it to varying degrees over the lifetime of their projects. Prior to executing our study, this assumption was based on our industrial experience and on the TFD study descriptions (particularly those that distinguished between "iterative test-first" and "iterative test-last"). Now that we have completed our study, this assumption has been validated by our data—each of the projects we studied exhibited variability with respect to the degree of TWD that was practised, both by individual developers and their teams.

Second, we assumed that we could mine information from project repositories to measure the degree of TWD that developers had practised, and to measure some related effects on team speed and product quality. Prior to executing our study, this assumption was based on the assumptions and successes of others that were mining software repositories. Now that we have completed our study, this assumption has been validated—although we had to formulate a new model for the degree of TWD and for some attributes related to team speed first, this model and two pre-existing models enabled us to mine the relevant information.

Third, we assumed that measuring within a project is better than measuring across projects, because we assume that "contextual variables" (Basili et al. 1986, 1999) are less likely to be different within a project. That is, we assume that the people, process, product, and technology vary less within a project than they do across projects. Because of this assumption, we did not attempt to find

similar projects to compare; rather, we compared within projects. However, although a study of this assumption could provide further valuable information, we did not attempt to validate it within our study.

Finally, we assumed that studying data from long-term real-life projects would be a valuable addition to the knowledge related to TWD and TFD. To date, most or all of the TFD studies have been short-term; but, since many industrial projects are long-term (Emam & Koru 2008) we assumed that a study of similar long-term projects would be valuable. Because of this assumption, we only studied projects that spanned multiple years. However, again, we did not attempt to validate this assumption with our study.

1.3.3 Contributions

Software Change Model

Our first contribution to knowledge is our software change model, which we developed to facilitate measurement of the degree of TWD for a product change as well as two attributes related to team speed for a change: initial net size and discards over time. Soon after beginning our study, we realised that we could mine information from project repositories to measure attributes related to product quality, either directly or indirectly, and with the help of existing models and measurement tools. However, we also realised that we would need to develop a model to support measurement of the degree of TWD and of attributes related to team speed; since then Hannay et al. has encouraged this approach of developing a model to facilitate measurement (2007).

With this software change model, we can describe the degree to which an individual developer practised TWD when they made a change to their team's product. Depending on how many of the product changes were exercised by tests, the developer practised TWD to a degree: *none*, *some*, or *all*. In addition, with this model we can describe the initial net size of a developer's change. Also, this model enables us to estimate how much of a developer's change is discarded

over time by the team, as they continue to develop the product. Together, the initial net size and the discards over time provide a novel measure of the team's speed: the net size of the change over time. Finally, by adopting two pre-existing models, this model enables us to estimate the changes to two attributes of a product's quality: the number of potential bugs and the average method complexity.

We present our software change model in Chapter 3.

Measurement Method

Our second contribution to knowledge is our measurement method, which we developed to facilitate measurement of multiple long-term projects. This measurement method includes our method to calibrate the discards over time aspect of our software change model. It also includes our method to measure the degree of TWD for a change, two of the effects on team speed (initial net size and discards over time), and two of the effects on product quality (the change in the number of potential bugs and the change in the average method complexity).

After defining our software change model, and in preparation for our main study, we conducted an auxiliary study to calibrate the discards over time measure of our software change model. We recognised that to get a "perfect" measurement for the discards over time of a product change, we would need access to the *final* version of that product. But, for products that have a lifetime which extends into the future, this is not practical. So, we sampled discard data from multiple open-source projects and analysed it. We found that, of the changes that will be discarded, 80–90% of them will be discarded within 150 days, and that few of them will be discarded after that. Therefore, we can approximate the discards over time for a particular change if we have access to versions of the product at least 150 days after that change.

With our software change model defined and calibrated, we then designed and performed our main study, a multiple comparative study of six open-source projects. For each of the projects, we rebuilt their product version histories and

then measured, for each product change, the degree of TWD, the initial net size, the discards over time, the change in the number of potential bugs, and the change in the average method complexity. Then, we made the appropriate comparisons and analysed the results to determine the differences and their significances.

We present the method and results for our auxiliary study in Chapter 4. We present the method for our main study in Chapter 5.

Study Results

Our third contribution to knowledge is our main study results; this contribution has been externally validated by its publication in the Empirical Software Engineering Journal (Bannerman & Martin 2010) (the abstract from this publication is included in Appendix A). These results indicate that when the developers that we studied practised TWD while making product changes, the effects of those changes on the attributes related to team speed and product quality were significantly different, as compared to when they did not. Further, the degrees to which they practised TWD corresponded with different effects on the specific attributes that we studied.

When they exercised *some* of their product changes with tests, on average they made significantly bigger product changes over time that resulted in significantly less average method complexity in the product. In contrast, when they exercised *all* of their product changes with tests, they made significantly smaller product changes over time that did not significantly increase or decrease the average method complexity in the product. Finally, the number of potential bugs within the product was not significantly different according to the degree of TWD that a developer practised.

We interpret these results to indicate that automated testing of product changes, *before* sharing those changes with the rest of the team, enabled the teams that we studied to reduce their product's internal complexity more quickly than when they did not. Further, we interpret these results to indicate that the

teams that we studied reduced their product's internal complexity incrementally, rather than all at once.

Based on these interpretations, we conclude that teams can increase their efficiency by practising TWD. In particular, we conclude that they can reduce their product's complexity more quickly when they practice TWD, incrementally adding automated tests to support its simplification. Thus, they can increase their team speed while they are increasing their product's quality.

We present our main study results in Chapter 6. We present our conclusions and summarise our work in Chapter 7.

Chapter 2

Test-First Development

2.1 Introduction

Researchers and practitioners of Scientific, Quality, and Reengineering Management have found that teams can increase their efficiency by completing their products more quickly, or at a higher quality, or both (Chapter 1). Within the realm of Software Engineering, researchers and practitioners have evolved Software Process Models to enable software development teams to structure their activities so that they can develop their products more quickly or at a higher level of quality, culminating in the Iterative and Incremental and Agile Models (Chapter 1). The Extreme Programming (XP) Method is a member of the Agile Model family and its potential to increase a team's efficiency has been the subject of various research studies.

We begin this chapter by introducing XP, TDD, and TFD and analysing why teams executing them might increase their speed or their product's quality. We conclude it by summarising the results of those who have studied TFD's effects on team speed or product quality and by identifying some limitations of their studies.

2.2 Extreme Programming

In 1996, Kent Beck and his colleagues began to define XP. While working on the Chrysler Comprehensive Compensation (C3) project, they refined a set of "best practises" from others in the software community and took them to "extreme" levels (Beck 1999*a*), leading to the method's name. The XP Method can be adopted by teams executing either the Iterative and Incremental or Agile Software Process Models.

The basic activities in the XP method, for a member or pair of members on a team, are: *listen* to a customer story, write a set of *automated tests* for that story, evolve the system's *design* and *code* until all of the tests are working, and then *integrate* those changes. They execute this cycle as frequently as they can within the context of an iteration, which typically lasts two weeks, and develop the next increment of their product. The customer and the team plan an iteration together, based on the customer priorities and the team's estimates. These basic activities are the context for a larger set of practises (Beck 1999*b*).

The "technical premise" of XP is that the "cost of change" (Boehm 1981, McConnell 1993) need not be exponential—in fact, it can approach a constant (Beck 1999*b*). Because of all of the programming advances by the software development community, as partially described in Appendix B, a software team can now change their system in response to a requirements change, with approximately the same effort, whether that change be now or in the future (Beck 1999*b*). Beck reasons that, if this premise is true, the software team can plan, specify, design, program, test and deliver the individual features that are the most valuable to their customer *now*, without worrying about all of the additional features and how much it will cost to change their product *later*. In fact, XP encourages them *not* to add anything into the product that is not absolutely necessary now; it might never be needed.

By executing the practises of XP, a team might be able to deliver their software more quickly. They plan frequent, small releases so that they get their

product into the hands of their customers quickly. By listening to a "real, live" customer, instead of reading a document written or sponsored by the customer, they can better and sooner understand the specific feature that they need to work on next (Beck 1999*b*). By working in pairs, they reap the benefits of design and code inspections immediately (Sfetsos et al. 2006), rather than receiving that feedback at a later date (if at all) (Wood & Kleb 2003). And by investing in automated tests, they get fast and frequent feedback about whether or not their latest changes have adversely affected their product, which removes the need for some manual testing and re-testing (Ellims et al. 2004, 2006). Finally, by continuously integrating their changes, the team has a product that is "always" ready to be delivered to their customer and they avoid the time and cost of "big-bang" integration (Duvall 2007).

By following the XP practises, a team might also be able to deliver a product with higher quality. They are encouraged to refactor the system when they have the need or an opportunity to do so, to improve the internal quality of the product (Beck 1999*b*). They should write their code according to coding standards, so that it is easier for others to read and change (Beck 1999*b*). And they should always keep the automated tests running, to ensure that the product quality is maintained or improved (Beck 1999*b*). Each of these practises promotes higher quality, in the sense that they reduce later rework.

Finally, in several ways, XP is an example of Reengineering Management applied to Software Engineering Process, particularly the Stagewise/Waterfall Model. It revamps the team's activities (Champy 1996) at the process level (Malhotra 1998) and takes advantage of new technology (Davenport & Short 1990) to do so across organisational boundaries (Hammer 1990). One revamped idea is "test-first" development (TFD), supported by automated tests, in contrast to "test-last" development (TLD).

Therefore, the XP Method provides a team the potential to both increase its speed and the quality of their product—that is, to increase their efficiency. Within XP, the TDD practice and its TFD step provide some of this potential.

2.3 Test-First Development

TDD is a practice that a team can follow to write automated tests for a particular feature, *before* they design and code that feature (Beck 2003). After they have discovered what they have to do, by either listening to their customer or their teammates, they write a representative test or tests and ensure that the tests *fail*—this is the TFD step within the TDD practice. Then, they evolve their design and code by adding and refactoring, until the failing tests and all of the other tests pass. Next, if necessary, they refactor the code to remove duplication and improve its quality. By making sure that the new tests move from failing to passing, they become confident that the new feature has been implemented.

TDD is a lot more than testing—in fact, it is a combination of analysis, specification, design, coding and testing (Beck 2001). It is analysis because the developer listens and builds an understanding of what they need to do. It is specification because they then write automated tests that describe the pre-conditions and post-conditions for new or changed product behaviour, usually in the form of assertions. It is design and coding because they evolve a solution once they have their tests in place. And, it is testing because they ensure that the new tests, and all of the existing automated tests, execute without error or failure.

By executing the TDD practice, and the TFD step, a team might be able to deliver individual features more quickly. The failing tests that they write provide very specific guidance on the problems that they need to focus on and solve, potentially decreasing wasted "motion," much like the Gilbreths sought to decrease the wasted motions of manufacturers (Wood & Wood 2003). And because they automate their tests, they use technology to receive rapid feedback about how far they have come and how much farther they have to go, rather than relying on manual execution of those tests. And, should any of the existing tests begin to fail, due to their design, coding, and refactoring, they receive the

same benefits from those tests: guidance and rapid feedback.

A team executing TDD and TFD might also be able to deliver individual features at a higher level of quality. Because refactoring is encouraged, and even mandated, the team is free to improve the internal quality of the product. For example, they can remove duplication, decrease coupling, increase cohesion, and simplify their design when they have the opportunity to do so. The automated tests give them more confidence that they have not broken any of the existing features.

Finally, a team executing TDD and TFD is more likely to have "test-assets" and these assets provide some of the potential for a team to increase their speed or their product's quality. In contrast to TFD teams, teams executing traditional practises (TLD) have developed their automated tests *last*. However, sometimes time constraints or external pressures can prevent a TLD team from developing their automated tests. As a result, a TFD team is more likely to have automated "test-assets"—some studies have confirmed this (Erdogmus et al. 2005, Canfora et al. 2006, Janzen & Saiedian 2006).

2.4 Relevant Studies

Because of its potential to increase team efficiency, several researchers have performed studies to investigate TFD's effects on specific attributes of team speed or product quality; the results of their studies have been mixed. Some have found that team speed increases or stays roughly the same when individuals or a team execute TFD. Others have found that team speed decreases, if only slightly. However, some have not found any effect on speed. Further, some have found that either the internal or external product quality increases. But, others have been unable to determine the effect of TFD on product quality.

2.4.1 Study Summaries

We have summarised the type of study and the effects on speed and quality that these researchers found in Tables 2.1 and 2.2. Table 2.1 summarises the studies where students were the subjects. If anything, those who practised TFD in these studies increased their speed without affecting their quality. Table 2.2 summarises the studies where professionals were the subjects. In contrast, those who practised TFD in these studies lost some speed but increased their quality.

Additionally, we have summarised the attributes of speed and quality that these researchers studied and the durations of the development tasks for their subjects in Tables 2.3 and 2.4. Table 2.3 summarises the attributes of speed and quality that were studied. A variety of speed attributes were studied, and most of them have an element of time. As well, a variety of quality attributes were studied; some of the attributes measured internal quality and some of the attributes measured external quality. Table 2.4 summarises the development durations for each study's subjects. About half of the studies had a duration of a day or less; the other half had durations of weeks or months and most were less than nine months.

Publication	*Type*	*Speed*	*Quality*
Muller & Hagner (2002)	Controlled	No difference	No difference
Pancur et al. (2003)	Controlled	–	No difference
Erdogmus et al. (2005)	Controlled	Faster	No difference
Madeyski (2005)	Controlled	–	Lower
Janzen & Saiedian (2006)	Controlled	Faster	No difference
Gupta & Jalote (2007)	Controlled	Faster	No difference

Table 2.1: Effects of TFD–Students

2.4.2 Study Descriptions

Muller & Hagner performed a controlled experiment with nineteen computer science graduate students, during a two month period (2002). In their experiment, they investigated the effect of TDD on process time (total time) and product quality (reliability). Their control group executed a "traditional" process while

Publication	*Type*	*Speed*	*Quality*
George & Williams (2003)	Controlled	Slower	Higher
Maximilien & Williams (2003)	Case Study	No difference	Higher
Williams et al. (2003)	Case Study	–	Higher
Geras et al. (2004)	Controlled	No difference	–
Abrahamsson et al. (2005)	Case Study	–	–
Bhat & Nagappan (2006)	Case Study	Slower	Higher
Canfora et al. (2006)	Controlled	Slower	–
Damm & Lundberg (2006)	Case Study	–	Higher
Sanchez et al. (2007)	Case Study	–	Higher
Siniaalto & Abrahamsson (2007)	Case Study	–	Higher
Nagappan et al. (2008)	Case Study	Slower	Higher

Table 2.2: Effects of/Correlations with TFD–Professionals

Publication	*Team Speed*	*Product Quality*
Muller & Hagner (2002)	total time	reliability
George & Williams (2003)	productivity (time)	acceptance tests
Maximilien & Williams (2003)	LOC per month	defect rate
Pancur et al. (2003)	–	code coverage, tests passed
Williams et al. (2003)	–	defect density
Geras et al. (2004)	total effort	unplanned failures
Abrahamsson et al. (2005)	LOC, effort, productivity	–
Erdogmus et al. (2005)	productivity (# of tests)	acceptance test conformance
Madeyski (2005)	–	acceptance tests passed
Bhat & Nagappan (2006)	time estimates	defect density
Canfora et al. (2006)	time to complete	–
Damm & Lundberg (2006)	–	fault rate
Janzen & Saiedian (2006)	total effort	internal metrics
Gupta & Jalote (2007)	effort, productivity	acceptance tests passed
Sanchez et al. (2007)	–	defect density, complexity
Siniaalto & Abrahamsson (2007)	–	program design, code coverage
Nagappan et al. (2008)	time estimates	defect density

Table 2.3: Studied Attributes of Speed and Quality

Publication	*Development Duration*
Muller & Hagner (2002)	2 months
George & Williams (2003)	1 day (or less)
Maximilien & Williams (2003)	7 months
Pancur et al. (2003)	5 months
Williams et al. (2003)	7 months
Geras et al. (2004)	1 day (or less)
Abrahamsson et al. (2005)	9 weeks
Erdogmus et al. (2005)	26 hours (or less)
Madeyski (2005)	8 weeks
Bhat & Nagappan (2006)	7-12 months
Canfora et al. (2006)	1 day (or less)
Damm & Lundberg (2006)	12-18 months
Janzen & Saiedian (2006)	1 day (or less)
Gupta & Jalote (2007)	3 weeks
Sanchez et al. (2007)	5 years
Siniaalto & Abrahamsson (2007)	9 weeks
Nagappan et al. (2008)	7-12 months

Table 2.4: Development Durations for Study Subjects

their experiment group executed a "test-first" process. They found that there were no significant differences between the speed or quality of the two groups.

George & Williams performed a controlled experiment with twenty-four professional programmers from three companies (2003)—their study's development tasks took less than a day. In their experiment, the control group was pairs of programmers who were asked to develop a small Java program in the following (Waterfall) sequence: design, code, test, and debug. Their other group, the TDD group, was also pairs of programmers who were asked to develop the same small program in a TDD manner: writing tests before or along with their design and code activities. They measured two attributes for each of the groups: productivity (time) and external quality (acceptance tests). On average the TDD group developed the program 16% slower with an 18% higher quality.

Maximilien & Williams conducted a case study on an IBM Retail Store Solutions project, studying data from a seven month development effort (2003). In their case study, a team of nine full-time engineers in two locations developed a Java Point of Sale (POS) application. This team practised TDD, as compared to the "ad-hoc" testing that had been the norm in the organisation. For this

team, they measured their productivity (lines of code per month) and external quality (defect rate). Then, they compared these measurements with similar projects that had been completed within the same organisation. The team that practised TDD developed at approximately the same speed with a 50% improvement in quality.

Pancur et al. performed a controlled experiment with thirty-eight senior undergraduate computer science students, during a five month period (2003). In their experiment, the control group executed an iterative test-last (ITL) process while the experiment group executed an iterative test-first process (TDD). They investigated the influence of TDD on product quality (code coverage and tests passed). They found that there were no significant differences between the quality of the two groups. If anything, both the code coverage and tests passed of the experiment group were slightly less than the control group (both approximately 2.5% less).

Later, Williams et al. published additional results from their previously mentioned case study of the IBM Retail Store Solutions project (2003). While the case study details are the same, they also analysed the effect of TDD on another aspect of quality: defect density. The team that practised TDD developed their product while introducing 40% fewer defects per line of code.

Geras et al. performed a controlled experiment with professional programmers that volunteered to participate (2004)—their study's development tasks took less than a day. In their experiment, the control group executed a test-last process while the experiment group executed a test-first (TDD) process. They investigated the influence of TDD on process speed (total effort) and product quality (unplanned failures). They found that both groups expended approximately the same total effort. They were not able to determine whether one group received more unplanned failures than the other.

Abrahamsson et al. conducted a controlled case study on a VTT Technical Research Centre project, studying data from a nine week development effort (2005). In their case study, a team of four developers, three of which

were senior students, developed a monitoring application for mobile Java devices. This team practised TDD, to a very limited extent—the team did not see much value in doing so. For this team, they measured their productivity (lines of code per hour) and the number of tests written (lines of code). They reported these measurements, but did not compare them with similar projects or a control group.

Erdogmus et al. performed a controlled experiment with undergraduate students (2005)—their study's development tasks took less than twenty-six hours. In their experiment, the control group executed a test-last iterative and incremental process while the experiment group executed a test-first iterative and incremental (TDD) process. They investigated the influence of TDD on process speed (productivity due to an increase in written tests) and product quality (conformance to acceptance tests). They found that the test-first group wrote more tests, and inferred that their productivity was higher because more tests, across both groups, was found to imply higher productivity. They also found that neither group outperformed the other in product quality.

Madeyski performed a controlled experiment with graduate students (2005)—their study's development tasks took eight weeks (90 minutes per week). In their experiment, the two control groups (single and pair programming teams) executed a classic testing approach while the experiment groups (single and pair programming teams) executed TDD. They investigated the influence of TDD on the external quality of the product (number of acceptance tests passed). They found that both of the experiment groups provided significantly lower quality products, although they did not indicate how much lower.

Bhat & Nagappan conducted two industrial case studies within two divisions at Microsoft, studying data from seven and twelve month development efforts (2006). In their case study, one team of six developed a Windows product and another team of between five and eight developed a Microsoft Network (MSN) product. These teams practised TDD, and were compared with projects that had previously been completed within the same division (and presumably

did not practice TDD). For these teams, they measured their speed (time estimates) and quality (defect density). They found that the Windows TDD team developed 25–35% more slowly but with 2.6 times less defects. Similarly, they found that the MSN TDD team developed 15% more slowly but with 4.2 times less defects.

Canfora et al. performed a controlled experiment with twenty-eight professional developers at a Spanish software company (2006)—their study's development tasks took less than a day. In their experiment, each subject executed a test-last and a test-first process; the test-last executions constituted the control group and the test-first executions were designated as the experiment group. They investigated the influence of TDD on process speed (time to complete) and product test quality (number of assertions) (but did not investigate product quality). They found that the test-first group took more time but developed more and better tests.

Damm & Lundberg conducted a controlled case study on an Ericsson AB project, studying data from a twelve to eighteen month development effort (2006). In their case study, four teams of around fifty people developed mobile network components. Two teams practised TDD and two teams did not. For this study, they measured and compared the product quality (faults that slipped through). They found that the teams that practised TDD reduced the number of faults that slipped through by 60%, which reduced the overall cost of the faults by 30%.

Janzen & Saiedian performed a controlled experiment with students (2006)—their study's development tasks took less than a day. In their experiment, the control group executed a test-last iterative process while the experiment group executed a test-first iterative (TDD) process. They also trained and defined a third group, which executed a no-test process, for interest and comparison purposes. They investigated the influence of TDD on process speed (total effort) and product quality (internal metrics). They found that the test-first group wrote more tests and were more productive. They also found that the test-first

and test-last groups developed a higher quality product than the no-test group, but found that there was no significant difference between the test-first and test-last product qualities.

Gupta & Jalote conducted a controlled experiment with students (2007)—their study's development tasks took around three weeks. In their experiment, the control group executed a conventional development process while the experiment group executed a TDD process. They investigated the influence of TDD on process speed (productivity and effort) and product quality (passed acceptance tests). They found that the TDD group developed at a higher speed (increased productivity and less effort). They also found that the TDD group developed a lower quality product (less passed acceptance tests).

Sanchez et al. published additional post-hoc case study results from the previously mentioned IBM Retail Store Solutions project (2007). In their study, they analysed two additional quality attributes of the product—at that point, the project had been ongoing for five years. They found that practising TDD resulted in a higher quality product (a sustained lower defect density and a lower complexity increase over time).

Siniaalto & Abrahamsson conducted a controlled case study on a VTT Technical Research Centre project, studying data from a nine week development effort (2007). In their case study, three teams of four to five people developed intranet applications. One team practised TDD and two teams practised an iterative test-last approach, and all teams were encouraged to write automated tests. For this study, they measured and compared the product quality (program design and code coverage). They found that the team that practised TDD generated greater code coverage; however, they did not find a difference in program design.

Most recently, Nagappan et al. published the results of four industrial case studies that they had conducted within IBM and three divisions at Microsoft (2008). Three of the case studies were previously described (Maximilien & Williams 2003, Bhat & Nagappan 2006) and the fourth involved a team of

seven, developing the Visual Studio product at Microsoft, during a seven month time period. This team practised TDD, and was compared with a project that had previously been completed within the same division. They measured their speed (time estimates) and quality (defect density). They found that the TDD team developed 20–25% more slowly but with 90% fewer defects.

2.4.3 Study Limitations

Based on our analysis of these studies, we have identified several limitations.

First, most of these studies have been relatively short in duration: hours, days, weeks, and months rather than months or years (Table 2.4). But, according to a replicated survey of global IT executives, at least half of the projects in 2005 and 2007 (both successful and failed) had a duration of more than nine months, and less than ten percent of them were less than a month (Emam & Koru 2008). Therefore, what we have learned from these shorter duration TFD studies may not be applicable to most real-life projects because of the differences in project duration.

Second, all of these studies have assumed that a team and its developers either practise TFD or they do not. Based on our experience in industry, this is a bad assumption; although a few projects are quite disciplined about practising TFD, we have seen and we also suspect that even they "slip up" now and then, whether intentionally or unintentionally. And, based on our study results (Chapter 6), each project we studied exhibited variability with respect to the degree of TWD that their developers practised. Therefore, what we have learned from these TFD studies may not be applicable to projects where TFD is practised to a degree, rather than wholly or not at all.

Third, none of the studies have contrasted TWD and TLD; some have contrasted one variant of TWD with another and others have contrasted TFD with TLD. But, as we stated in our TWD definition in Chapter 1, developers sometimes vary when they change their automated product tests and their product. Therefore, what we have learned from these TFD studies may not be applica-

ble to projects where the three variants of TWD are practised, as opposed to projects where only one variant is practised.

Finally, each of the studies have either contrasted groups in a controlled environment or they have compared similar but not identical real-life projects, rather than contrasting or comparing *within* a project. But, as we stated in our assumptions in Chapter 1, measuring within a project provides for better comparisons because "contextual variables" such as people, process, product, and technology are less likely to change within a project. Therefore, although great care was taken to compare similar individuals or teams in these prior TFD studies, their results may be suspect due to differences in "contextual variables."

2.5 Summary

XP in general, and TFD in particular, provide the potential for software teams to increase their efficiency. In theory, by planning and executing small releases, listening to their customer, working in pairs, refactoring their code, and writing and maintaining automated tests before changing their product, they are capable of delivering a higher quality product more quickly.

Accordingly, several researchers have studied the effects of TFD on team speed or product quality, and the results have been mixed. These researchers have found that there is a slight increase in speed when they study students that practice TFD. In contrast, they have also found that there is a slight decrease in speed and a slight to considerable increase in quality when they study professionals that practice TFD.

In addition to the mixed results of these studies, we recognise a few limitations that affect their applicability to real-life projects. Most of these studies have been relatively short in duration. And, all of them have assumed that a team practises TFD or that they do not. Further, none of the studies have contrasted TWD and TLD. Finally, they have either contrasted groups in a

controlled environment or they have compared similar but not identical real-life projects, rather than contrasting or comparing *within* a project.

Chapter 3

Software Change Model

3.1 Introduction

Teams can increase their efficiency by completing their products more quickly, or at a higher quality, or both (Chapter 1). Within the realm of Software Engineering, they can execute a Software Process Model to structure their activities so that they can develop their products more quickly or at a higher level of quality (Chapter 1). Test-With Development (TWD) is a technique that they can practice to potentially increase their speed and their product's quality (Chapters 1 and 2). But, because of the mixed results and some of the limitations of previous related studies (Chapter 2), we were motivated to study the effects of TWD. However, before we could design and perform a study, we needed a model to support the measurements for it, an approach that has since been recommended by Hannay et al. (2007).

We begin this chapter by presenting our objectives for our software change model. Then we present an example, using Java code, to introduce some of its terminology and ideas. Finally, we present it formally using the Z notation (Davies & Woodcock 1996), to communicate our measures precisely. Within its formal definition, we discuss some of our modelling decisions.

3.2 Model Objectives

Ideally, we wanted our model to support direct measurements on versions of a product itself—the third degree method identified by Lethbridge et al. (2005). Otherwise, we would have been faced with the significant and potentially overwhelming challenge of normalising and associating a disparate set of artifacts, from a disparate set of repositories, with different levels of associations between them. But, by studying the versions of the products themselves, we could avoid this overhead.

For example, one team might represent the features they have implemented with index cards and another might might use an issue management system. If we were to measure team speed with the number of features added for a product change, then we would have to convert the index card and issue measurements to a normalised measure for the number of features. Assuming that we had access to the information, this normalisation task would require some effort for each speed variant that we wanted to study. Further, we would have to relate that normalised measure to a particular product change, based on an association between the two. Assuming that such an association existed and was accessible, this association task would also require some effort.

Additionally, we wanted our model to support analysis of compiled versions of a product, as opposed to source code versions of a product. Source code analysis of a product may be imprecise, as was the case when Zaidman et al. inferred a relationship between product classes and product test classes based on a naming convention for the corresponding source files (2008). And, source code analysis of a product may be subjective, as has been the case when many have measured productivity with Lines of Code (LOC) measurements (Boehm et al. 2000). In contrast, compiled code analysis can be precise and objective, because relationships within the code are explicit and organisational changes, such as modifying blank or commented lines and alphabetising methods, are ignored during compilation.

So, to make the best use of our efforts and to facilitate answering our research question across *many* projects, we defined a model and adopted two others that allow us to make direct measurements on compiled versions of products. We defined a model which allows us to make measurements for the degree of TWD that a developer practised while making a product change, as well as two attributes related to the team speed: initial net size and discards over time. We also adopted two pre-existing models and tools which allow us to make measurements for two attributes related to product quality: the change in the number of potential bugs and the change in the average method complexity.

3.3 Specific Changes Example

3.3.1 Product Changes

Product Methods

Consider a contrived software product, written in Java. Further, consider a single product class, Amount, within that product which could be used to model the attributes and behaviour for an amount of money. Listing 3.1 shows a single product method for this class.

```
public class Amount {
  public Amount add(Amount amount) {
    return new Amount(getValue() + amount.getValue());
  }
}
```

Listing 3.1: Product Revision 1

Within the product method, the developer has coded instructions that will be compiled for a (virtual) machine to execute. She has instructed the machine to get the receiver's value, get the other amount's value, to add them together, and then construct and return a new amount object with the result.

Product Method Exercise

Now, assuming the team is maintaining an automated suite of product tests, a test class named AmountTest might look like Listing 3.2. It includes a simple test to ensure that the original Amount remains unchanged and that the new Amount has the correct value, within a "blue sky" add amount scenario—"blue sky" is sometimes used to describe tests for normal circumstances, or positive tests, as opposed to "non-blue sky" which is sometimes used to describe tests for abnormal circumstances, or negative tests.

```
public class AmountTest extends TestCase {
  public void testBlueSkyAdd() {
    Amount srcAmt = new Amount(5.0);
    Amount dstAmt = srcAmt.add(new Amount(10.0));
    assertEquals(srcAmt.getValue(), 5.0, 0.0);
    assertEquals(dstAmt.getValue(), 15.0, 0.0);
  }
}
```

Listing 3.2: Product Tests Revision 1

Because this test method will exercise the product "add(Amount)" method when it is compiled and executed, we consider the product method to be exercised by the product tests. Further, if the team is practising TWD, they will have developed, integrated, and committed this test method *with* the product method, rather than do so after they have developed several other features.

Product Method Changes

Next, to make the product more useful, a particular developer modifies the Amount class so that amounts of other currencies can be added to an existing amount. He adds a new and more specific method, which accepts an exchange rate (Listing 3.3).

```
public class Amount {
  public Amount add(Amount amount) {
    return new Amount(getValue() + amount.getValue());
  }
  public Amount add(Amount amount, double exchangeRate) {
    double exchangeValue = amount.getValue() * exchangeRate;
    return new Amount(getValue() + exchangeValue);
  }
}
```

Listing 3.3: Product Revision 2

3.3.2 Degree of TWD

Assuming that this developer did not modify the automated tests before committing the product changes, the original product method is still exercised by one test. However, the new product method is not. Therefore, in this case, the developer only practised TWD to a degree.

3.3.3 Team Speed

Initial Net Size

In making the changes from product revision 1 (Listing 3.1) to product revision 2 (Listing 3.3), the developer added a method. At a more precise level, the developer added a certain number of instructions, both in the source code and in the resulting compiled instructions. And, since he did not delete any compiled instructions, the initial net size of his change is equal to the size of the compiled instructions that he added.

Discards Over Time

Now, if we look at this class six months later, we might observe that the team has added a currency attribute to the Amount class. Thus, they may have removed the "new" method that had accepted an exchange rate because it was no longer necessary. The newest product method, which has the same signature as the original method, now determines the exchange rate by considering the currency of both amounts, the receiver and the argument (Listing 3.4).

```
public class Amount {
  public Amount add(Amount amount) {
    double exchangeRate = getExchangeRate(amount);
    double exchangeValue = amount.getValue() * exchangeRate;
    return new Amount(getValue() + exchangeValue);
  }
}
```

Listing 3.4: Product Revision N

Now, if we reconsider the Amount class at revision 1 (Listing 3.1), we notice that the signature of the "add" method has remained unchanged, with respect to revision N (Listing 3.4). However, the compiled instructions within the method have changed. Some of the original compiled instructions have persisted, others have been discarded over time, and new ones have been added.

Net Size Over Time

Similarly, if we reconsider the Amount class at revision 2 (Listing 3.3), we notice that the team has deleted the second method, with respect to revision N (Listing 3.4). Therefore, all of its compiled instructions were discarded over time. Thus, we can adjust the initial net size of the change by the discards over time to represent the net size over time for the change—in this case the net size over time is zero instructions per six months because all of the initial compiled

instructions were deleted.

3.3.4 Product Quality

Further, we can compare the compiled class from revision 1 with the compiled class from revision 2 to determine the change in two of its attributes related to product quality: the number of potential bugs and its average method complexity. By calculating the number of potential bugs within the class in revision 1 and again within revision 2, we can determine the effect of the change on the number of potential bugs. Similarly, by calculating the average method complexity of the class in revision 1 and again in revision 2, we can determine the effect of the change on the average method complexity.

3.4 General Changes Model

In the preceding specific example, we introduced our software change model. Products are composed of methods, which may be exercised by product tests. And, for each change, a developer practises TWD to a certain degree, making changes of a certain initial net size, some of which may be discarded over time (by a particular future revision), resulting in a net size over time. Further, each change may affect the number of potential bugs within the product and the average method complexity of the product.

Now, we define these concepts formally, except for the product quality concepts, by presenting them using the Z notation; since we adopt the product quality models and tools, we simply refer the reader to their definitions. And, although we considered using the Unified Modeling Language (UML) (OMG 2007) to communicate our model, we feel that the Z notation is both more precise and concise for our particular needs. Additionally, because we used the Z notation, we were able to describe our model formally and to use the "fuzz" type checker (http://spivey.oriel.ox.ac.uk/mike/fuzz) to validate its syntax.

3.4.1 Product Changes

Product Methods

Consider a particular build, build i, that contains a set of classes, C_i, and that within those classes are a set of methods, M_i, and that within those methods are a set of instructions, I_i. Assume that each of these are made up of two distinct subsets: (1) the product classes, CP_i, and the test classes, CT_i; (2) the product methods, MP_i, and the test methods, MT_i; and (3) the product instructions, IP_i, and the test instructions, IT_i.

$$C_i = CP_i \cup CT_i$$
$$M_i = MP_i \cup MT_i$$
$$I_i = IP_i \cup IT_i$$

Product Method Exercise

Typically, a team writes test methods that execute tests against particular product operations, which are implemented by product methods. When a developer does this, some of the instructions in a test method will cause a "message send" when it executes, directly or indirectly, to the object under test, which will cause the execution of a product method.

As a result, in a build we have a relation between the test instructions and the product methods. Some of the product methods will have test instructions that "exercise" them, MPE, but others will be "unexercised," MPU. Additionally, some of the test instructions may exercise multiple product methods, if they send messages indirectly through an interface, rather than directly through a class.

$$testsOperation : (IT \leftrightarrow MP)$$
$$MPE = \operatorname{ran} testsOperation$$
$$MPU_i = (MP_i) \setminus (MPE_i)$$

Product Method Changes

To describe the degree to which a team practised TWD for a particular change set, we first need to know which product methods changed. We can do so by recognising two characteristics of a method: its signature and its implementation. Each method has a signature, which both identifies its declaring class and distinguishes it from other methods in that class. And each method declares an implementation, which specifies instructions to a (virtual) machine.

$$sig : Method \rightarrow Signature$$
$$impl : Method \rightarrow Implementation$$

Consider two builds, builds i and j. If a product method is not in build i, and is in build j, then it has been added and is a member of MPA. Similarly, if it is in build i, but not in build j, then it has been deleted and is a member of MPD. Finally, if it is in both builds, then it has been retained, and the pair of methods is a member of MPR.

$$MPA_{ij} = \{\, x : MP_j \mid \neg\, (\exists\, y : MP_i \bullet sig(x) = sig(y)) \,\}$$
$$MPD_{ij} = \{\, x : MP_i \mid \neg\, (\exists\, y : MP_j \bullet sig(x) = sig(y)) \,\}$$
$$MPR_{ij} = \{\, x : MP_i;\ y : MP_j \mid (sig(x) = sig(y)) \bullet (x, y) \,\}$$

Within the retained product methods, MPR, each method pair is either exactly the same in both builds or it has been modified. If its instructions have changed, or the order of its instructions has changed, then it has been modified and is a member of MPM. MPM^{old} represents the modified methods from build i while MPM^{new} represents the modified methods from build j. Otherwise, it is identical and is a member of MPI.

$$MPM_{ij}^{old} = \{\, x, y : Method \mid (x, y) \in MPR_{ij} \wedge impl(x) \neq impl(y) \bullet x \,\}$$
$$MPM_{ij}^{new} = \{\, x, y : Method \mid (x, y) \in MPR_{ij} \wedge impl(x) \neq impl(y) \bullet y \,\}$$
$$MPI_{ij} = \{\, x, y : Method \mid (x, y) \in MPR_{ij} \wedge impl(x) = impl(y) \bullet y \,\}$$

3.4.2 Degree of TWD

We can now define the degree of TWD that a developer practised when making a change, from build i to build j. We define it as the number of product methods that have been added or modified and are exercised by tests (in build j), divided by the number of product methods that have been added or modified (Equation 3.1).

$$DegreeOfTWD_{ij} = \frac{\#((MPA_{ij} \cup MPM_{ij}^{new}) \cap MPE_j)}{\#(MPA_{ij} \cup MPM_{ij}^{new})} \tag{3.1}$$

3.4.3 Team Speed

Initial Net Size

Product methods are either added, deleted, modified, or identical when a developer makes a product change. By considering the size of each of the changes to the product methods, from build i to build j, we can define the initial net size of a product change.

Product methods are composed of product instructions. When a developer adds product methods in build j, they add those instructions, IPA_{ij}. When a developer deletes product methods from build i, they delete those instructions, IPD_{ij}. When a developer modifies product methods in build j from build i, they modify instructions in the old method, IPM_i, thereby creating a new method IPM_j.

$$
\begin{array}{|l}
methodInstructions : Method \rightarrow \mathbb{P}\, Instruction \\
\hline
IPA_{ij} = \bigcup \{\, x : MPA_{ij} \bullet methodInstructions(x) \,\} \\
IPD_{ij} = \bigcup \{\, x : MPD_{ij} \bullet methodInstructions(x) \,\} \\
IPM_i = \bigcup \{\, x : MPM_{ij}^{old} \bullet methodInstructions(x) \,\} \\
IPM_j = \bigcup \{\, x : MPM_{ij}^{new} \bullet methodInstructions(x) \,\}
\end{array}
$$

But, when a developer modifies a product method in build j, they either add, change, or delete instructions. If the instructions from the modified methods in build i, IPM_i, are bigger than the modified methods in build j, IPM_j, then the developer has deleted some instructions. If the instructions from the modified methods in build j, IPM_j, are bigger than the modified methods in build i, IPM_i, then the developer has added some instructions. Otherwise, the developer has changed some instructions but the number of instructions is the same—in this case there is no contribution to the initial net size of the change.

With these elements, we define the initial net size of a product change as the sum of the number of added instructions, IPA_{ij}, minus the number of deleted instructions, IPD_{ij}, plus the difference between the size of the modified instructions from build j and build i.

$$InitialNetSize_{ij} = \#(IPA_{ij}) - \#(IPD_{ij}) + (\#(IPM_j) - \#(IPM_i)) \tag{3.2}$$

Discards Over Time

Now, consider a future build, build N, along with builds i and j. We have already introduced the "parts" we need to represent the changes between build i and build j, Δ_{ij}. We can combine the added, deleted, and modified product methods, selecting the modified methods that are part of build j. Similarly, we can represent the changes between build i and a future build N, Δ_{iN}, and between build j and build N, Δ_{jN}.

$$\Delta_{ij} = MPA_{ij} \cup MPD_{ij} \cup MPM_{ij}^{new}$$
$$\Delta_{iN} = MPA_{iN} \cup MPD_{iN} \cup MPM_{iN}^{new}$$
$$\Delta_{jN} = MPA_{jN} \cup MPD_{jN} \cup MPM_{jN}^{new}$$

Assume the team has made a sequence of changes to their product and are now at build N. The changes they made in build j, Δ_{ij}, have either persisted, ΔP_{ij}, or been discarded, ΔD_{ij}, at build N. Put another way, the discarded changes from build j are equal to the changes from build j minus the persistent changes from build j (Equation 3.3).

$$\Delta D_{ij} = \Delta_{ij} \setminus \Delta P_{ij} \tag{3.3}$$

Now, Δ_{iN} and Δ_{jN} are by their nature a representation of changes that have persisted until build N—they abstract the sequence of persisted and discarded changes from their start build until build N. Therefore, the changes that have persisted from build j, ΔP_{ij}, are equal to the difference between them (Equation 3.4).

$$\Delta P_{ij} = \Delta_{iN} \setminus \Delta_{jN} \tag{3.4}$$

Hence, by substituting for ΔP_{ij}, from Equation 3.4 and into Equation 3.3, we arrive at a revised definition for discards for build j (Equation 3.5). With this definition, we can derive ΔD_{ij} by making three comparisons: Δ_{ij}, Δ_{iN}, and Δ_{jN}. Thus, we can measure the discards for a change, by making three supporting measurements.

$$\Delta D_{ij} = \Delta_{ij} \setminus (\Delta_{iN} \setminus \Delta_{jN}) \tag{3.5}$$

Now, consider that the discards from the initial changes of build i in build j will take place between the time of build j and the time of build N, Δt_{jN}. This consideration allows us to define the discards over time for the changes made to

build i in build j (Equation 3.6).

$$DiscardsOverTime_{ij} = \frac{\Delta D_{ij}}{\Delta t_{jN}} \tag{3.6}$$

Net Size Over Time

Finally, we can define our team speed measure for the changes made to build i in build j: the net size over time. Those changes have an initial net size (Equation 3.2). And, some of those changes will be discarded over time (Equation 3.6). So, we define the net size over time as the adjustment of the initial net size by the discards over time (Equation 3.7).

$$NetSizeOverTime_{ij} = InitialNetSize_{ij} - DiscardsOverTime_{ij} \tag{3.7}$$

3.4.4 Product Quality

Number of Potential Bugs

Rather than defining our own model for the number of potential bugs, we adopted the findbugs model (Ayewah et al. 2008). Additionally, we adopted the findbugs (http://findbugs.sourceforge.net) static analysis tool to support our measurement of the number of potential bugs within the product methods in a change. Some of our main study's reviewers have questioned the validity of our choice of this model; we discuss their concerns about it and our justification for it in Section 6.4.

Average Method Complexity

Similarly, rather than define our own model for average method complexity, we adopted the cyclomatic complexity model (McCabe 1976). Additionally, we adopted the cyvis (http://cyvis.sourceforge.net) static analysis tool to support our measurement of the average method complexity for the product methods in a change.

3.5 Summary

To study the effects of TWD product changes on a team's speed and their product's quality, we first needed the capability to describe the degree to which a developer had practised TWD when making a change to their product. We also needed the capabilities to measure the effects of such a change on some attributes of the team's speed and the product's quality. And, we desired to make direct measurements to avoid unnecessary overheads and to allow our model to be used to measure *many* products. To meet these needs, we developed a software change model that we could implement (or adopt) to measure data and support our study.

According to our model, the degree of TWD that a developer has practised when making a product change is equal to the number of changed and exercised methods divided by the number of changed methods (Equation 3.1). In addition, it defines the initial net size of a change as the number of added instructions, minus the deleted instructions, plus the net size of the modified instructions (Equation 3.2). It also defines the discards over time for a change, by a certain future product version, as the difference between that change and its predecessor regarding the changes that have been preserved in each (Equation 3.6). Then, it defines the net size over time for a change as the initial net size of a change adjusted by the discards over time for that change (Equation 3.7). Finally, it adopts two pre-existing models and tools to define and measure the changes in the number of potential bugs and the average method complexity.

Chapter 4

Auxiliary Study

4.1 Introduction

Teams can increase their efficiency by completing their products more quickly, or at a higher quality, or both (Chapter 1). Within the realm of Software Engineering, they can execute a Software Process Model to structure their activities so that they can develop their products more quickly or at a higher level of quality (Chapter 1). Test-With Development (TWD) is a technique that they can practice to potentially increase their speed and their product's quality (Chapters 1 and 2). But, because of the mixed results and some of the limitations of previous related studies (Chapter 2), we were motivated to study the effects of TWD. In preparation for designing and performing a study, we defined a model to support the direct measurement of the degree of TWD and some attributes of team speed and product quality (Chapter 3). However, prior to using that model in our main study design and execution, we calibrated it with an auxiliary study so that it would be useful for studying ongoing projects.

In the following sections, we use the recommendations and guidance from several authors, to communicate our study design (Pinsonneault & Kraemer 1993, Kitchenham et al. 1995, Basili et al. 1999, Lethbridge et al. 2005, Runeson & Host 2009). First, we describe our design; then we present the results.

4.2 Study Design

4.2.1 Motivation

Within our software change model (Chapter 3), the changes that a developer makes in a particular revision will either be preserved or discarded, when comparing that revision to a future revision. But, since we cannot always compare the particular revision with the *final* revision for a product (because it may not have been created yet), we needed a way to estimate the discards over time for a revision. So, we wanted to find a future revision that was "far enough" away from the original revision to give a reasonably accurate estimate. Otherwise, our model would have been limited to studying projects that are complete. Therefore, we were motivated to design and execute an auxiliary study to identify how far into the future we needed to "look," to get a reasonable estimate of the discards over time for a particular revision.

4.2.2 Plan

Objective

Our objective for this study was to explore and describe the discard patterns of multiple projects. We wished to explore the discard patterns of individual projects, to see if there was a common pattern among the projects. And, we wished to describe the common discard pattern for all of the projects, if one existed. Our hope was that we could identify and describe a common discard pattern which we could then use to make reasonable estimates of discards over time for other projects.

Research Question

Our research question for this study was: *Is there a time that is "far enough" beyond a product change that will allow us to make a reasonable estimate regarding the discards over time for that product change*?

Frame of Reference

Our frame of reference for this study was our software change model (Chapter 3). In particular, we were focused on the discards over time aspects of this model.

Subject Definition and Selection Strategy

Although study subjects should be selected intentionally (Runeson & Host 2009), we selected ours based on availability, as do many studies (Benbasat et al. 1987) and experiments (Sjoberg et al. 2005). Because open-source projects are publicly available and many have been in development for several or more years, we selected some of these products as our subjects.

Data Collection Methods

Generally, we planned to perform a quantitative study by representing the discards of product changes over time with numbers (Fenton & Pfleeger 1997). Specifically, we planned to compare each product revision with future product revisions so that we could determine discard patterns over time.

4.2.3 Protocol

Data Collection Procedures

First, we selected and reconstructed the revisions for six projects (Table 4.1). Five of these projects are Java open-source projects and one was a Java industry project to which we had access. We reconstructed the revisions for these projects, by identifying the revisions that developers had made, and then assembling, building, and storing each revision that could be built (without compilation errors). We describe this procedure in more detail in Appendix C, rather than here, because we also used it to reconstruct the revisions for our main study.

Second, we identified the comparisons that we needed to make to describe the changes that had been discarded over time. Per our software change model,

Project	*Description*
ehcache	A distributed cache toolset
hibernate	An object-relational mapping toolset
industry	A BPM toolset
tomcat	A Web server providing servlet and JSP technology
xerces	An XML parser toolset
xstream	An object-XML mapping toolset

Table 4.1: Auxiliary Study Subject Projects

determining that for a particular revision, or change, involves comparisons with that revision, its previous revision, and a future revision. Thus, to describe the discards of a particular change, over time, those comparisons would need to be made with *each* future revision.

However, because we did not have the computational resources to perform exhaustive comparisons, we needed to identify a reasonable set of "comparison samples." Real-life projects often have tens or hundreds of revisions per week, resulting in an exponential growth in the number of comparisons required, if all permutations are considered. So, we identified a set of comparison samples with these rules: (1) we treated all of the changes for a month as a "monthly change set"; (2) we sampled the discards for each revision within the monthly change set; and (3) outside of the monthly change set, we sampled the discards on a weekly basis, as far as we could into the future. We decided to sample exhaustively within a monthly change set to identify short-term patterns. We decided to sample on a weekly basis, outside of the monthly change set, due to resource limitations.

For example, for a revision created on the 7th of a month, we calculated the discards for that revision, as compared to every future revision created in that same month. But, we only calculated its discards, as compared to revisions created after the end of that month, on a weekly basis.

Third, after reconstructing the revisions and identifying the comparisons that we needed to make, we made the comparisons and calculated the discard over time percentage values for each revision. We compared the identified revisions, utilising a custom toolset which leverages the Byte Code Engineering

Library (http://jakarta.apache.org/bcel). Then, we calculated the discard percentages according to our software change model. Thus, with each comparison and calculation we gathered a sample: the percent that had been discarded for a particular change, over a certain duration.

Comparison Steps

We made a comparison by defining it in an XML specification and then processing it with our custom toolset, jeanda (http://jeanda.tigris.org), which generated an XML comparison result. In an XML specification, we defined the sources of the two sets of Java classes to be compared (Listing 4.1). Then, we directed the custom toolset to make the class comparison(s) with a command (Listing 4.2); the toolset can compare individual classes or sets of classes. Finally, we collected and stored the XML result(s) of the comparison; the individual results look like Listing 4.3 and their summary looks like Listing 4.4.

```
<comparison>
  <jar−source−1 filename="/some/directory/revision−1.jar" />
  <jar−source−2 filename="/some/directory/revision−2.jar" />
  <output−directory dirname="/some/other/directory/1_2" />
</comparison>
```

Listing 4.1: Comparison Specification

```
java
  −cp jeanda.jar
  org.tigris.jeanda.metrics.cmd.JarComparator
  <comparison specification file path>
```

Listing 4.2: Comparison Command

```
<class−comparison>
  <type>CHANGED</type>
  <oldSourcePath>/some/directory/revision−1.jar</oldSourcePath>
  <oldProductKey class=”string”>Amount.class</oldProductKey>
  <newSourcePath>/some/directory/revision−2.jar</newSourcePath>
  <newProductKey class=”string”>Amount.class</newProductKey>
  <methodBytesAdded>74</methodBytesAdded>
  <methodBytesEqual>20</methodBytesEqual>
  <methodBytesDeleted>27</methodBytesDeleted>
  <methodInstructionsAdded>42</methodInstructionsAdded>
  <methodInstructionsEqual>12</methodInstructionsEqual>
  <methodInstructionsDeleted>17</methodInstructionsDeleted>
  <methodsAdded>4</methodsAdded>
  <methodsChanged>1</methodsChanged>
  <methodsEqual>2</methodsEqual>
  <methodsDeleted>1</methodsDeleted>
</class−comparison>
```

Listing 4.3: Comparison Output

```
<class−comparison−sum>
  <methodBytesAdded>21888.0</methodBytesAdded>
  <methodBytesDeleted>4012.0</methodBytesDeleted>
  <methodInstructionsAdded>10829.0</methodInstructionsAdded>
  <methodInstructionsDeleted>1866.0</methodInstructionsDeleted>
  <methodsAdded>353.0</methodsAdded>
  <methodsChanged>105.0</methodsChanged>
  <methodsDeleted>34.0</methodsDeleted>
</class−comparison−sum>
```

Listing 4.4: Comparison Summary

Calculation Steps

We calculated the discard percentage, for revision j, which immediately follows revision i, by future revision N, by making three comparisons and then by making three calculations. The three comparisons correspond to the deltas identified in Equation 3.5. The three calculations use the results of those comparisons to derive the discards of revision j, by revision N, and ultimately the discard percentage. We use three revisions from the ant project to illustrate our comparisons and our calculations; revision $i = 268509$ was created on 2001–01–23, revision $j = 268517$ was created on 2001–01–24, and revision $N = 269221$ was created on 2001–06–24. Revision N was created 150 days after revision j.

First, we compared build i and build j to establish the initial changes made in revision j. This comparison corresponds to Δ_{ij} in our software change model. Listing 4.5 shows the results of this comparison for our example.

```
<class−comparison−sum>
  <methodBytesAdded>134.0</methodBytesAdded>
  <methodBytesDeleted>229.0</methodBytesDeleted>
  <methodInstructionsAdded>63.0</methodInstructionsAdded>
  <methodInstructionsDeleted>118.0</methodInstructionsDeleted>
  <methodsAdded>1.0</methodsAdded>
  <methodsChanged>4.0</methodsChanged>
  <methodsDeleted>4.0</methodsDeleted>
</class−comparison−sum>
```

Listing 4.5: Comparison Summary Δ_{ij}

Second, we compared build j and build N to establish the persistent changes made between revision j and revision N. This comparison corresponds to Δ_{jN} in our software change model. Listing 4.6 shows the results of this comparison for our example.

```
<class−comparison−sum>
  <methodBytesAdded>47815.0</methodBytesAdded>
  <methodBytesDeleted>10995.0</methodBytesDeleted>
  <methodInstructionsAdded>23086.0</methodInstructionsAdded>
  <methodInstructionsDeleted>5102.0</methodInstructionsDeleted>
  <methodsAdded>756.0</methodsAdded>
  <methodsChanged>221.0</methodsChanged>
  <methodsDeleted>98.0</methodsDeleted>
</class−comparison−sum>
```

Listing 4.6: Comparison Summary Δ_{jN}

Third, we compared build i and build N to establish the persistent changes made between revision i and revision N. This comparison corresponds to Δ_{iN} in our software change model. Listing 4.7 shows the results of this comparison for our example.

```
<class−comparison−sum>
  <methodBytesAdded>47826.0</methodBytesAdded>
  <methodBytesDeleted>11101.0</methodBytesDeleted>
  <methodInstructionsAdded>23093.0</methodInstructionsAdded>
  <methodInstructionsDeleted>5164.0</methodInstructionsDeleted>
  <methodsAdded>757.0</methodsAdded>
  <methodsChanged>222.0</methodsChanged>
  <methodsDeleted>102.0</methodsDeleted>
</class−comparison−sum>
```

Listing 4.7: Comparison Summary Δ_{iN}

After we made the comparisons, we calculated the discard percentage by subtracting to get the persistent changes resulting from revision j, dividing to determine the persisting percentage, relative to the initial changes, and then subtracting from unity because the remaining percentage was discarded.

First, we subtracted the comparison results corresponding to Δ_{jN} from those corresponding to Δ_{iN}, to calculate the changes that had persisted from revision j to revision N. The result of this subtraction corresponds to ΔP_{ij} in our software change model. Listing 4.8 shows this result for our example.

```
<persisted−changes>
  <methodBytesAdded>11.0</methodBytesAdded>
  <methodBytesDeleted>106.0</methodBytesDeleted>
  <methodInstructionsAdded>7.0</methodInstructionsAdded>
  <methodInstructionsDeleted>62.0</methodInstructionsDeleted>
  <methodsAdded>1.0</methodsAdded>
  <methodsChanged>1.0</methodsChanged>
  <methodsDeleted>4.0</methodsDeleted>
</persisted−changes>
```

Listing 4.8: Persisted Summary ΔP_{ij}

Second, we divided the relevant portions of the persisted changes results by the original results corresponding to Δ_{ij} (Listing 4.5), to calculate the percentage of the persisted changes. As defined in the software change model, we summed the additions, modifications, and deletions and then we divided the persisted changes values by the original values. Listing 4.9 shows the result of this division for our example, augmented with the steps of the calculation.

```
<persisted−percentages>
  <methodBytes>(11+106)/(134+229)=0.32</methodBytes>
  <methodInstructions>(7+62)/(63+118)=0.38</methodInstructions>
  <methods>(1+1+4)/(1+4+4)=0.66</methods>
</persisted−percentages>
```

Listing 4.9: Persisted Percentages

Third, we subtracted the percentages of the persisted changes from unity, to

calculate the discarded changes. Listing 4.10 shows the result of this subtraction for our example.

```
<discarded−percentages>
  <methodBytes>1.00−0.32=0.68</methodBytes>
  <methodInstructions>1.00−0.38=0.62</methodInstructions>
  <methods>1.00−0.66=0.34</methods>
</discarded−percentages>
```

Listing 4.10: Discarded Percentages

As a result of these comparisons and calculations, we arrived at our discard percentage. And, because we knew the time over which the team had discarded the changes, we had our discards over time sample. For our example, 62% of the method instruction changes were discarded after 150 days. By repeating these comparisons and calculations for other revision pairs, we were able to model the discard patterns for each of the projects.

4.3 Study Results

We present our results below from the preserved changes perspective, as opposed to the discarded changes perspective, since we feel it is a slightly more natural way to view the data. Preserved changes are complements to discarded changes. If 25% of a revision's changes have been discarded at a certain point in time, then 75% of its changes have been preserved.

Figure 4.1 presents the absolute preserved change measurements for the ehcache project, according to the day of the year for which they were measured. For example, the measurements for the 2006–06 monthly change set began around the end of May, which is approximately day 151. The rest of the figures present relative preserved change measurements, where the absolute curves have been shifted to a common origin—this shifting is intended to make it easier to visualise the patterns in a curve group. Finally, these curves present

the preserved compiled instructions for each of the projects.

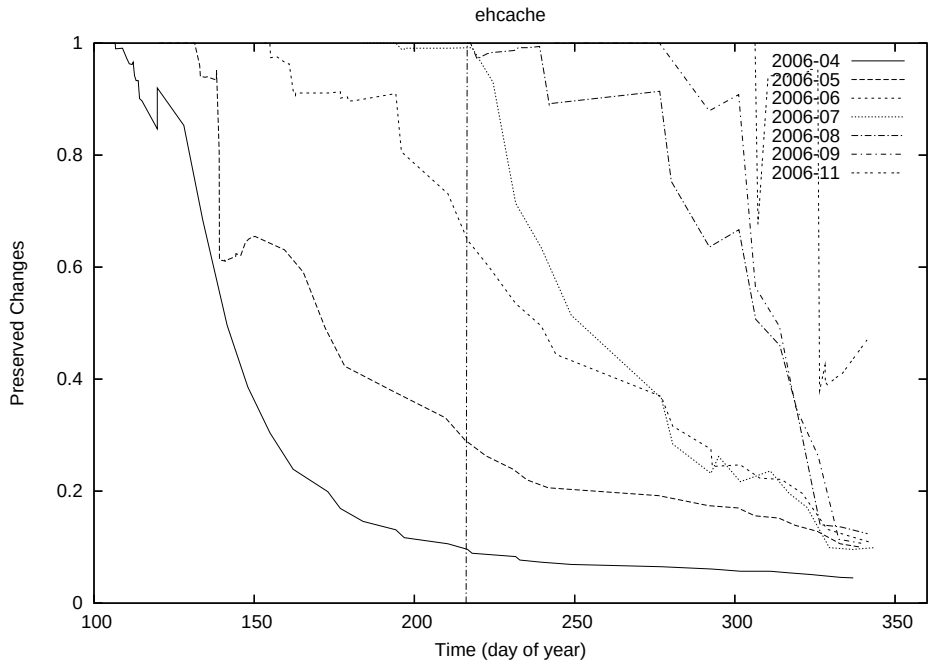


Figure 4.1: ehcache—Preserved Changes over Time (Absolute)

4.3.1 Analysis

Based on our preserved/discarded changes over time measurements, we observed three patterns across certain periods of time: stabilisation, decay, and constant. During the stabilisation period, changes were discarded in an erratic pattern, if there was a pattern at all. During the decay period, changes were discarded rapidly at first and then slowly toward the end. Finally, during the constant period, few of the changes that had been preserved were discarded.

Stabilisation Period

Our first observation was that there appeared to be a "stabilisation period" for the preserved changes of a monthly change set. For all of the projects we measured, this period seemed to be about 40–50 days and was characterised by "erratic" or random fluctuations.

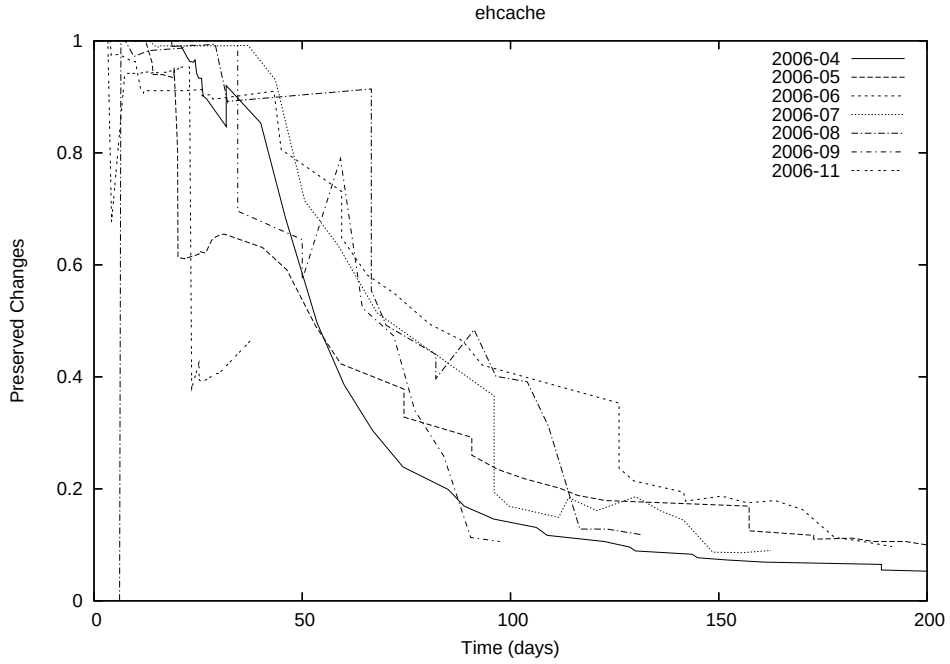


Figure 4.2: ehcache—Preserved Changes over Time (Relative)

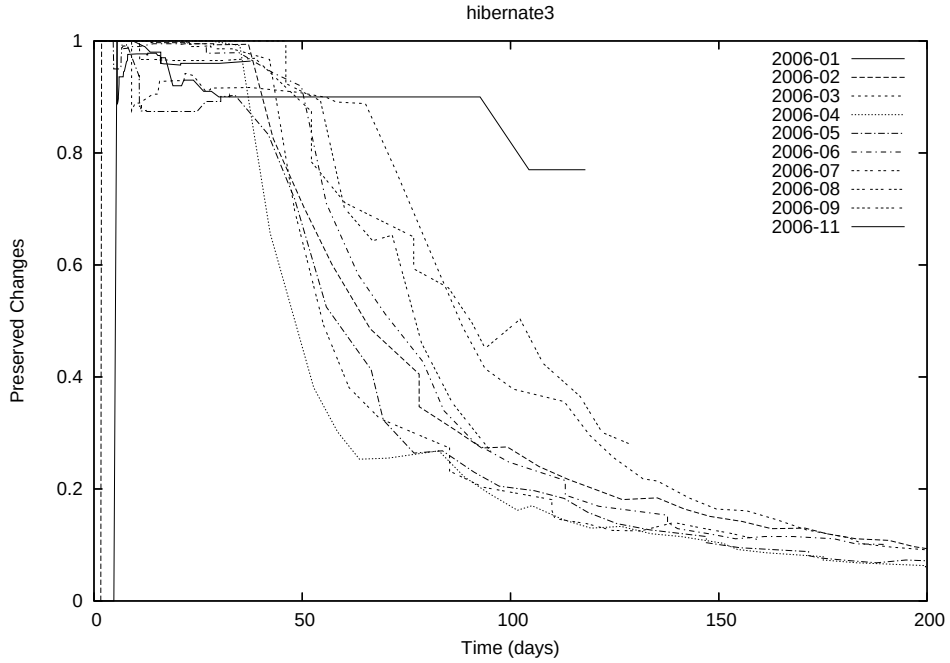


Figure 4.3: hibernate3—Preserved Changes over Time (Relative)

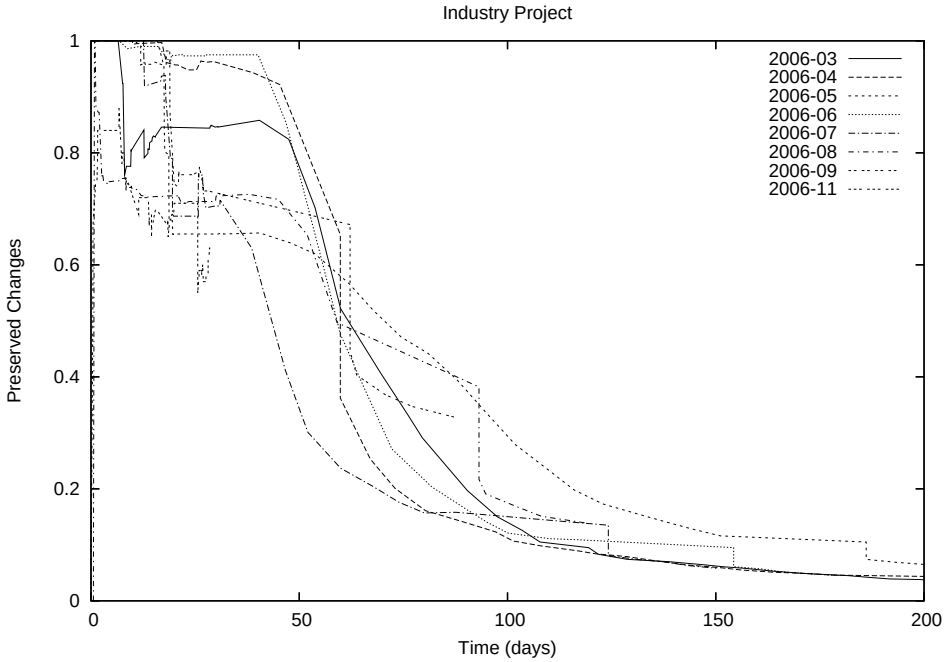


Figure 4.4: industry—Preserved Changes over Time (Relative)

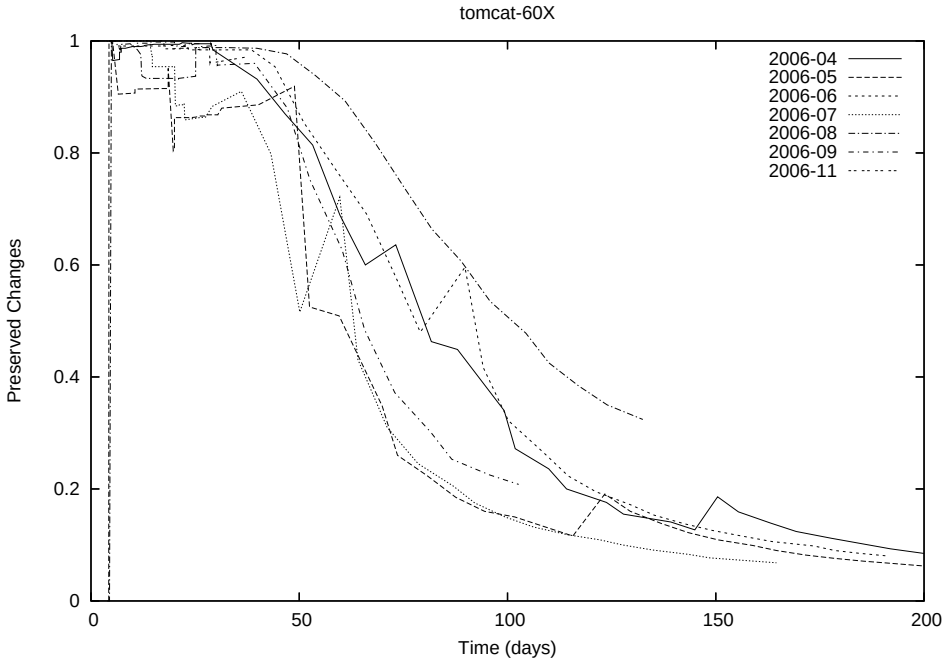


Figure 4.5: tomcat-60X—Preserved Changes over Time (Relative)

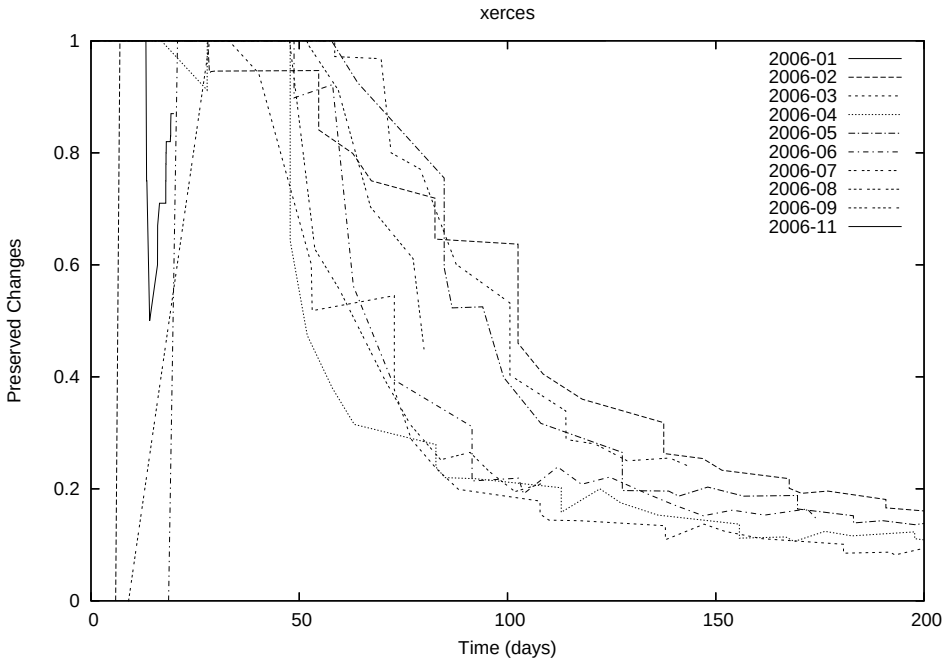


Figure 4.6: xerces—Preserved Changes over Time (Relative)

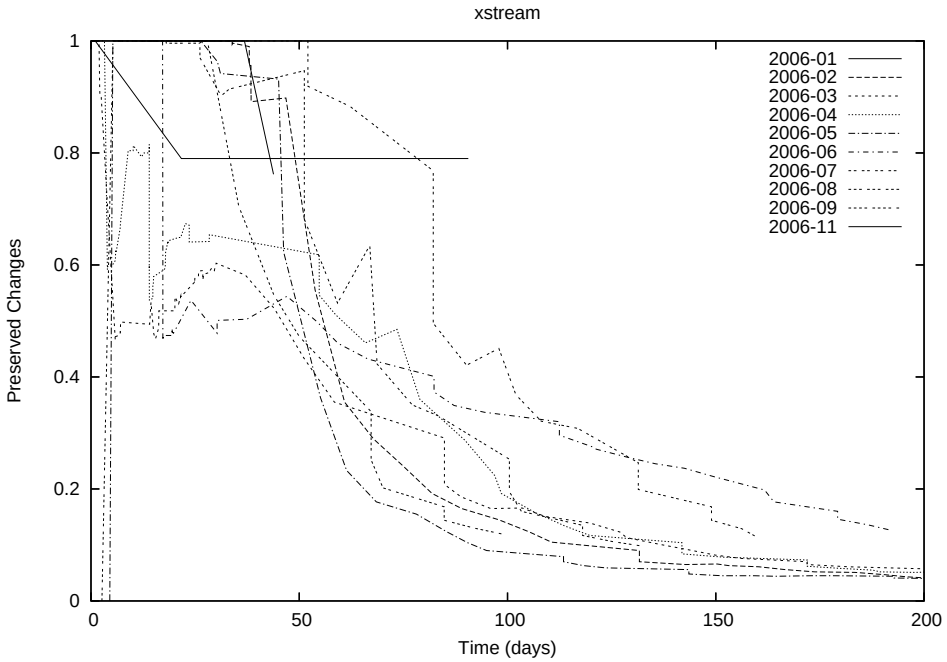


Figure 4.7: xstream—Preserved Changes over Time (Relative)

At first, this period was a little confusing. Since we had sampled all of the revisions within a monthly change set, we expected this period to be characterised by a smooth decay from 1. We did not expect to see increases in preserved changes.

However, upon closer inspection, it made more sense. During this period, a "critical mass" of changes is built up. Since our starting point is arbitrary (the start of the month), changes can come at any time and in any size. If a relatively small number of discarded changes is followed by a larger number of preserved changes, then we see a local increase in preserved changes—the larger changes overpower the smaller ones.

Decay Period

Our second observation was that there was a "decay period." For all of the projects we measured, this period seemed to start about 40–50 days and end about 150–200 days. During this period, the preserved changes decayed from between about 75–95% to about 10–20%.

We also noted that the decay was not linear. That is, the preserved changes do not drop off at a constant rate. Rather, their decay is exponential—most of the decay happens rather quickly. This would seem to indicate that changes which will eventually be discarded, will more likely be discarded sooner rather than later.

Finally, we noted that the shape of the decay curve is not identical for each of the projects. Since each project has many "context variables" (Basili et al. 1986), and overall differences, this was not terribly surprising.

Constant Period

Our third observation was that there was a "constant period." For all of the projects we measured, this period seemed to start at about 150–200 days. During this period, the preserved changes approach a constant (somewhere between 10% and 20%).

When the preserved changes approach a constant, for the set of changes in question, pretty much all of the discarding has taken place and the remaining changes are preserved. That is, the resilient changes remain within the product into this constant period—the rest are discarded during either the stabilisation period or the decay period.

4.3.2 Discussion

As a result of our analysis of the study results, we decided to select 150 days as the answer to our research question; that is, we decided that we would assume that 150 days in the future was "far enough" to accurately estimate the discards over time for a product change. Although all of the curve families were somewhat flat by 100 days, and nearly flat by 200 days, we decided to select 150 days because it was in the middle of the two. While that decision may seem arbitrary, our reasoning was as follows:

We reasoned that if we selected 100 days and assumed that it was "far enough" in the future, then we would both potentially reap a benefit and be subject to a detriment, both as compared to selecting and assuming 150 days. We would potentially reap a benefit because more data, and more up to date data, could be mined for studied projects; rather than requiring that 150 days of future revisions be in place to measure the discards over time, only 100 days of future revisions would be necessary. However, we would potentially be subject to a detriment as well, because comparing with revisions that are not "far enough" in the future could threaten the validity of the results.

Similarly, we reasoned that if we selected 200 days and assumed that it was "far enough" in the future, then we would both potentially reap a benefit and be subject to a detriment, again as compared to selecting and assuming 150 days. By choosing "too far" into the future, we would be more likely to have valid data but we would be able to mine less of it.

Therefore, to balance the potential benefits and detriments related to data quality and data quantity, we selected 150 days as "far enough" in the future.

4.4 Summary

To estimate the proportion of a developer's change that has been discarded over time, we needed to know how far into the "future" we had to look. In theory, as described in Chapter 3, we could compare a product change with the final revision of that product to measure the relevant discards over time. However, that is not practical for all products because some of them are still actively developed. Therefore, we performed an auxiliary study to see if there was an appropriate "rule of thumb" to provide for a reasonable estimate.

Once the preserved change curves from our study results were placed at a common origin, we identified three periods common to each project: stabilisation, decay, and constant. After about 150 days, most of the curves had dropped from 100% preserved to around 20% preserved. That means that only about 20% of the changes were persisted past 150 days. Said another way, about 80% of the discards happen by then. For these projects, we observed the "80/20 rule," also known as the Pareto principle (Juran 1988), at the 150 day mark.

Hence, we identified a practical rule to help us estimate the discards over time for a particular developer's change. If we compare a build with another build at least 150 days into the future (assumed to be in or near the constant period), then our estimate of preserved and discarded changes for that build will be more accurate than if we compare it to a more recent build (in the stabilisation or decay period). We used the result of this auxiliary study as a guideline for measuring and estimating the discards over time for product changes in our main study.

Chapter 5

Study Design

5.1 Introduction

Teams can increase their efficiency by completing their products more quickly, or at a higher quality, or both (Chapter 1). Within the realm of Software Engineering, they can execute a Software Process Model to structure their activities so that they can develop their products more quickly or at a higher level of quality (Chapter 1). Test-With Development (TWD) is a technique that they can practice to potentially increase their speed and their product's quality (Chapters 1 and 2). But, because of the mixed results and some of the limitations of previous related studies (Chapter 2), we were motivated to study the effects of TWD. Once we had defined a software change model (Chapter 3) and calibrated it (Chapter 4), we were prepared to design our main study.

In the following sections, we again use the recommendations and guidance from several authors, to communicate our study design (Pinsonneault & Kraemer 1993, Kitchenham et al. 1995, Basili et al. 1999, Lethbridge et al. 2005, Runeson & Host 2009). Each section describes a particular perspective of our study design: (1) its Motivation; (2) its Plan; and (3) its Protocol.

5.2 Motivation

In addition to being motivated to study the potential benefits of TWD (Chapter 1), and being motivated to improve upon some of the limitations of previous TFD studies (Chapter 2), multiple precedents of mining information from repositories motivated us to design and perform a similar type of study. Card et al. evaluated software engineering technologies using empirical and statistical methods (1987). By identifying and matching past projects, they were able to derive productivity and reliability values from historical project data. Then, they compared the matched projects to evaluate the effects of particular technologies. And, Cook et al. proposed that exploratory empirical studies could be both useful and cost-effective (1998). They suggested that organisations can correlate process "factors" and their effects, by using data from existing process and product repositories, And, since 2004, a community at the International Conference on Software Engineering (ICSE) has been working to evolve the practice of "mining software repositories" (Hassan et al. 2005, Diehl et al. 2005, 2006, Gall et al. 2007, Lanza et al. 2008). A basis for their work is that "software repositories contain a wealth of valuable information for empirical studies in software engineering" (Hassan et al. 2005).

5.3 Study Plan

5.3.1 Objective

Given our motivation in general, and our motivation to study historical data in particular, our objective was both exploratory and descriptive (Robson 2002, Runeson & Host 2009)—we could not manipulate the projects (and therefore not explain by proving causal relationships) and we were not trying to improve TFD or TWD. We wished to explore the relationships between the degree of TWD for product changes and the resulting effects on some attributes related to team speed and product quality. We also wished to describe any such relationships,

if they existed. Additionally, assuming that our study might identify significant relationships, our other objective was to provide a basis for further research.

5.3.2 Research Question

Our research question, as introduced in Chapter 1 and specified in Chapter 3 is: *What are the relationships between practising TWD, to varying degrees, and two attributes related to team speed (initial net size and discards over time) and two attributes related to product quality (potential bugs and average method complexity), when those practising it are real-life software development teams, developing their product over the course of many years*?

Given the existing TFD research results, which have been mixed and are described in Chapter 2, and the lack of any previous TWD results, our initial hypotheses were based solely on our experience in the software industry. We expected that:

- *H1*: Increasing the degree of TWD would correspond with a positive effect on the attributes related to team speed, because the increased amount of automated tests would give the developers the confidence necessary to make bigger changes over time.
- *H2*: Increasing the degree of TWD would correspond with a positive effect on the attributes related to product quality, because the increased amount of automated tests would give the developers the confidence necessary to increase the internal quality of their code.

5.3.3 Frame of Reference

Although using theories to develop a research direction is not well established in software engineering (Hannay et al. 2007), we developed a theory for our study. The frame of reference, or theory, for our study is our software change model (Chapter 3)—in this precise set-theory model, we characterise changes to product and automated product test classes, methods, and compiled instructions.

With our model, we had a basis to estimate the degree of TWD for a change as well as for measuring the two attributes related to team speed (initial net size and discards over time). We relied on previously defined theories and tools to measure the two attributes related to product quality (potential bugs and average method complexity).

5.3.4 Subject Definition and Selection Strategy

As illustrated in the Java example in Chapter 3, the subjects of our study are products as they change over time. And, as we did for our auxiliary study (Section 4.2.2), we selected ours based on availability, as do many studies (Benbasat et al. 1987) and experiments (Sjoberg et al. 2005). Because open-source projects are publicly available and many have been in development for several or more years, we selected some of these products as our subjects.

At first, our selection criteria was simple: a Java project with more than two years of development history, that maintained both a product and a corresponding set of automated product tests. However, due to our tool set, and problems building projects using other build systems (primarily maven), our criteria became more refined: we also required that the project have its own ant (http://ant.apache.org) build file. We selected several subject projects from the Apache family of open-source projects (http://www.apache.org), and also selected a couple of projects that we were already familiar with, because of our work in industry.

Additionally, we chose to study two of the subject projects from our auxiliary study in our main study. We did so for practical reasons, even though this is not generally considered a best-practice, because of potential biases or threats that may result (Pesola 2004). Since we already had several years of the product revisions built, and because the manual overhead for building revisions for projects was significant, we decided to reuse the already built product revisions from two of these products and to build more product revisions for them rather than building all of the product revisions for two new projects. In hindsight,

we do not think either study was biased or invalidated by this decision—we would have arrived at the same answer from our auxiliary project without the overlapping projects and the results from the overlapping projects in our main study were consistent with the other non-overlapping projects.

5.3.5 Data Collection Methods

Because we were motivated to study multiple products and thousands of product revisions, we planned on utilising partial or full automation. With automation support, we selected all of the self-contained commits—those commits that contained changes to product source code (and optionally changes to automated test source code) and which could be compiled.

Generally, we planned to perform a quantitative study by representing the changes to the attributes of the products with numbers (Fenton & Pfleeger 1997). Specifically, we performed a static analysis of each product revision and of the appropriate product revision pairs to estimate the degree of TWD that its developer practised, as well as its initial net size, discards over time, change in the number of potential bugs, and change in average method complexity.

5.4 Study Protocol

Before we could measure product version changes, we needed to build a representation of each change that we were going to study. We also needed to be able to analyse the results.

For these needs, we utilised software tools. We were able to use several publicly available tools for some of the tasks. But, for some others, we needed to develop custom tools. In the following sections, we describe the tools and the protocol we used to build, measure, and analyse the product revisions of the products that we studied.

5.4.1 Data Collection Procedures

In a typical open-source project, multiple geographically distributed developers collaborate to build and evolve a software product (Raymond 1999). Usually, the team of developers shares and coordinates their work via a network and a configuration management tool, such as the Concurrent Versioning System (CVS) (http://www.cvshome.org) or its successor, the Subversion system (SVN) (http://subversion.tigris.org). Once a particular developer has added or changed a feature, and optionally added or changed automated tests, he or she commits their changes so that the rest of the team and any continuous integration servers have access to them. The result of the commit is a new revision in the team's repository.

To build the revisions of a product, we used a specific version of the Sun JDK, a specific release version of the ant product, specific versions of any third party dependencies, and a combination of custom shell scripts and the ant build scripts provided by the teams themselves. First, we identified all of the revisions that included Java files in the product change set. Then, we used scripts to update our working copy from the team's repository and to clean and build a particular revision of the product and the product tests. We executed this sequence for each pertinent revision and stored the resulting artifacts, mapped by revision number. We describe this procedure elsewhere, in Appendix C, because we used it in our auxiliary study as well.

Because the team's product and their build scripts evolved during the lifetime of their project, we had to keep our shell scripts in sync with them. When a build failed, we had to analyse why it had failed and then change our custom shell script to incorporate the correct versions of the JDK, ant, and the thirdparty dependencies, and to select the team's ant tasks which were intended to compile their product and their automated product tests. Thus, unfortunately, we could only partially automate this process.

Once we had rebuilt the product revisions and automated product test revisions for a product, we were able to use our custom and thirdparty tools to

measure and estimate values for the degree of TWD, the initial net size, the discards over time, and the changes to the number of potential bugs and the average method complexity. We describe our procedure for doing so in the following subsections.

Independent Variable—Degree of Test-With Development

As illustrated in our Java example, and defined in our software change model (both in Chapter 3), we assume that the degree to which a developer practises TWD, for a particular change set, will be reflected by the relationships between the automated product tests and the product methods within that change set.

To measure the degree of TWD for a change, we developed and used a custom tool called jeanda (Java Efficiency and Agility) (http://jeanda.tigris.org). Within it, we model and compare revisions of compiled classes, methods, and instructions of Java code, leveraging the Byte-Code Engineering Library (BCEL) (http://jakarta.apache.org/bcel). First, it determines the compiled methods that have changed within a product revision pair. Second, it determines which of those methods were referenced by any of the compiled instructions within the automated product test methods, directly or indirectly. Then, it determines the fraction of the changed product methods that were referenced, and presumed exercised, by tests. We describe how we used this tool, by example, in Appendix D.

If all of the changed product methods had automated test methods related to them, then we categorised the change set as "all." Likewise, if none of the changed product methods had automated test methods related to them, then we categorised the change set as "none." However, if some of the changed product methods had automated test methods related to them, then we categorised the change set as "some." These categories provided the basis for our comparative analysis in the present study; our future work section (Chapter 7) describes an alternative and potentially more accurate approach, using dynamic analysis rather than static analysis.

Zaidman et al. used an alternative static approach to determine whether or not a team was practising TDD, in addition to analysing the co-evolution of a product and its product tests in more general terms (2008). They inferred a relationship between product classes and product test classes based on a naming convention for the corresponding source files. They also inferred that changes to both of the files implied that the developer practised TDD. However, we wanted a more precise and granular approach and thus chose to probe the relationships between the automated product tests and the product, within the compiled code.

Dependent Variable—Effect on Speed (Initial Net Size)

With the Lines of Code (LOC) measurement, bigger changes per time period have been assumed to represent higher team speed, as compared to smaller changes in the same time period. But, this is not always the case (Boehm et al. 2000). However, in contrast to LOC measurements, our initial net size measurement considers changes to compiled instructions, or bytecode instructions, rather than lines of source code. Additionally, it measures the net size of a change, rather than the gross size of a change.

We chose to consider compiled instruction changes rather than LOC changes because the compiler removes organisational changes and allows us to focus on behavioural changes. That is, organisational changes such as alphabetising methods within a class source file, modifying comments, adding blank lines, and placing multiple statements on one line do not cause bytecode instruction changes when a class is compiled. And, although some more advanced LOC analysis tools might be capable of doing the same thing, we chose to use the compiler because it was simple for us to incorporate it into our study.

Additionally, we measure the net size of a developer's change, rather than the gross size of a developer's change, so that renaming of namespaces, classes, and methods are not counted twice—we also consider these to be organisational changes rather than behavioural changes. We want our measure to recognise

that the simple renaming of a class resulted in X compiled instructions being added to the product and X compiled instructions being deleted from the product, and that the net size of the change was 0 (rather than 2X).

To determine the initial net size of a developer's change set, we also used the jeanda tool. First, it models the classes and methods from before the change. Second, it models the classes and methods from the change. Third, it compares each of the methods to determine which compiled instructions have been added, changed, and deleted. Then, it calculates the net size of the change set, in compiled instructions, by summing the differences for the individual methods and classes. We describe how we used this tool, by example, in Appendix D.

Nonetheless, we realise that the initial net size of a developer's change only represents one attribute of a team's speed; it is missing the time dimension. That is why we also estimated discards over time—we also wanted to measure the changes that persisted because of a developer's change (and presumably continued to add value).

Dependent Variable—Effect on Speed (Discards over Time)

As illustrated in our Java example and defined in our software change model (both in Chapter 3), each product revision introduces two subsets of changes to the product: a subset of changes that will persist within the product at least until a certain time, and a subset that will be discarded before that. Therefore, by measuring the fraction of a change set that is discarded over time, we can adjust the initial net size measurement and determine a net size over time.

For example, consider a product revision which adds two product methods. As well, consider another product revision which adds four product methods. Assume that each of the compiled methods has ten bytecode instructions. If after six months, two product methods have persisted from the first revision and only one product method has persisted from the second, then 0% of the first revision has been discarded over time and 75% of the second revision has been discarded over time. Further, the first revision has a bigger net size over

time (20 bytecode instructions added per 6 months) than the second revision (10 bytecode instructions added per 6 months), even though the second revision has a bigger initial net size (40 bytecode instructions as compared to 20 bytecode instructions).

To determine the fraction of a developer's change set that was discarded, we also used the jeanda tool. First, it compares the changes between the current revision and a future revision which is at least 150 days beyond the next revision (Chapter 4). Second, it compares the changes between the next revision and that same future revision. If more changes were persisted by the current revision as compared to the next revision, both relative to the future revision, then the current revision is credited with providing them. Any difference between what the current revision provided initially and what it provided persistently is the fraction that was discarded. We describe how we used this tool, by example, in our auxiliary study design in Chapter 4.

Dependent Variable—Effect on Quality (Potential Bugs)

Product revisions may introduce or remove actual bugs in the classes and methods that change. However, direct automatic measurement of actual bugs is difficult or impossible. But, if we measure the number of potential bugs before and after a product revision with a tool, we can estimate the effect of the product revision on the number of actual bugs.

For example, consider a potential bug type named "double assignment of field." If the product has five potential bugs of this type before the change, and ten after the change, and the number of other potential bug types has remained the same, then the number of potential bugs has increased by five (+5.0).

To estimate the effect of a developer's change set on the number of bugs in a product revision, we used the findbugs (http://findbugs.sourceforge.net) tool. This tool executes a static analysis of compiled Java code to find potential bugs in three broad categories: correctness, bad practice, and dodgy (Ayewah et al. 2008). First, we isolated the changed product classes from before the

change and executed findbugs against them. Second, we isolated the changed product classes from the change and then similarly executed findbugs against them. Thus, the number of potential bugs increased if the second measurement was bigger than the first; for this study, we did not distinguish between the categories of bugs.

Dependent Variable—Effect on Quality (Average Method Complexity)

Similar to the change in the number of potential bugs, product revisions may increase or decrease the average method complexity of the classes that change. Therefore, if we measure the average method complexity before and after a product revision, we can determine the effect of the product revision on the average method complexity.

For example, consider the cyclomatic complexity of a method. If the complexity is ten before the change, and five after the change, and none of the other methods has changed, then the method complexity has decreased by five (-5.0). And, if the class contains two other methods, each with a method complexity of 10.0, then the average method complexity for the class has decreased from 10.0 (30.0 / 3) to 8.33 (25.0 / 3), which is a change of -1.67.

To determine the effect of a developer's product revision on the complexity of the product, we used the cyvis (http://cyvis.sourceforge.net) tool. This tool executes a static analysis to measure the cyclomatic complexity (McCabe 1976) of methods of compiled Java code. First, we isolated the changed classes from before the change and then measured the complexities of their methods to calculate the old average method complexity. Second, we isolated the changed classes from the change and then measured their method complexities to calculate the new average method complexity. Thus, the complexity increased if the new average method complexity was bigger than the old average method complexity.

5.4.2 Analysis Procedures

Before analysing our data, we had to relate the data from our independent variable to each of our dependent variables. We related the data by writing custom scripts to select and join the pertinent information from each of the individual result sets. For example, to determine a discard over time measurement for revision 200 of a particular product, we selected the particular row from our degree of TWD data for revision 200, selected the particular row from our discard over time data for revision 200, and joined the pertinent columns from each row.

Once we had the related data, we utilised the R project for Statistical Computing (http://www.r-project.org) to support our analysis of it, in accordance with the preliminary guidelines for empirical research (Kitchenham et al. 2002). First, we used it to provide a descriptive analysis of our data; we used it to create scatterplots, histograms, and boxplots of the data and to identify potential extreme outliers. Second, we used it to provide an inferential analysis of our data; we used it to compare and analyse the differences between the means for the three categories of TWD and their related effects.

Based on the descriptive analyses of our data, we identified the extreme outliers after transforming the data (if necessary) and then applying a thresholding rule. Generally, the data was not skewed but it did exhibit excess kurtosis (usually centred around zero); therefore, when appropriate we transformed the data into more normal distributions with log transformations. Then, we applied a thresholding rule—we considered the outliers to be extreme if they were outside three standard deviations from the mean.

After identifying the extreme outliers, we investigated their causes to convince ourselves that they could be considered outside of "normal," so that we could justify excluding them from our data. We identified several causes for these extreme outliers:

- Major reorganisations of the product (for example, namespace changes).
- Merging two streams of product code (for example, incorporating "sand-

box" code into the trunk stream once it was considered ready).

- Moving from one version of the JDK to another (for example, from 1.3 to 1.4)—this occurred at most two times per project we studied.
- Multiple developments that were "queued up" because of compilation problems (for example, several or many commits were made while the build was broken).
- Removing part of the product (for example, part of the product had been extracted and was now available as a third party library).

Since none of these infrequent activities has any bearing upon the degree of TWD that a developer practises, and since all of the extreme outliers were due to one of these causes, we excluded the extreme outliers.

To compare and analyse the differences between the means for the effects of the three categories of TWD (none, some, all), we used the Welch Two Sample unpaired t-test because the scatterplots and boxplots indicated that the variances were different and the data was not paired. Before doing so, we transformed the discards over time data sets from non-normal distributions to normal distributions, via bootstrapping, so that we could determine the confidence intervals for the differences between their means, in addition to their p-values; the other data sets were already in normal form. For the Welch Two Sample unpaired t-test, we used the 90% confidence level.

5.5 Summary

We were motivated to execute an empirical study to explore and describe the effects of TWD product changes on team speed and product quality. Based on our motivation, we decided to execute a multiple comparative study. Further, we decided to study open-source projects by mining their repositories, because they contain a wealth of freely available information.

Once we had decided to execute our study, we developed our plan for it incrementally as we iterated over what we wanted to achieve, the frame of reference for our study, what we wanted to study, and how we wanted to collect our data (Robson 2002, Runeson & Host 2009). We settled on exploring and describing the degree of TWD practised by a team and the effects of the related changes on specific attributes related to team speed (initial net size and discards over time) and product quality attributes (potential bugs and average method complexity). Then, we evolved our study plan and our study protocol to facilitate collection and analysis of valid data.

Chapter 6

Study Results

6.1 Introduction

Teams can increase their efficiency by completing their products more quickly, or at a higher quality, or both (Chapter 1). Within the realm of Software Engineering, they can execute a Software Process Model to structure their activities so that they can develop their products more quickly or at a higher level of quality (Chapter 1). Test-With Development (TWD) is a technique that they can practice to potentially increase their speed and their product's quality (Chapters 1 and 2). But, because of the mixed results and some of the limitations of previous related studies (Chapter 2), we were motivated to study the effects of TWD. So, we prepared to study TWD's effects by defining a software change model (Chapter 3) and calibrating it (Chapter 4). Once we had designed our study (Chapter 5), we were prepared to execute it, analyse the results, and evaluate their validity.

In this chapter, we summarise and interpret the analysis results, discuss the results per degree of TWD, and then examine the validity of the results. To keep this results section focused (Robson 2002), we have not included the summary results for the individual projects—we present these project summary results in Appendix E. Nor have we included the histogram and boxplot diagrams of

the measurements—we present these project detailed results in Appendix F. Instead, we present lattice plots to summarise all of the project mean values for each of the measures and the confidence intervals for each of the comparisons (except for the net size over time measure, which is derived from the initial net size and the discards over time measures). Additionally, we present the percent differences for some of the comparisons, to highlight significant differences. Finally, we discuss and examine the validity of the results.

6.2 Results Summary

The projects we studied were: (1) ant; (2) cayenne; (3) commons-codec; (4) ehcache; (5) hadoop-core; and (6) xstream. Each of these are open-source projects that have been developed over many years.

6.2.1 Initial Net Size

Figure 6.1 summarises the average initial net size of the changes that developers made, in terms of method instructions, for each project. These results show that when these developers exercised some of their changes with tests, they made bigger changes than when they exercised none of their changes with tests. Similarly, these results show that when these developers exercised all of their changes with tests, they made similarly sized or smaller changes than when they exercised none of their changes with tests.

Figure 6.2 summarises the confidence intervals for the differences between the some-none and the all-none initial net size measurement comparisons. It also indicates the significances of the differences, with shading, according to the vertical colour key on the far right of the figure. The significant confidence intervals confirm that the developer changes had bigger initial net sizes when they exercised some of their changes with tests, as compared to when they exercised none of their changes with tests; typically, the intervals indicate that those changes were between 0 and 100 method instructions bigger. They also

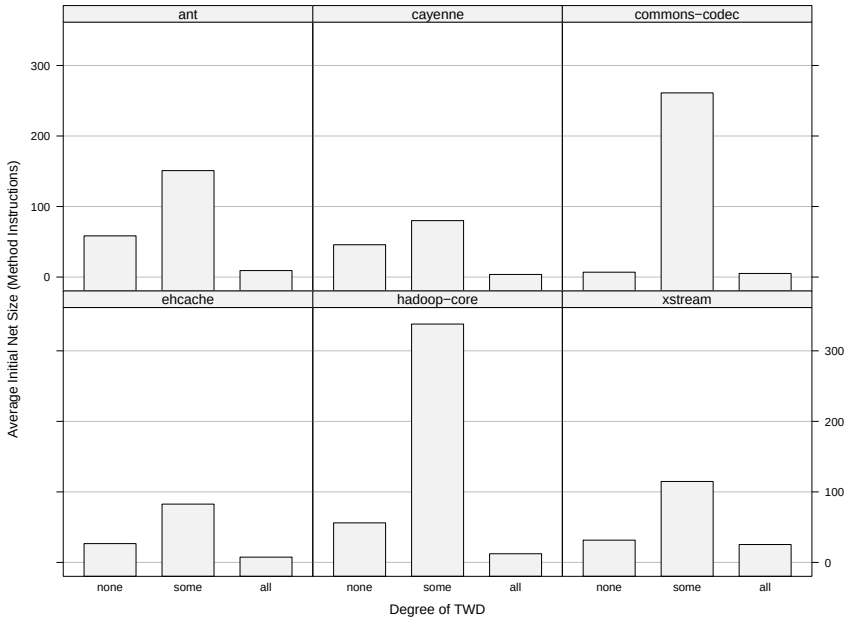


Figure 6.1: Average Initial Net Size

confirm that the developer changes had smaller initial net sizes when they exercised all of their changes with tests, as compared to when they exercised none of their changes with tests; typically, the intervals indicate that those changes were between 0 and 100 method instructions smaller.

Figure 6.3 summarises the percentage differences between the averages for the some-none and the all-none initial net size measurement comparisons, relative to the none measurements. These results show that when these developers exercised some of their changes with tests, on average they made 75–500+% bigger changes than when they exercised none of their changes with tests. Similarly, these results show that when these developers exercised all of their changes with tests, on average they made 25–100% smaller changes than when they exercised none of their changes with tests.

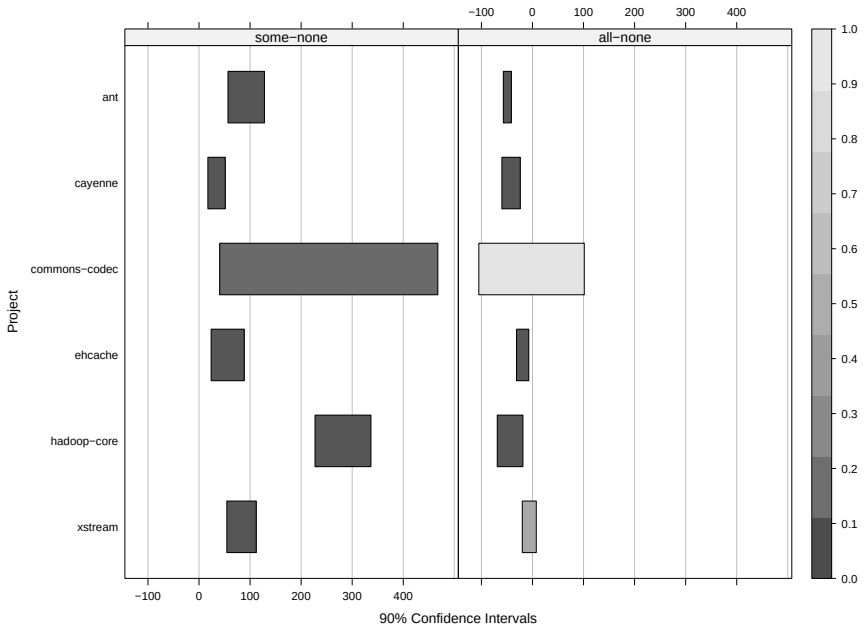


Figure 6.2: Confidence Intervals for Initial Net Size

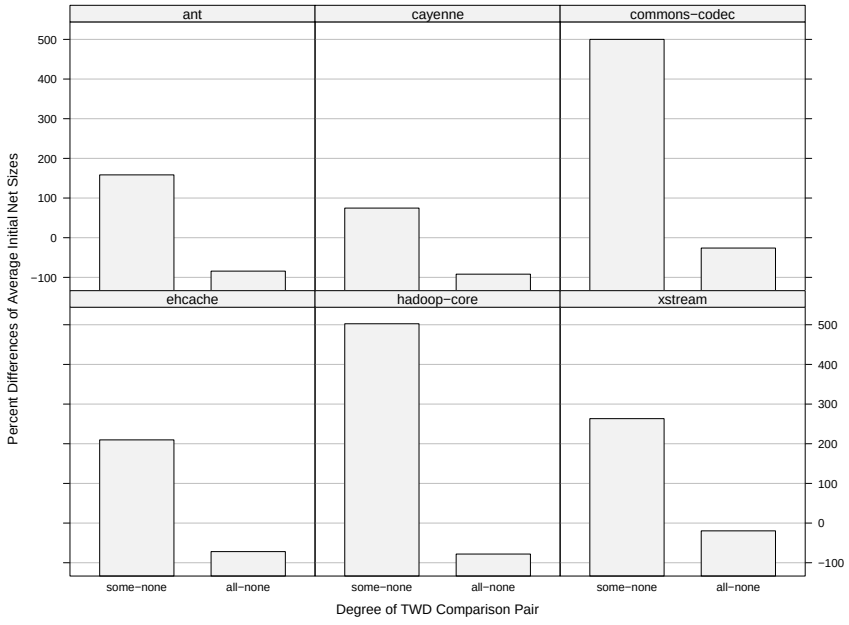


Figure 6.3: Percentage Differences for Average Initial Net Size

6.2.2 Discards Over Time

Figure 6.4 summarises the discards over time for the changes that developers made, in terms of the percentages of the initial changes that were discarded, for each project. These results show that between 0 and 45% of the initial changes were discarded over time. However, there is no pattern that corresponds to the degree of TWD that the developers practised. When these developers exercised some of their changes with tests, sometimes more and sometimes less of their changes were discarded over time, as compared to when they exercised none of their changes with tests. The same is true for when they exercised all of their changes with tests, as compared to when they exercised none of their changes with tests.

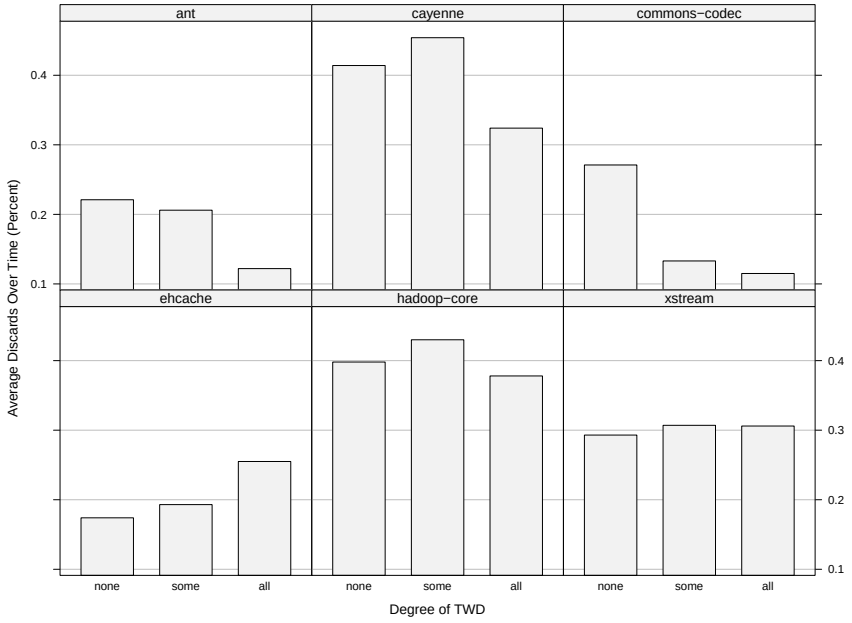


Figure 6.4: Average Discards Over Time

Figure 6.5 summarises the confidence intervals for the differences between the some-none and the all-none discards over time measurement comparisons. It also indicates the significances of the differences, with shading, according to the vertical colour key on the far right of the figure. All but one of the confi-

dence intervals are significant, indicating that there were significant differences between the groups that were compared. However, in both the some-none and all-none comparison groups, there are cases where developers exercised their changes with tests and fewer of their changes were discarded over time (cayenne in the all-none comparison group) and cases where developers exercised their changes with tests and more of their changes were discarded over time (hadoop-core in the some-none comparison group). Additionally, these results show that the differences between the discards over time typically varied between +/-10%, for the projects that we studied.

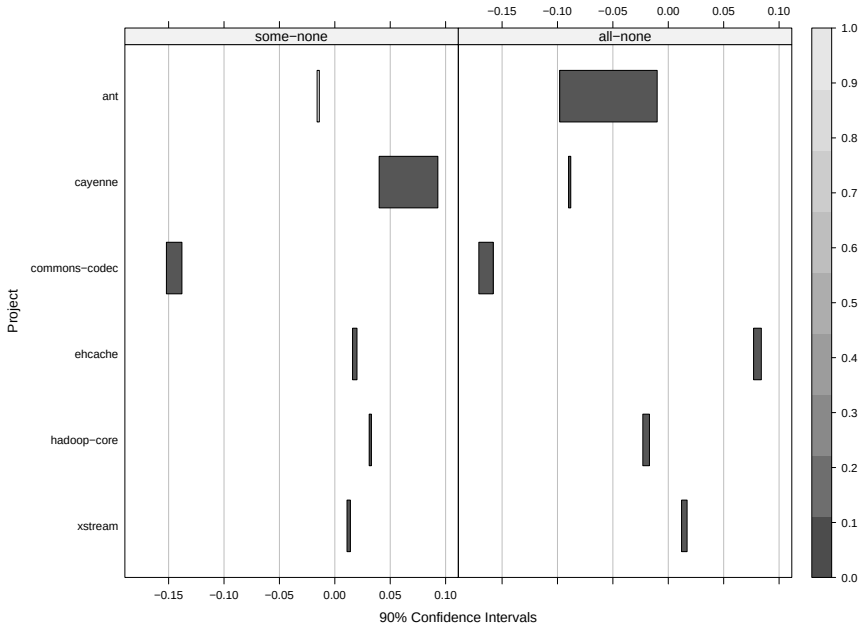


Figure 6.5: Confidence Intervals for Discards Over Time

6.2.3 Net Size Over Time

Figure 6.6 summarises the net size over time of the changes that developers made, derived from the average initial net size and the average discards over time values, for each project. These results show that when these developers exercised some of their changes with tests, they made bigger changes than when

they exercised none of their changes with tests. Similarly, these results show that when these developers exercised all of their changes with tests, they made smaller changes than when they exercised none of their changes with tests. So, even though there was up to a 10% difference with respect to the percentage of changes that were discarded over time, the pattern from the results for the initial net size comparisons remained similar over time.

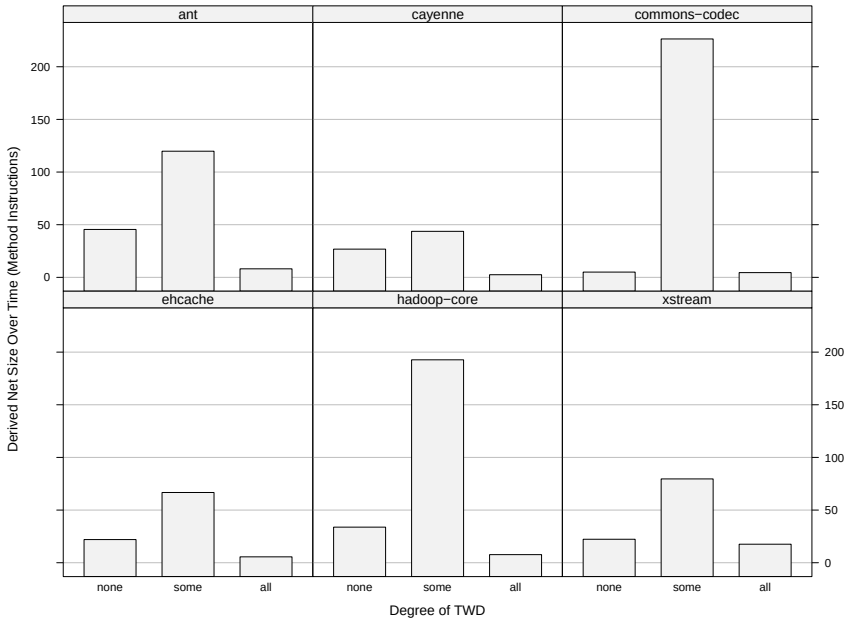


Figure 6.6: Derived Net Size Over Time

Figure 6.7 summarises the percentage differences between the derived values for the some-none and the all-none net size over time measurement comparisons, relative to the none measurements. These results show that when these developers exercised some of their changes with tests, on average they made 50–500+% bigger changes than when they exercised none of their changes with tests. Similarly, these results show that when these developers exercised all of their changes with tests, on average they made 20–100% smaller changes than when they exercised none of their changes with tests.

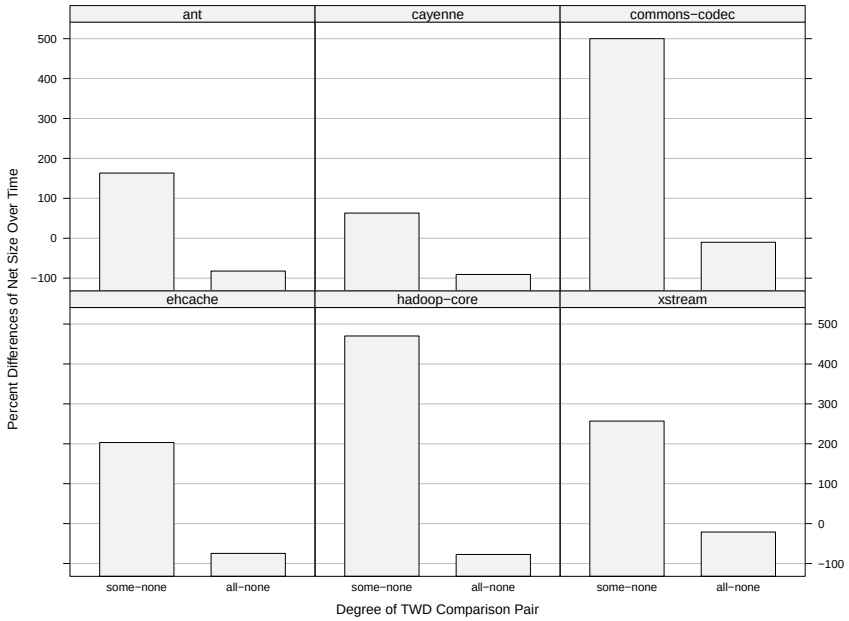


Figure 6.7: Percentage Differences for Net Size Over Time

6.2.4 Potential Bugs

Figure 6.8 summarises the change in the potential bugs for the changes that developers made, for each project. These results show that, on average, all of the types of changes resulted in fewer potential bugs; whether the developers practised TWD or not, there were fewer potential bugs in the affected code after they made their changes. However, these results do not identify a clear pattern where the developers practised TWD, to some degree or fully, and removed more or less potential bugs than when they did not practice it.

Figure 6.9 summarises the confidence intervals for the differences between the some-none and the all-none change in potential bugs measurement comparisons. It also indicates the significances of the differences, with shading, according to the vertical colour key on the far right of the figure. Only two of the confidence intervals are significant. And, most of the confidence intervals straddle 0. Therefore, the comparisons do not indicate more or less potential bugs after a developer has made a change, practising TWD to some degree or

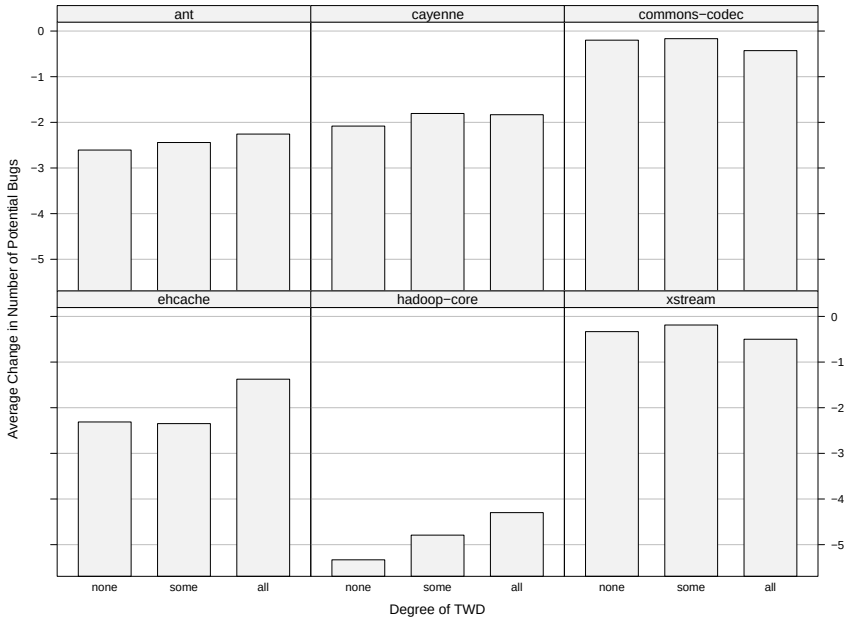


Figure 6.8: Average Change in Potential Bugs

fully, both as compared to not practising TWD at all. But, if we interpret the trend in these intervals, they indicate that practising TWD to some degree or fully when making a change, as compared to not practising it, may result in 0–1 fewer potential bugs being removed by the change.

6.2.5 Average Method Complexity

Figure 6.10 summarises the average change in the average method complexity for the changes that developers made, for each project. These results show that when these developers exercised some of their changes with tests, on average they reduced the average method complexity of the methods that they changed. In contrast, the results also show that these developers, on average, increased the average method complexity of those methods when they exercised none of their changes with tests. Also in contrast, the results show that these developers, on average, sometimes increased and sometimes decreased the average method complexity of the methods that they changed when they exercised all of their

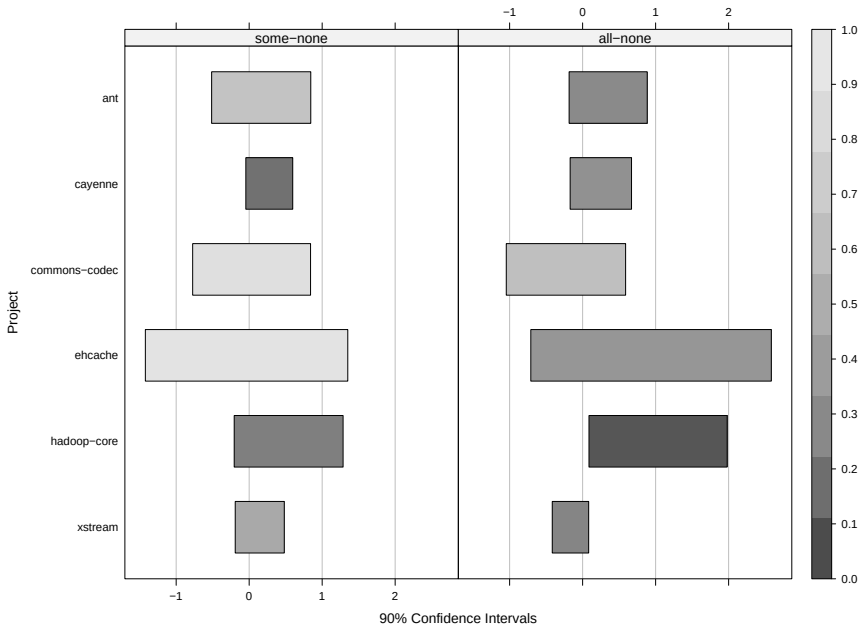


Figure 6.9: Confidence Intervals for Change in Potential Bugs

changes with tests.

Figure 6.11 summarises the confidence intervals for the differences between the some-none and the all-none change in average method complexity measurement comparisons. It also indicates the significances of the differences, with shading, according to the vertical colour key on the far right of the figure. Three significant confidence intervals and three other consistent confidence intervals in the some-none comparison indicate that the developer changes reduced the average method complexity when they exercised some of their changes with tests, as compared to when they exercised none of their changes with tests; typically, the intervals indicate that those changes were between 0 and 0.1 less complex. And, although there are no significant confidence intervals in the all-none comparison, these intervals seem to confirm the neutrality of developers making changes when they practice TWD fully, as compared to not practising it at all; sometimes they increase the average method complexity of the methods they change and sometimes they decrease it.

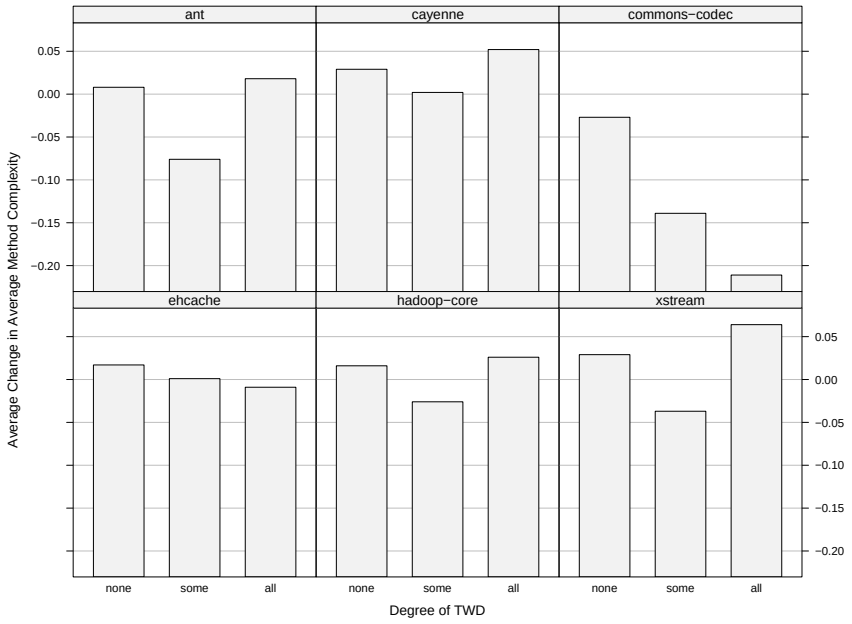


Figure 6.10: Average Change in Average Method Complexity

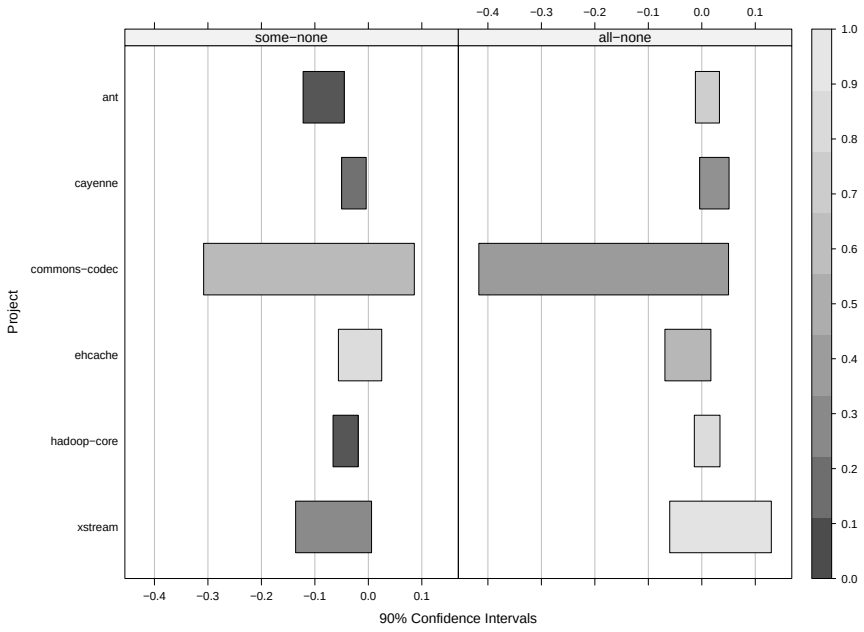


Figure 6.11: Confidence Intervals for Change in Average Method Complexity

Finally, Figure 6.12 summarises the percentage differences between the averages for the some-none and the all-none average method complexity measurement comparisons, relative to the none measurements. These results show that when these developers exercised some of their changes with tests, on average they reduced the average method complexity of the methods they changed by 50–600%, as compared to when they exercised none of their changes with tests. However, these results also show that when these developers exercised all of their changes with tests, on average they reduced or increased the average method complexity of the methods they changed by 200–600% or 50–150% respectively, as compared to when they exercised none of their changes with tests.

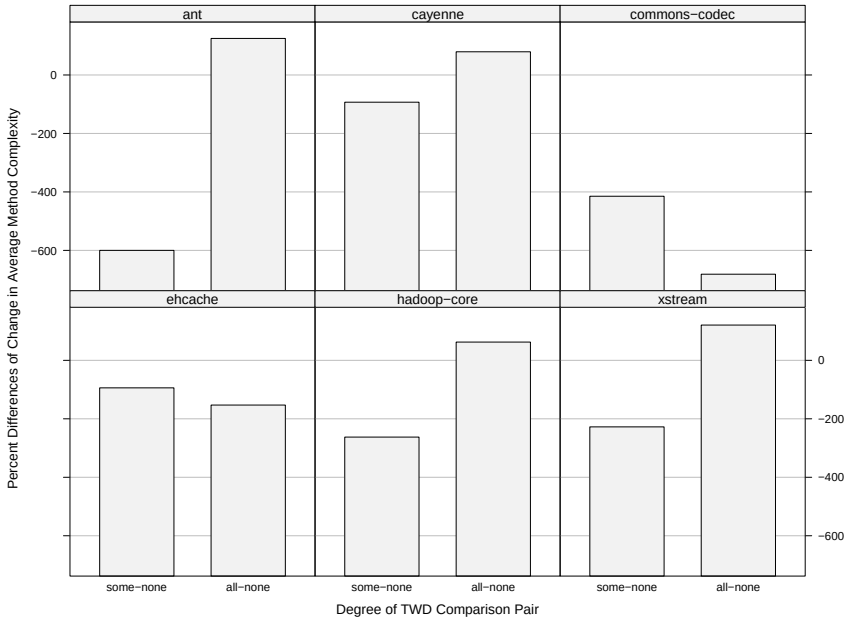


Figure 6.12: Percentage Differences for Change in Average Method Complexity

6.3 Results Discussion

6.3.1 Some TWD Results

As summarised above, when the developers practised TWD to some degree, they tended to make significantly bigger changes over time, to decrease the number of potential bugs by less, and to significantly decrease the average method complexity, all compared to when they did not practice TWD to any degree. They made significantly bigger changes over time, while significantly reducing the average method complexity of their product, but did not significantly affect the number of potential bugs in that product. Within these results, there are some that merit some additional discussion.

First, with respect to bigger changes over time, the commons-codec project was extreme: the other projects ranged from 63% to 262% but it was 4428%; in Figure 6.3 we only showed 500% so that the other comparisons would not be overwhelmed. This extreme result had at least a couple of factors. As compared to the other projects, the initial net size and the net size over time was quite low when the developers did not practice TWD. And, also as compared to the other projects, the initial net size and the net size over time was quite high when the developers practised TWD to a degree. After analysing a sampling of the source code changes, we conclude that the team's commitment to adding and modifying automated tests whenever they made substantial changes resulted in these extreme values.

Second, with respect to reducing the average method complexity, the ant project was extreme: the other projects with significant results ranged from 93% to 262% but it was 1050%; in Figure 6.12 we only showed 600% so that the other comparisons would not be overwhelmed. After analysing a sampling of the source code changes, we conclude that the team's commitment to refactoring and simplifying their design, when they practised TWD to a degree, resulted in this extreme value.

6.3.2 All TWD Results

Further, as summarised above, when the developers practised TWD fully, they tended to make significantly smaller changes over time, to decrease the number of potential bugs by less, and to decrease or increase the average method complexity, all compared to when they did not practice TWD to any degree. They made significantly smaller changes over time, but did not significantly affect the number of potential bugs or the average method complexity within that product. Within these results, there are some that merit some additional discussion.

First, with respect to smaller changes over time, two of the projects did not have significant results. Although their results were consistent with smaller changes over time, the commons-codec and the xstream projects had much smaller average differences, 10% and 21%, as compared with the other projects (75% to 91%). After analysing a sampling of the source code changes, we conclude that there are different reasons for each of these two projects. As mentioned above, for the commons-codec project, the initial net size and the net size over time was quite low when the developers did not practice TWD, because they were committed to practising TWD for substantial changes; therefore, the smaller changes they made when they practised TWD fully were more similar in size. But, for the xstream project, the developers added/changed more automated tests per change when they practised TWD fully, which increased the size of their product changes and made them more similar in size to the changes where they did not practice TWD at all.

Second, with respect to decreasing or increasing the average method complexity, the commons-codec and ehcache projects were different. For all of the other projects, the developers increased the average method complexity slightly when they practised TWD fully. After analysing a sampling of the source code changes, we conclude that because these changes were generally smaller, developers tended to just make the changes rather than taking the extra steps to refactor and simplify them. But, for the commons-codec and ehcache projects,

they reduced the average method complexity when they practised TWD fully; in these projects, the developers took the extra steps to refactor and simplify, which resulted in lower average method complexity.

6.4 Threats to Validity

6.4.1 Compilation Instructions

Some reviewers of our study identified a threat to the validity of its data. Specifically, they suggested that unstable compilation instructions could have introduced noise into the data. Below, we describe the threat, our analysis to remove it, and the results of our analysis.

If the compilation instructions for a project are not stable, then noise may be introduced into the data. When compilation instructions and developer changes intersect, compiled instruction changes result from changes in the compilation instructions *and* from source code changes made by a developer. The compiled instruction changes resulting from the compilation instructions can be thought of as noise, whereas the changes resulting from the source code changes can be thought of as signal. Therefore, the signal to noise ratio in the data could degrade if the compilation instructions change too often or if the intensity is too great; the noise could obfuscate results or cause the results to be interpreted incorrectly. Specifically, increased noise due to changes in compilation, rather than actual developer changes, could have raised the initial net size and discards over time values. This may have made it more difficult to identify that practising a certain degree of TWD results in bigger or smaller changes over time.

To remove this threat, we performed two types of analysis to confirm that the compilation instructions were stable for all of the project lifetimes that we studied. Initially, we gathered all of the revisions of the ant build files for the projects and compared the XML elements (and their compilation attributes) that the teams used to describe their Java compilation tasks. Following that, we replaced the Java class in the tools.jar that the ant javac task uses with a

"decorator" which logged the actual compilation instructions, so that we could compare those compilation instructions from revision to revision as we rebuilt the revisions.

As a result of these comparisons, we found that the compilation instructions were very stable, with a couple of specific exceptions: changing versions of the JDK and one specific compilation task attribute for one revision in the ant project. Since we were studying projects over the course of many years, all of the projects made a change from one version of the JDK to another; plus, a couple of the projects went through this transition twice. As well, when comparing the XML elements and their attributes, we also identified a single revision in the ant project (revision 267600) where the javac task's default deprecation attribute changed. Therefore, each project had at most three revisions where the compilation instruction noise was mixed together with the developer source code signal. As described in Chapter 5, when the data from these revisions was outside of the normal distribution, it was removed.

6.4.2 Static TWD Analysis

Another threat to the validity of our data is our measurement for the degree of TWD that a developer practised while making a product revision (Section 5.4.1). Presently, our measure is determined by static analysis. This analysis gives the developer the "benefit of the doubt" by potentially overestimating the degree to which they practised TWD—due to dynamic binding in Java we treat indirect references from an automated test method to a product method as if the product method would be exercised by the test method. It also treats a method as either completely exercised or not at all exercised, potentially crediting a developer with taking a TWD approach even if they have not exercised the new or changed paths within a particular changed product method.

These two problems could cause some data in our study to be mis-classified. A developer's product change that was made without exercising the product changes with automated tests could be interpreted as if the developer had prac-

tised TWD to some degree, or even fully. Because previous automated tests were in place to exercise some or all of the changed methods, changes that should be classified as none could be classified as some or all. Similarly, changes that should be classified as some could be classified as all.

In practise, the first problem with static analysis is nearly moot; after analysing all of the revisions of all of the projects we studied, 82% of the relationships from automated test methods to product methods are direct. Further, the other 18% of the relationships are from abstract automated test methods to abstract product methods; when we manually analysed 50 of these, only two product change methods had been treated as if they had been exercised when they were not. These results are consistent with our industrial experience—most automated tests are written at the unit level and test a public method of that unit directly.

With respect to the second problem, we manually analysed 100 samples of the changes that were made, across all projects, to see if we correctly classified them with respect to the degree of TWD that the developer actually practised: none, some, or all. For 96 of these samples, our static analysis classification and our manual classification match. Nonetheless, as identified in the future work of Chapter 7, we think that the degree of TWD that a developer practised should be determined with a dynamic approach.

Indeed, we would have preferred to have used a dynamic approach to measure the relationships between the automated product tests and the product. Rather than potentially overestimating the relationships and the path exercises, we could have measured the actual exercising of those changed product methods by executing the product tests. But, our early efforts to build and execute product tests against each product revision were stymied. We found doing so to be difficult or nearly impossible for the projects we studied; often it was difficult to build the product so that it could be monitored and more often the tests would not execute (due to further environmental setup requirements that changed over time). Therefore, we accepted this limitation for this study.

6.4.3 Tool Bias

A third threat to the validity of our data is tool bias, which could result from developers using the same tools as us (Section 5.4.1) to help them create bigger or smaller changes, reduce the number of potential bugs, or reduce the average method complexity. For example, if certain developers on a team were more prone to practice TWD and if they also used an additional tool to help them reduce the number of potential bugs when they made a change, this could confound the relationships between practising TWD and its effects on the number of potential bugs.

To investigate this threat, we downloaded and searched the mail archives[1] for each of the projects that we studied. Our search results do not indicate that any of the projects used additional tools to help them create bigger or smaller changes (specifically jeanda). Similarly, our search results do not indicate that any of the projects used additional tools to help them reduce their average method complexity (specifically cyvis).

However, our search results do indicate that a couple of the developers from two of the projects, commons-codec and hadoop-core, discussed using the findbugs tool during the final year of our study of their projects. Since there were no significant results related to findbugs within the commons-codec project, there are no results to invalidate. But, within the hadoop-core project, there was a significant result which indicated that practising TWD fully resulted in between 0 and 2 more potential bugs, on average. So, if we assume that developers using the findbugs tool would on average remove more potential bugs, as measured by findbugs, and we assume that the two hadoop-core developers used findbugs but did not practise TWD fully, then they may have created an artificial difference between themselves and the developers who did practise TWD fully. Therefore, this hadoop-core potential bugs significant result might be invalid.

[1] We used case-insensitive grep to search the archives for "cyvis," "findbugs," and "jeanda."

6.4.4 Potential Bugs

Another threat to the validity of our data, specifically our potential bugs data, is the indirect relationship between practising TWD and removing potential bugs, as defined by the findbugs tool. We and some of our study reviewers agree that developers that practise TWD are generally *not* intentionally and directly removing potential bugs from their product. And, we and they agree that it is more likely that the developers are practising TWD to ensure that their product changes function properly and to give them confidence that they haven't broken anything. But, we think that developers that practise TWD might indirectly remove more potential bugs, because their increased confidence allows them to refactor and simplify their product, which could remove bugs that were introduced because of complexity. In contrast, they are sceptical that practising TWD could result in a significant reduction in potential bugs, even indirectly, because they feel that the types of potential bugs removed as a result of refactoring are different than the types of potential bugs found by the findbugs tool.

Our thinking was and is based on both our experience and some of the past study results. In our experience, practising TWD can remove all sorts of potential problems, including bugs, from source code. And, several case studies have found a reduction in a variety of measures related to actual bugs when developers have practised TFD: (1) defect density (Williams et al. 2003, Bhat & Nagappan 2006, Sanchez et al. 2007, Nagappan et al. 2008); (2) defect rate (Maximilien & Williams 2003); and (3) fault rate (Damm & Lundberg 2006). Therefore, we think it was reasonable to study a potential relationship between developers practising TWD and the resulting changes in the number of potential bugs.

Their thinking is based on their experience and a lack of past study results specific to findbugs. In their experience, several other types of development activities are also likely to cause the removal of potential bugs (as defined by findbugs)—for example, design and review activities. And, they wonder

whether developers would even agree with the findbugs definitions of potential bugs, which could result in a higher measure of potential bugs even when the developers themselves would think that they had removed potential bugs. Also, although past studies have found a reduction in measures related to actual bugs, they point out that none of them have found a reduction in potential bugs (again, as defined by findbugs). Therefore, they question whether it was reasonable to study this potential relationship.

So, even if our study had found significant relationships between TWD and potential bugs, these results would have been subject, due to the threats to validity that these reviewers suggest.

6.5 Summary

Once we had designed our study, we executed it and then correlated and analysed the results. First, we determined whether each developer's change was made using a TWD approach, and to what degree, or if it was made using a TLD approach. Second, we measured some particular attributes of speed and quality for each of the developer changes. Third, we correlated the speed and quality measurements of the developer changes with the appropriate degree of TWD category. Finally, we performed statistical analyses on the relationships.

Overall, when the developers we studied practised TWD to some degree, they made significantly bigger changes over time and significantly reduced the average method complexity within their changes. As well, when they practised TWD fully, they made significantly smaller changes over time but did not significantly affect the average method complexity within their changes. In neither case did they significantly affect the potential bugs within their changes.

Chapter 7

Conclusion

Teams can increase their efficiency by completing their products more quickly, or at a higher quality, or both (Chapter 1). Within the realm of Software Engineering, they can execute a Software Process Model to structure their activities so that they can develop their products more quickly or at a higher level of quality (Chapter 1). Test-With Development (TWD) is a technique that they can practice to potentially increase their speed and their product's quality (Chapters 1 and 2). But, because of the mixed results and some of the limitations of previous related studies (Chapter 2), we were motivated to study the effects of TWD. So, we prepared to study TWD's effects by defining a software change model (Chapter 3) and calibrating it (Chapter 4). Then we designed our study (Chapter 5), executed it, analysed its results, and evaluated their validity (Chapter 6).

In this chapter, we present our conclusions, drawn from our interpretations of those study results. Additionally, we relate our results to the results of the previous studies on TFD. Then, we discuss the implications and limitations of our results. Finally, we suggest some potential future work that we have identified during the planning, execution, and reporting of this study.

7.1 Conclusions

Based on our study design (Chapter 5) and our study results (Chapter 6), when a developer practised TWD while making product changes, the effects of those changes on the attributes related to team speed and product quality were significantly different than when they did not. Further, the degrees to which they practised TWD corresponded with different effects on the specific attributes that we studied.

When they exercised some of their product changes with tests, on average they made significantly bigger product changes over time that resulted in significantly less average method complexity in the product. In contrast, when they exercised all of their product changes with tests, they made significantly smaller product changes over time that did not significantly increase or decrease the average method complexity in the product. Finally, the number of potential bugs within the product was not significantly different according to the degree of TWD that a developer practised.

These results agree in part and disagree in part with our initial hypotheses, in Chapter 5. The results for developers practising TWD to some degree agree with our initial hypotheses: bigger changes over time and less average method complexity. However, the results for developers practising TWD fully do not agree with our initial hypotheses: smaller changes over time with no significant differences in average method complexity. Finally, we did expect that practising TWD to some degree, and fully, would decrease the number of potential bugs; our results do not support that conclusion.

We interpret these results to indicate that automated testing of product changes, *before* sharing those changes with the rest of the team, enabled the teams that we studied to reduce their product's internal complexity more quickly than when they did not. Further, because of the ratio of some TWD to all TWD changes (between 2:1 and 4:1, Appendix E), we interpret these results to indicate that the teams that we studied reduced their product's internal

complexity incrementally, rather than all at once.

We interpret the bigger changes over time and the lower average method complexity to be the result of refactoring; when a developer has automated tests to support refactoring, they are more likely to simplify their product and remove duplication from it because they have more confidence that they will not break anything. Bigger changes over time that increase product quality are consistent with the "perfective" changes described by Hindle et al. (2008) and the refactoring changes described by Ratzinger et al. (2008). And, we interpret the smaller changes over time to be the ultimate benefit of reducing the product's complexity; once areas of a product have been simplified then only smaller changes will be necessary to achieve the desired effects.

Based on these interpretations, we conclude that teams can increase their efficiency by practising TWD. In particular, we conclude that they can reduce their product's complexity more quickly when they practice TWD, incrementally adding automated tests to support its simplification. Thus, they can increase their team speed while they are increasing their product's quality.

7.2 Relation to Existing Studies

The existing studies on TFD are either case studies or controlled experiments which have compared the effects of TFD between projects and products, rather than within a product. That is, the unit of analysis has been the product, and how TFD affected it as compared to another, rather than how it affected the changes to the same product over time. Additionally, each of the studies, including our own, has studied different attributes related to team speed and product quality (see Chapter 2, Table 2.3). Finally, we have studied TWD, which includes TFD and two additional variants that also describe a developer delivering fine-grained changes to their product in addition to tests for those changes. These differences make it difficult to compare our results to the results of the other studies.

Nonetheless, when we do compare them, our results match some parts of the other studies' results. Our results for developers practising TWD to some degree match the increase in attributes related to speed from the results of the student studies (Chapter 2, Table 2.1) and the increase in attributes related to quality from the results of the professional studies (Chapter 2, Table 2.2). In contrast our results for developers practising TWD fully match the decrease in attributes related to speed from the results of the professional studies.

7.3 Implications and Limitations

One implication of our study is that the extra effort (perceived or actual) involved in maintaining automated tests can be justified. If bigger changes over time and reducing the average method complexity are deemed to be worth the extra effort, then a team should practice TWD to some degree. This implication might be particularly useful to a team that knows they need to make many changes over time to simplify a complex system. Plus, if eventual stability is valued by a team, practising TWD may allow them to get to the point where all product changes are exercised by automated tests and thus facilitate smaller changes over time.

Another implication of the study is that a team might benefit from testing their fine-grained product changes, even if they write their automated product tests along with or after their product changes. That is, they might benefit from TWD even if they do not practice TFD in a strict manner—it may be more effective to develop tests *with* product changes than to develop them *before* product changes.

However, our study design prevents us from proving (statistically) that practising TWD to a degree is the cause for the significant effects that we observed. Rather, we have only been able to prove (statistically) that there are significant differences between the means (averages) for some of the effects we studied—in particular, the net size over time and the average method complexity. This is

the main limitation of our study.

As well, we would not yet recommend generalising beyond the Java open-source population. Although we have no particular reasons to believe that the results may differ significantly, differences such as technology, experience, product type, and timelines could yield conflicting results. Hopefully, future studies can expand the population for which these results are valid and identify the boundaries of those where they are not.

7.4 Future Work

The most compelling question raised by this study is whether practising TWD to a degree is the cause (or a cause) of the significantly different effects we observed on the attributes related to team speed and product quality. So, we submit that this is an important area of future work.

As well, as described in Chapter 6, we think that measuring the degree of TWD more accurately is an important area of future work. That is, rather than measure the degree of TWD with a static analysis tool, we think that future work should attempt to measure it with a dynamic analysis tool (such as Clover, Cobertura, or Emma).

In addition, this study has identified that a team can benefit from developing and delivering product tests *with* their product changes. That is, they can benefit from practising TWD, which may or may not require practising TFD. So, we submit that an important area of future work will be determining whether the benefits of TWD come from developing product tests first (as in TFD), or just from developing product tests with product changes, as opposed to TLD.

We submit that another important area of future work is identifying the boundaries of the external validity for our results. For example, do these results hold if we expand to tens or hundreds of Java open-source projects? And, are the results similar for C# .NET open-source projects and for "closed-source" projects?

A final important area of future work is identifying the scope of the potential benefits of TWD. Are the potential benefits limited to increased net size over time and decreased average method complexity or are there others? And, are there potential detriments? To find out, we or others will need to study the effects of practising TWD on other attributes of team speed or product quality. Additionally, is there a profile with which a team could optimise the potential benefits of TWD by practising it to a certain degree, such as 25%, 50%, 75%, or 100%? Further, should a team practice TWD to different degrees, according to the current complexity of the product, or should they always practice it to a certain degree?

7.5 Summary

To provide a context for our work, we surveyed the history of team efficiency in general, and software development team efficiency in particular (Chapter 1). Then, after understanding the potential benefits of TWD and the results and limitations of previous TFD studies (Chapter 2), we were motivated to perform our own study to begin to answer our research question.

Our work to answer this research question has resulted in three contributions to knowledge: (1) our software change model; (2) our measurement method; and (3) our study results.

While preparing for our study, we defined an original software change model to support our measurements of team changes to their product and the effects of those changes (Chapter 3). With this software change model, we provide a way to measure the degree of TWD that a developer has practised when they made a change to their product. With it, we also provide ways to measure the initial net size of a developer's change, the discards over time of that change, and to derive the net size over time for the change. Along with this original model, we adopted two pre-existing models to facilitate measurements related to product quality (potential bugs and average method complexity).

As well, we defined an original method to measure the developer changes to their products. First, we calibrated our software change model with an auxiliary study and found that most of the discards from a change happen before 150 days (Chapter 4). Next, we designed our main study so that we could mine granular change data from multiple project repositories over long periods of time (Chapter 5)—in doing so, we were careful to contribute a repeatable approach and reusable supporting tools. Then, once we had executed our main study design, we correlated and analysed the data to obtain our results—here again, we contributed a reusable approach and more reusable supporting tools (Chapter 6).

Our final contribution to knowledge is our main study results. Our study results indicate that there are significant differences between developers practising TWD and the resulting effects on both of the team speed (initial net size and discards over time) and one of the product quality (average method complexity) attributes that we studied (Chapter 6). We interpret these results to indicate that the teams that practice TWD can reduce their product's complexity more quickly than when they do not. Therefore, we conclude that teams that practice TWD can increase their efficiency, because they can increase their speed and their product quality.

Appendix A

External Validation

We received external validation of our multiple comparative study from the empirical software engineering community. As a result of this validation, the study was published in the Empirical Software Engineering Journal (Bannerman & Martin 2010). Below is the abstract from that publication (in US English):

A.1 Abstract

Researchers have typically studied the effects of Test-First Development (TFD), compared to Test-Last Development (TLD), across groups or projects, and for relatively short durations. We defined Test-With Development (TWD) as more general than the fine-grained step of TFD, but also in contrast to the large-grained phase of TLD. With our definition, we performed a multiple comparative study to explore and describe TWD product changes, and the effects of those changes on two attributes related to team speed and two attributes related to product quality, within six long-term open-source projects. Our results indicate that when developers exercised some of their changes with automated tests, on average they made significantly larger changes over time while significantly reducing their product's complexity. And, when they exercised all of their changes with tests, on average they made significantly smaller changes

over time. We interpret these results to indicate that practicing TWD supports faster simplification of a product. Therefore, we conclude that teams that need to reduce their product's complexity can benefit from practicing TWD.

Appendix B

Programming Advances

From Assembly languages to C#, programming languages have evolved to help programmers develop more quickly, and to develop products of higher quality. For a couple of decades, structured programming languages allowed programmers to target different machines with the same source code and to structure their programs better. For the last couple of decades, object-oriented programming languages have allowed programmers to target different machines with the same compiled code and to structure their programs even better. Both of these categories of advances have helped programmers to focus more and more on the problems they are trying to solve, rather than trying to find a way to solve them with a language. Ultimately, these advances have helped programmers develop faster and better.

B.1 Structured Programming

In the 1970s and 1980s, structured programming and programming languages supporting this paradigm gained prominence. Spurred by encouragement from Böhm & Jacopini (1966) and Dijkstra (1968), in addition to the NATO conference results, the software community began to move from more complex, machine specific programming (assembly languages) to simpler, machine inde-

pendent programming (procedural languages). They began to structure their programs in hierarchies of procedures, or subroutines. By writing structured programs, programmers could focus more on the problem at hand and less on the details of the hardware. This shift allowed them to write programs more quickly, and at a higher quality–they had to write less code to do the same things and what they did write was easier to "prove" or analyse.

C and Pascal were two of the most popular languages during these decades, and they continue to be fairly popular today. Between 1969 and 1972, Dennis Ritchie invented and evolved the C programming language. He first used it to port the kernel of the UNIX operating system from an assembly language to a structured programming language (Kernighan 1988). And in 1970, Niklaus Wirth invented the Pascal programming language to promote good structured programming practises (Wirth 1971). It became the standard to teach computer science students about structured programming and has also been used extensively in industry.

B.2 Object-Oriented Programming: Foundation

During this same time frame, a competing programming paradigm was evolving: object-oriented programming. In the mid to late 1960s, Kristen Nygaard and Ole-Johan Dahl introduced the Simula 67 programming language (Nygaard & Dahl 1981). With it, programmers could model real-world objects, group their algorithms with the data that they acted upon, and reuse algorithms and data in an inheritance hierarchy. In 1972, David Parnas published his famous "information hiding" paper (Parnas 1972), advocating this paradigm. In it, he suggested that programs should be structured by hiding design decisions, rather than be based on the structure of a subroutine in a flowchart. By doing so, developers could isolate areas most likely to change and thus complete changes to a program more quickly and with higher quality.

Goldberg & Robson described the Smalltalk programming language (1989), which provided several important innovations. Apart from being the first fully object-oriented language, it provided a graphical user interface (GUI), an integrated development environment (IDE) with an incremental compiler and an interactive debugger, running on a virtual machine (VM) with a garbage collector (GC), and a very simple syntax. These properties allowed a programmer to work more quickly: they could browse the class library, change their program, and test and debug it without leaving their IDE or experiencing the typical compile and link delay. Arguably, it also allowed programmers to create higher quality programs, because of its simplicity.

Bjarne Stroustrup introduced the C++ language in the early 1980s, extending the C language with object-oriented constructs (Stroustrup 2000). While C++ did not provide much innovation, it did expose many more programmers to the object-oriented paradigm and facilitated reuse of existing C class libraries. It also provided a "production speed" object-oriented language, since Smalltalk had higher hardware resource requirements than were generally acceptable or practical at that time.

B.3 Object-Oriented Programming: Mainstream

However, despite the potential benefits, structured programming was still the most prevalent paradigm. until the early 1990s. It was then that the software community began migrating to object-oriented programming and its accompanying languages. This migration coincided with the Reengineering Management movement, spearheaded by Hammer and his colleagues (Hammer 1990). It also coincided with faster and cheaper hardware, which made object-oriented languages and development environments more practical. Software managers began to adopt this paradigm with the hope and belief that it would make their organisations more efficient, by helping their programmers program more quickly and at a higher quality level.

Since then, Java and C# have joined C++ as the most prolific object-oriented languages. In 1995, James Gosling introduced the Java programming language (Gosling 2005). The Java language team adopted a simplified C/C++ syntax, provided more economical support for most of the innovations from the Smalltalk language, and introduced some of their own. In particular, they popularised the "write once, run anywhere" idea, introduced "first class" interfaces, and provided access to a plethora of reusable class libraries. Because of these, Java programmers can write, test, and debug their programs faster and at a higher quality level.

More recently, in 2001, Anders Hejlsberg released the C# programming language (Archer 2001), which is substantially the same as the Java programming language. However, a notable additional benefit is that C# supports language inter-operability in conjunction with the .NET framework. Therefore, C# programmers can also use components in other .NET languages, potentially reducing their programming time and increasing their product quality.

Appendix C

Product Reconstruction

Our first step, for both the auxiliary study (Chapter 4) and the main study (Chapter 5), was to reconstruct the revisions of a product, according to its configuration management log. At a high level, this involved identifying the revisions for the product, assembling the files for each particular revision, and building and storing each revision of the product and the product tests. We partially automated this process, so that it could run unattended until it encountered a problem.

C.1 Revision Identification

Subversion (SVN) (http://subversion.tigris.org) is a "next-generation" software configuration management (SCM) tool. Each time a software developer commits a set of changes to subversion, a "snapshot" is created. Each snapshot is associated with a revision number.

We used the subversion log tool to determine all of the revisions for a project in a relevant time period. Listing C.1 is an example of the subversion log output, excerpted from the Tomcat 6.0X project. It shows the log output for two changes: revisions 414965 and 415610. With this information, we mapped the time of a revision, the revision number, and the files that changed.

```
r415610 | funkman | 2006-06-20 07:12:30 -0400 (Tue, 20 Jun 2006) | 3 lines
Changed paths:
   M /tomcat/tc6.0.x/trunk/java/org/apache/catalina/startup/Bootstrap.java
Remove JMX warning message since java5 is required
------------------------------------------------------------------------
r414965 | fhanik | 2006-06-16 20:12:20 -0400 (Fri, 16 Jun 2006) | 3 lines
Changed paths:
   M /tomcat/tc6.0.x/trunk/java/org/apache/coyote/http11/Http11AprProcessor.java
   M /tomcat/tc6.0.x/trunk/java/org/apache/coyote/http11/InternalAprInputBuffer.java
Revert, if Content-Length: <some large value> is an acceptable work around
```

Listing C.1: svn log information

C.2 Revision Assembly

We used the subversion update tool to assemble a particular revision. Listing C.2 is an example of the subversion update output, also for the Tomcat 6.0X project, updating to the 415610 revision.

```
[tomcat-60X]# svn update -r 415610
U java/org/apache/catalina/startup/Bootstrap.java
Updated to revision 415610.
```

Listing C.2: svn update information

C.3 Revision Build

Once we had a snapshot of a project revision, we used ant (http://ant.apache.org) to compile the source code and to archive the resulting class files. Ant is a standard build tool for Java software projects. Software developers describe ant build targets in an XML file, usually called build.xml.

However, many ant build files differ in the naming, ordering, and content of their build targets. Therefore, we needed to analyse each build file to determine

how to compile the Java code and where to locate the output class files. Not surprisingly, because it is consistent with our experience, the build files and their component build tasks often change over the course of a project's lifetime.

Listing C.3 is an example command, also for the Tomcat 6.0X project, which cleans up the output from a build, prepares for a build, and then compiles the source code.

```
ant clean build-prepare compile
```

Listing C.3: Sample build command

Once we had the class files for a particular snapshot of the project, we archived them into two files: a product file and a product-tests file. Then, we packaged the archive files in a zip file, named according to the revision; Listing C.4 shows the contents of a particular package file.

```
[tomcat-60X]# unzip -l product.415610.zip | grep ".*.tgz$"
product.415610.tgz
product-tests.415610.tgz
```

Listing C.4: Sample package contents

We then repeated this "building" activity for the next revision, according to the revision history. At certain points, a build would fail. Sometimes, the build would fail because the classes would not compile. However, more often, the build would fail because the build file had changed. So, upon encountering a build error, we needed to analyse the build file and figure out how to build the particular revision.

C.4 Revision Storage

Once we had built a particular revision, we stored it in a "catalogue" for the project. We did this for at least a couple of reasons: we did not want to lose the

representation of a revision (when the next build ran) and we needed to have the revision to execute comparisons and measurements. Listing C.5 shows an abbreviated list of the builds that we were able to reconstruct for the Tomcat 6.0X project.

```
[tomcat−60X]# ll ∗.zip |awk '{print $9}'
product.389946.zip
product.390055.zip
product.390155.zip
...
product.414906.zip
product.414965.zip
product.415610.zip
```

Listing C.5: Sample package listing

C.5 Studied Projects

Listing C.6 shows the subversion URLs for the source projects the we studied in our auxiliary study (Chapter 4); the "industry" project URL cannot be shown as permission to do so was not received. Listing C.7 shows the subversion URLs for the source projects the we studied in our main study (Chapter 5).

```
ehcache: https://svn.sourceforge.net/svnroot/ehcache/trunk
hibernate: http://anonhibernate.labs.jboss.com/trunk/Hibernate3
industry: <industry project: permission not received>
tomcat: http://svn.apache.org/repos/asf/tomcat/tc6.0.x/trunk
xerces: http://svn.apache.org/repos/asf/xerces/java/trunk
xstream: svn://svn.xstream.codehaus.org/xstream/trunk/xstream
```

Listing C.6: Auxiliary study project URLs

```
ant: http://svn.apache.org/repos/asf/ant/core/trunk
cayenne: http://svn.apache.org/repos/asf/incubator/cayenne/main/trunk
commons-codec: http://svn.apache.org/repos/asf/commons/proper/codec/trunk
ehcache: https://ehcache.svn.sourceforge.net/svnroot/ehcache/trunk
hadoop-core: http://svn.apache.org/repos/asf/hadoop/core/trunk
xstream: http://svn.codehaus.org/xstream/trunk/xstream
```

Listing C.7: Main study project URLs

Appendix D

Measurement Details

In the following sections, we describe how we used the jeanda tool to measure the degree of TWD and the initial net size for a product change. For both sections, we use the same measurement specification (Listing D.1), which identifies the ant product changes made between revision 268541 and 268543. In Chapter 4, as part of our auxiliary study design, we described how we used the jeanda tool to measure the discards over time for a product change (Section 4.2.3).

```
2001-01/product.268541.tgz 2001-01/product.268543.tgz
```

Listing D.1: Measurement Specification

D.1 Degree of TWD

To measure the degree of TWD for a product change, we: (1) specified the product changes that we wanted to measure (Listing D.1); (2) executed the measurement with the jeanda toolset (Listing D.2); and (3) stored the results of the measurement (Listing D.3) [1].

[1] The measurement output is partially elided to save space

```
java
  -cp jeanda.jar
  org.tigris.jeanda.metrics.tdd.DirCoverageMultiEvaluator
  <comparison specification file path>
```

Listing D.2: Measurement Command

```
--
Processing product type: ExecuteWatchdog...
--
Processing product method: ExecuteWatchdog/protected void cleanUp()...
--
Method add/change not covered.
--
.
.
.
Processing product method: ExecuteWatchdog/public synchronized void start(Process arg1)...
--
Add/change covered directly by: ExecuteWatchdogTest/public void testFailed()...
Add/change covered directly by: ExecuteWatchdogTest/public void testManualStop()...
Add/change covered directly by: ExecuteWatchdogTest/public void testNoTimeOut()...
Add/change covered directly by: ExecuteWatchdogTest/public void testTimeOut()...
--
Processing product method: ExecuteWatchdog/public synchronized void stop()...
--
Method add/change not covered.
--
Product type method coverage: 1 / 6
Product type instruction coverage: 6 / 31
Product type byte coverage: 10 / 57
--
.
.
.
--
```

```
Product overall method coverage ratio: 5 / 17
Product overall method coverage percent: 29%
--
Product overall instruction coverage ratio: 81 / 304
Product overall instruction coverage percent: 27%
--
Product overall byte coverage ratio: 153 / 595
Product overall byte coverage percent: 26%
--
```

Listing D.3: Measurement Output

D.2 Initial Net Size

To measure the initial net size for a product change, we: (1) specified the product changes that we wanted to measure (Listing D.1); (2) executed the measurement with the jeanda toolset (Listing D.4); and (3) stored the results of the measurement (Listing D.5).

```
java
  -cp jeanda.jar
  org.tigris.jeanda.metrics.cmd.DirSumMultiComparator
  <comparison specification file path>
```

Listing D.4: Measurement Command

```
<class-comparison-sum>
  <methodBytesAdded>1358.0</methodBytesAdded>
  <methodBytesEqual>3924.0</methodBytesEqual>
  <methodBytesDeleted>1381.0</methodBytesDeleted>
  <methodInstructionsAdded>676.0</methodInstructionsAdded>
  <methodInstructionsEqual>1874.0</methodInstructionsEqual>
  <methodInstructionsDeleted>602.0</methodInstructionsDeleted>
  <methodsAdded>20.0</methodsAdded>
  <methodsChanged>8.0</methodsChanged>
```

```
  <methodsEqual>43.0</methodsEqual>
  <methodsDeleted>7.0</methodsDeleted>
</class-comparison-sum>
```

Listing D.5: Measurement Output

Appendix E

Project Summary Results

In the sections below, we present the project summary results from our main study (Chapter 6). We present them with tables that show the descriptive and inferential statistics of the summarised measurements, one project per section.

E.1 ant

The ant (http://ant.apache.org) project team develops a free and open-source product that provides a build system for Java developers—developers write their build tasks in an XML format. More than 40 different developers have contributed to its development, since its inception in early 2000. The project development continues today. During that time, they have committed more than 10 000 changes to their configuration management repository, more than 7000 of which have been changes to their Java files, and more than 2600 of which resulted in changes to their compiled product.

Although the product was being developed in January of 2000, no automated tests were driving its development until about July of that year. Since then, the team has practised TWD to a varying degree, sometimes adopting a test-with approach, and sometimes opting for the test-last approach.

Of the 2653 "normal" changes to the product that we analysed: (1) about

80% had no exercise of the changed product methods by the product tests; (2) about 13% had some exercise of the changed product methods by the product tests; (3) about 7% had all of the changed product methods exercised by the product tests.

Table E.1 presents a summary of the measurement means for each group and overall, in addition to the overall standard deviation. Table E.2 presents a summary of the significance tests for the comparisons between groups.

Measure	*None* μ	*Some* μ	*All* μ	*Overall* μ	*Overall* σ
speed: initial net size	58.4	150.9	9.2	67.5	221.3
speed: discards over time	0.221	0.206	0.122	0.214	0.360
speed: net size over time	45.5	119.8	8.1	NA	NA
quality: potential bugs	-2.607	-2.442	-2.257	-2.57	5.77
quality: method complexity	0.008	-0.076	0.018	-0.003	0.325

Table E.1: ant—Summary of descriptive statistics

Measure	*comparison pair*	*ci lower*	*ci upper*	$p - value$
speed: initial net size	none,some	-128.1	-56.7	< 0.01
speed: initial net size	none,all	41.4	57.1	< 0.01
speed: discards over time	none,some	0.014	0.016	< 0.01
speed: discards over time	none,all	0.098	0.010	< 0.01
quality: potential bugs	none,some	-0.844	0.514	0.689
quality: potential bugs	none,all	-0.885	0.185	0.281
quality: method complexity	none,some	0.045	0.122	< 0.01
quality: method complexity	none,all	-0.033	0.012	0.439

Table E.2: ant—Summary of Welch Two Sample t-test

Consider the results presented in Table E.1. These represent the mean values of the dependent variables, categorised by the degree to which the developer of each change practised TWD: none, some, or all.

With respect to the attributes related to team speed, on average, the ant developers made bigger changes over time when they exercised some of their product changes with automated product tests (119.8 compiled instructions over 150 days) and smaller changes over time when they exercised all of them (8.1 compiled instructions over 150 days), both as compared to when they exercised none of them (45.5 compiled instructions over 150 days). Each of the net size over time mean values is derived from an initial net size mean value and the

discards over time value—for example, the initial net size mean value for the none category is 58.4 compiled instructions and the discards over time mean value is 0.221 (22.1%) over 150 days; therefore, the net size over time mean value is 45.5 compiled instructions over 150 days.

As well, the first four rows of Table E.2 indicate that the differences between these mean values for initial net sizes and discards over time are not likely due to chance (p-value < 0.01). Also, based on the confidence interval in the first row of Table E.2, we can be confident that the initial net size mean value from the some category is between 56.7 and 128.1 compiled instructions bigger than the initial net size mean value from the none category—the confidence intervals indicate the differences from the some or all categories to the none category. Similarly, based on the third row of Table E.2, we can be confident that the discards over time value from the some category is between 0.014 and 0.016 (1.4% and 1.6%) smaller than the discards over time mean value from the none category.

With respect to the attributes related to product quality, on average, the ant developers reduced the number of potential bugs by less when they exercised some of their product changes with automated product tests (-2.442) and by slightly less still when they exercised all of them (-2.257), both as compared to when they exercised none of them (-2.607); each of the categories reduced the number of potential bugs on average. Also, on average, the ant developers reduced the average method complexity when they exercised some of their product changes with automated product tests (-0.076) and increased it when they exercised all of them (0.018), both as compared to when they exercised none of them (0.008).

In contrast to the attributes related to team speed, the last four rows of Table E.2 indicate that there is only one row that is not likely due to chance (p-value < 0.01); the next to last row indicates that the average method complexity means between the none and some categories are significantly different. Also, based on that row, we can be confident that the method complexity mean from

the some category is between 0.045 and 0.122 less than the method complexity mean from the none category.

In summary, when the ant developers practised TWD to a degree (some), they made significantly bigger changes over time, decreased the number of potential bugs by less, and significantly reduced the average method complexity, all compared to when they did not practice TWD to any degree (none). On average, they made 163% bigger changes over time ($(119.8-45.5)/45.5 = 163\%$, from Table E.1). And, on average, they reduced the average method complexity by 1050% ($(-0.076-0.008)/0.008 = 1050\%$, also from Table E.1).

But, when they practised TWD fully (all), they made significantly smaller changes over time, decreased the number of potential bugs by even less, and increased the average method complexity, also all compared to when they did not practice TWD to any degree (none). On average, they made 82% smaller changes over time ($(8.1-45.5)/45.5 = 82\%$, from Table E.1).

E.2 cayenne

The cayenne (http://cayenne.apache.org) project team develops a free and open-source product that provides a persistence framework. In particular, the framework includes an object-relational mapping (ORM) tool and some remoting services.

Since its inception, in May of 2005, a relatively small team of 14 developers has contributed to its development. As of 2009, they have committed more than 6 000 changes to their configuration management repository. Nearly 3000 of these have been changes to their Java files, and more than 1600 of them resulted in changes to their compiled product.

Right from the start, the team began evolving a suite of automated tests. However, like the other projects, the team has practised TWD to a varying degree, sometimes adopting a test-with approach, and sometimes opting for the test-last approach.

Of the 1681 "normal" changes to the product that we analysed: (1) about 42% had no exercise of the changed product methods by the product tests; (2) about 42% had some exercise of the changed product methods by the product tests; (3) about 14% had all of the changed product methods exercised by the product tests.

Table E.3 presents a summary of the measurement means for each group and overall, in addition to the overall standard deviation. Table E.4 presents a summary of the significance tests for the comparisons between groups.

Measure	*None* μ	*Some* μ	*All* μ	*Overall* μ	*Overall* σ
speed: initial net size	45.8	80.0	3.7	55.3	194.6
speed: discards over time	0.414	0.454	0.324	0.420	0.436
speed: net size over time	26.8	43.7	2.5	NA	NA
quality: potential bugs	-2.082	-1.806	-1.833	-1.929	3.69
quality: method complexity	0.029	0.002	0.052	0.020	0.259

Table E.3: cayenne—Summary of descriptive statistics

Measure	*comparison pair*	*ci lower*	*ci upper*	$p - value$
speed: initial net size	none,some	-51.3	-17.3	< 0.01
speed: initial net size	none,all	24.0	60.0	< 0.01
speed: discards over time	none,some	-0.040	-0.039	< 0.01
speed: discards over time	none,all	0.088	0.090	< 0.01
quality: potential bugs	none,some	-0.597	0.046	0.158
quality: potential bugs	none,all	-0.670	0.171	0.329
quality: method complexity	none,some	0.004	0.050	0.058
quality: method complexity	none,all	-0.051	0.004	0.162

Table E.4: cayenne—Summary of Welch Two Sample t-test

In summary, assuming we interpret the results as we did in Section E.1, when the cayenne developers practised TWD to a degree (some), they made significantly bigger changes over time, decreased the number of potential bugs by less, and increased the average method complexity by significantly less, all compared to when they did not practice TWD to any degree (none). On average, they made 63% bigger changes over time ($(43.7 - 26.8)/26.8 = 63\%$). And, on average, they increased the average method complexity by 93% less ($(0.002 - 0.029)/0.029 = 93\%$).

But, when they practised TWD fully (all), they made significantly smaller

changes over time, decreased the number of potential bugs by less, and increased the average method complexity, also all compared to when they did not practice TWD to any degree (none). On average, they made 91% smaller changes over time ($(2.5 - 26.8)/26.8 = 91\%$).

E.3 commons-codec

The commons-codec (http://commons.apache.org/codec) project team develops a free and open-source framework for coding and decoding. The project originated as an effort to consolidate coding and decoding behaviour from various projects.

About 20 developers have contributed to the project, between its inception (2003) and the present (2009). During that time, they have committed about 450 changes to their configuration management repository, about half of which have been changes to their Java files, and more than 30 of which resulted in changes to their compiled product.

Right from the beginning, the team has developed their product and their automated product tests. However, like the other projects, the team has practised TWD to a varying degree, sometimes adopting a test-with approach, and sometimes opting for the test-last approach.

Of the 31 "normal" changes to the product that we analysed: (1) about 25% had no exercise of the changed product methods by the product tests; (2) about 25% had some exercise of the changed product methods by the product tests; (3) about 50% had all of the changed product methods exercised by the product tests.

Table E.5 presents a summary of the measurement means for each group and overall, in addition to the overall standard deviation. Table E.6 presents a summary of the significance tests for the comparisons between groups.

In summary, assuming we interpret the results as we did in Section E.1, when the commons-codec developers practised TWD to a degree (some), they

Measure	*None* μ	*Some* μ	*All* μ	*Overall* μ	*Overall* σ
speed: initial net size	6.9	261.1	5.1	74.2	290.3
speed: discards over time	0.271	0.133	0.115	0.156	0.324
speed: net size over time	5.0	226.4	4.5	NA	NA
quality: potential bugs	-0.200	-0.167	-0.429	-0.302	1.245
quality: method complexity	-0.027	-0.139	-0.211	-0.147	0.367

Table E.5: commons-codec—Summary of descriptive statistics

Measure	*comparison pair*	*ci lower*	*ci upper*	$p - value$
speed: initial net size	none,some	-467.8	-40.6	0.056
speed: initial net size	none,all	-101.4	105.1	0.976
speed: discards over time	none,some	0.138	0.152	< 0.01
speed: discards over time	none,all	0.158	0.171	< 0.01
quality: potential bugs	none,some	-0.841	0.774	0.944
quality: potential bugs	none,all	-0.589	1.046	0.636
quality: method complexity	none,some	-0.086	0.308	0.341
quality: method complexity	none,all	-0.050	0.417	0.192

Table E.6: commons-codec—Summary of Welch Two Sample t-test

made significantly bigger changes over time, decreased the number of potential bugs by less, and decreased the average method complexity, all compared to when they did not practice TWD to any degree (none). On average, they made 4428% bigger changes over time ($(226.4 - 5.0)/5.0 = 4428\%$).

But, when they practised TWD fully (all), they made smaller changes over time, decreased the number of potential bugs, and decreased the average method complexity, also all compared to when they did not practice TWD to any degree (none). None of these results were significant.

E.4 ehcache

The ehcache (http://ehcache.sourceforge.net) project team develops a free and open-source product that provides general purpose caching. This product enables Java developers to increase the performance of their applications.

Between its inception and now, only 3 developers have contributed to its development. During that time, they have committed more than 700 changes to their configuration management repository, about 400 of which have been

changes to their Java files, and more than 130 of which resulted in changes to their compiled product.

Right from inception, the team has co-evolved their product and their automated product tests. But, they have not always executed a test-driven approach. Rather, they have practised TWD to a varying degree, sometimes adopting a test-with approach, and sometimes opting for the test-last approach.

Of the 136 "normal" changes to the product that we analysed: (1) about 33% had no exercise of the changed product methods by the product tests; (2) about 50% had some exercise of the changed product methods by the product tests; (3) about 17% had all of the changed product methods exercised by the product tests.

Table E.7 presents a summary of the measurement means for each group and overall, in addition to the overall standard deviation. Table E.8 presents a summary of the significance tests for the comparisons between groups.

Measure	*None* μ	*Some* μ	*All* μ	*Overall* μ	*Overall* σ
speed: initial net size	26.7	82.7	7.5	49.3	113.9
speed: discards over time	0.174	0.193	0.255	0.196	0.327
speed: net size over time	22.0	66.7	5.6	NA	NA
quality: potential bugs	-2.312	-2.348	-1.375	-2.17	4.92
quality: method complexity	0.017	0.001	-0.009	0.004	0.139

Table E.7: ehcache—Summary of descriptive statistics

Measure	*comparison pair*	*ci lower*	*ci upper*	$p - value$
speed: initial net size	none,some	-88.4	-23.7	< 0.01
speed: initial net size	none,all	7.2	31.2	< 0.01
speed: discards over time	none,some	-0.020	-0.016	< 0.01
speed: discards over time	none,all	-0.084	-0.077	< 0.01
quality: potential bugs	none,some	-1.351	1.422	0.966
quality: potential bugs	none,all	-2.585	0.710	0.346
quality: method complexity	none,some	-0.025	0.056	0.519
quality: method complexity	none,all	-0.017	0.069	0.318

Table E.8: ehcache—Summary of Welch Two Sample t-test

In summary, assuming we interpret the results as we did in Section E.1, when the ehcache developers practised TWD to a degree (some), they made significantly bigger changes over time, decreased the number of potential bugs

slightly, and increased the average method complexity by less, all compared to when they did not practice TWD to any degree (none). On average, they made 203% bigger changes over time $((66.7 - 22.0)/22.0 = 203\%)$.

But, when they practised TWD fully (all), they made significantly smaller changes over time, decreased the number of potential bugs by less, and decreased the average method complexity, also all compared to when they did not practice TWD to any degree (none). On average, they made 75% smaller changes over time $((5.6 - 22.0)/22.0 = 75\%)$.

E.5 hadoop-core

The hadoop-core (http://hadoop.apache.org/core) project team develops a free and open-source product that provides a framework for distributed computing.

About 25 developers have contributed to its development. The project began in January of 2006 and continues today. During that time, the team has committed more than 3 000 changes to their configuration management repository, over 2200 of which have been changes to their Java files, and more than 1000 of which resulted in changes to their compiled product.

Soon after inception, the team began to develop automated tests in conjunction with their product code. Since then, the team has practised TWD to a varying degree, sometimes adopting a test-with approach, and sometimes opting for the test-last approach.

Of the 1069 "normal" changes to the product that we analysed: (1) about 55% had no exercise of the changed product methods by the product tests; (2) about 37% had some exercise of the changed product methods by the product tests; (3) about 8% had all of the changed product methods exercised by the product tests.

Table E.9 presents a summary of the measurement means for each group and overall, in addition to the overall standard deviation. Table E.10 presents a summary of the significance tests for the comparisons between groups.

Measure	*None* μ	*Some* μ	*All* μ	*Overall* μ	*Overall* σ
speed: initial net size	56.1	338.0	12.3	154.9	518.1
speed: discards over time	0.398	0.430	0.378	0.408	0.409
speed: net size over time	33.8	192.7	7.7	NA	NA
quality: potential bugs	-5.332	-4.791	-4.298	-5.06	7.51
quality: method complexity	0.016	-0.026	0.026	0.001	0.235

Table E.9: hadoop-core—Summary of descriptive statistics

Measure	*comparison pair*	*ci lower*	*ci upper*	$p - value$
speed: initial net size	none,some	-336.7	-227.2	< 0.01
speed: initial net size	none,all	18.7	69.0	< 0.01
speed: discards over time	none,some	-0.033	-0.031	< 0.01
speed: discards over time	none,all	0.017	0.023	< 0.01
quality: potential bugs	none,some	-1.287	0.205	0.233
quality: potential bugs	none,all	-1.980	-0.087	0.073
quality: method complexity	none,some	0.019	0.066	< 0.01
quality: method complexity	none,all	-0.034	0.014	0.510

Table E.10: hadoop-core—Summary of Welch Two Sample t-test

In summary, assuming we interpret the results as we did in Section E.1, when the hadoop-core developers practised TWD to a degree (some), they made significantly bigger changes over time, decreased the number of potential bugs by less, and decreased the average method complexity significantly, all compared to when they did not practice TWD to any degree (none). On average, they made 470% bigger changes over time ($(192.7 - 33.8)/33.8 = 470\%$). And, on average, they decreased the average method complexity by 262% ($(-0.026 - 0.016)/0.016 = 262\%$).

But, when they practised TWD fully (all), they made significantly smaller changes over time, decreased the number of potential bugs by significantly less, and increased the average method complexity slightly, also all compared to when they did not practice TWD to any degree (none). On average, they made 77% smaller changes over time ($(7.7 - 33.8)/33.8 = 77\%$). And, on average, they decreased the number of potential bugs by 19% less ($(-4.298 - -5.332)/-5.332 = 19\%$).

E.6 xstream

The xstream (http://xstream.codehaus.org) project team develops a free and open-source product that provides a framework for creating an XML document from a Java object and vice-versa.

About 10 developers have contributed to its development. They began their development in September 2003 and continue it even now. During this time, they have committed more than 1200 changes to their configuration management repository, more than 800 of which have been changes to their Java files, and more than 300 of which resulted in changes to their compiled product.

The team has evolved their product and their automated tests right from the start. However, the team has not executed TWD exclusively. Rather, the team has practised it to a varying degree, sometimes adopting a test-with approach, and sometimes opting for the test-last approach.

Of the 373 "normal" changes to the product that we analysed: (1) about 50% had no exercise of the changed product methods by the product tests; (2) about 38% had some exercise of the changed product methods by the product tests; (3) about 12% had all of the changed product methods exercised by the product tests.

Table E.11 presents a summary of the measurement means for each group and overall, in addition to the overall standard deviation. Table E.12 presents a summary of the significance tests for the comparisons between groups.

Measure	*None* μ	*Some* μ	*All* μ	*Overall* μ	*Overall* σ
speed: initial net size	31.6	114.8	25.4	60.8	138.0
speed: discards over time	0.293	0.307	0.306	0.300	0.405
speed: net size over time	22.3	79.6	17.6	NA	NA
quality: potential bugs	-0.333	-0.188	-0.500	-0.303	1.626
quality: method complexity	0.029	-0.037	0.064	0.010	0.392

Table E.11: xstream—Summary of descriptive statistics

In summary, assuming we interpret the results as we did in Section E.1, when the xstream developers practised TWD to a degree (some), they made significantly bigger changes over time, decreased the number of potential bugs

Measure	*comparison pair*	*ci lower*	*ci upper*	$p - value$
speed: initial net size	none,some	-112.0	-54.4	< 0.01
speed: initial net size	none,all	-7.5	19.9	0.455
speed: discards over time	none,some	-0.014	-0.011	< 0.01
speed: discards over time	none,all	-0.017	-0.012	< 0.01
quality: potential bugs	none,some	-0.481	0.191	0.475
quality: potential bugs	none,all	-0.083	0.416	0.271
quality: method complexity	none,some	-0.006	0.136	0.134
quality: method complexity	none,all	-0.130	0.060	0.545

Table E.12: xstream—Summary of Welch Two Sample t-test

by less, and decreased the average method complexity (nearly significantly), all compared to when they did not practice TWD to any degree (none). On average, they made 257% bigger changes over time $((79.6 - 22.3)/22.3 = 257\%)$. And, on average, they decreased the average method complexity by 228% $((-0.037 - 0.029)/0.029 = 228\%)$—however, this result was not quite significant (p-value 0.134).

But, when they practised TWD fully (all), they made smaller changes over time, decreased the number of potential bugs, and increased the average method complexity, also all compared to when they did not practice TWD to any degree (none). None of these results were significant.

Appendix F

Project Detailed Results

In the sections below, we present the project detailed results from our main study (Chapter 6). We present them with histogram and boxplot figures that show the descriptive statistics of the individual measurements, one project per section.

F.1 ant

Figure F.1 presents the overall distribution and segregated distributions of the initial net size measures. Figure F.2 presents the overall distribution and segregated distributions of the discards over time measures. Figure F.3 presents the overall distribution and segregated distributions of the change in potential bugs measures. Figure F.4 presents the overall distribution and segregated distributions of the change in average method complexity measures.

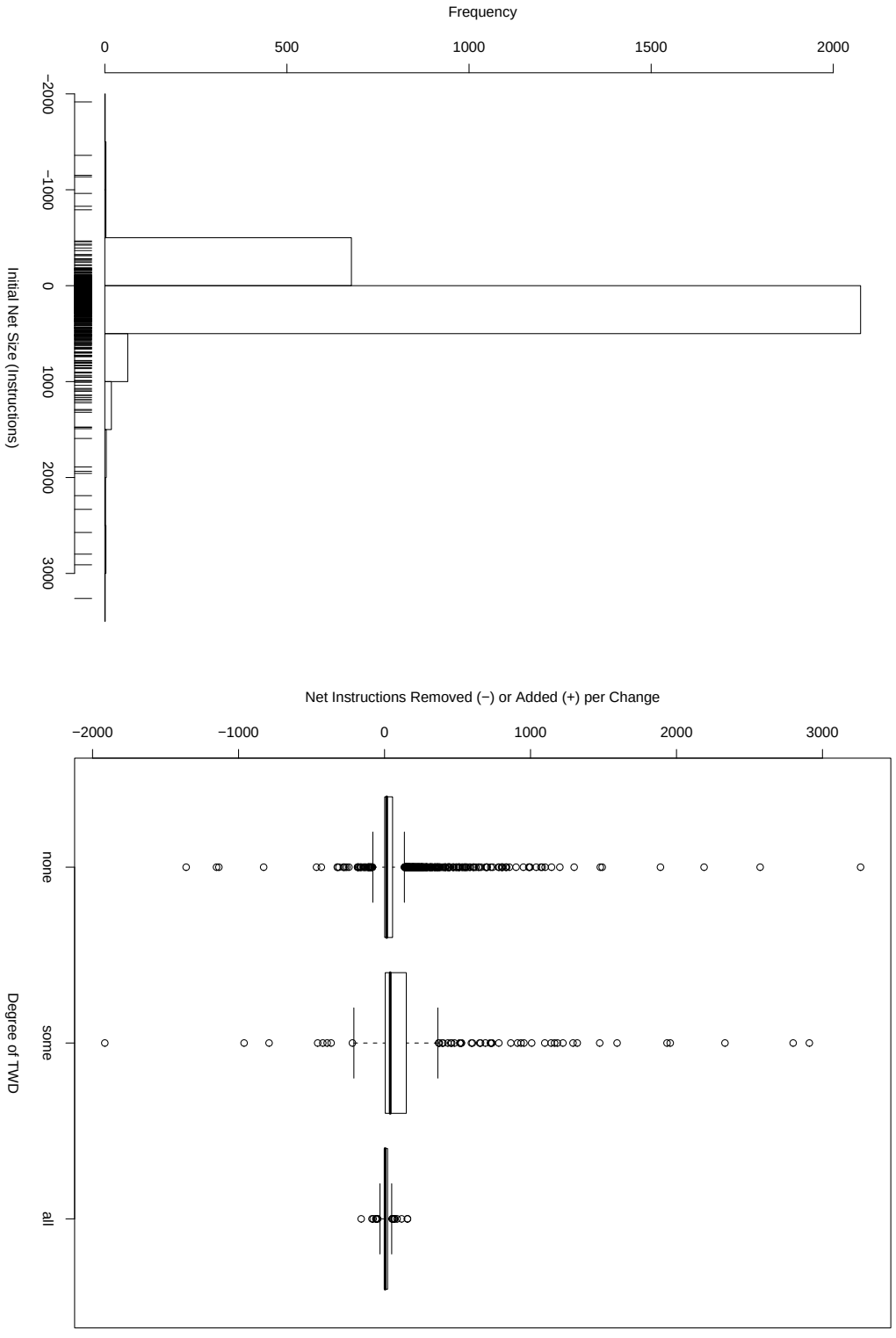


Figure F.1: ant—Histogram and boxplot for initial net size

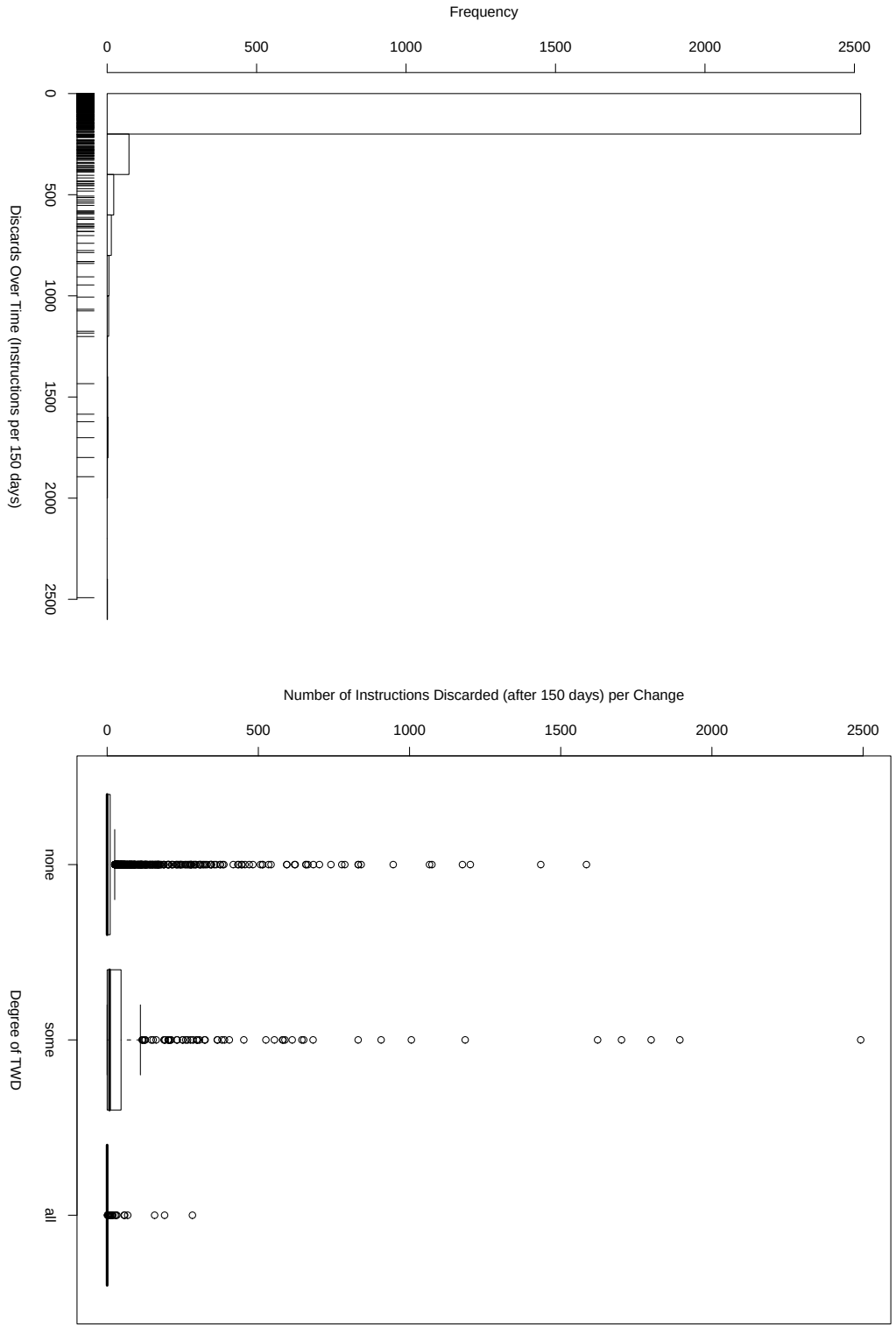


Figure F.2: ant—Histogram and boxplot for discards over time

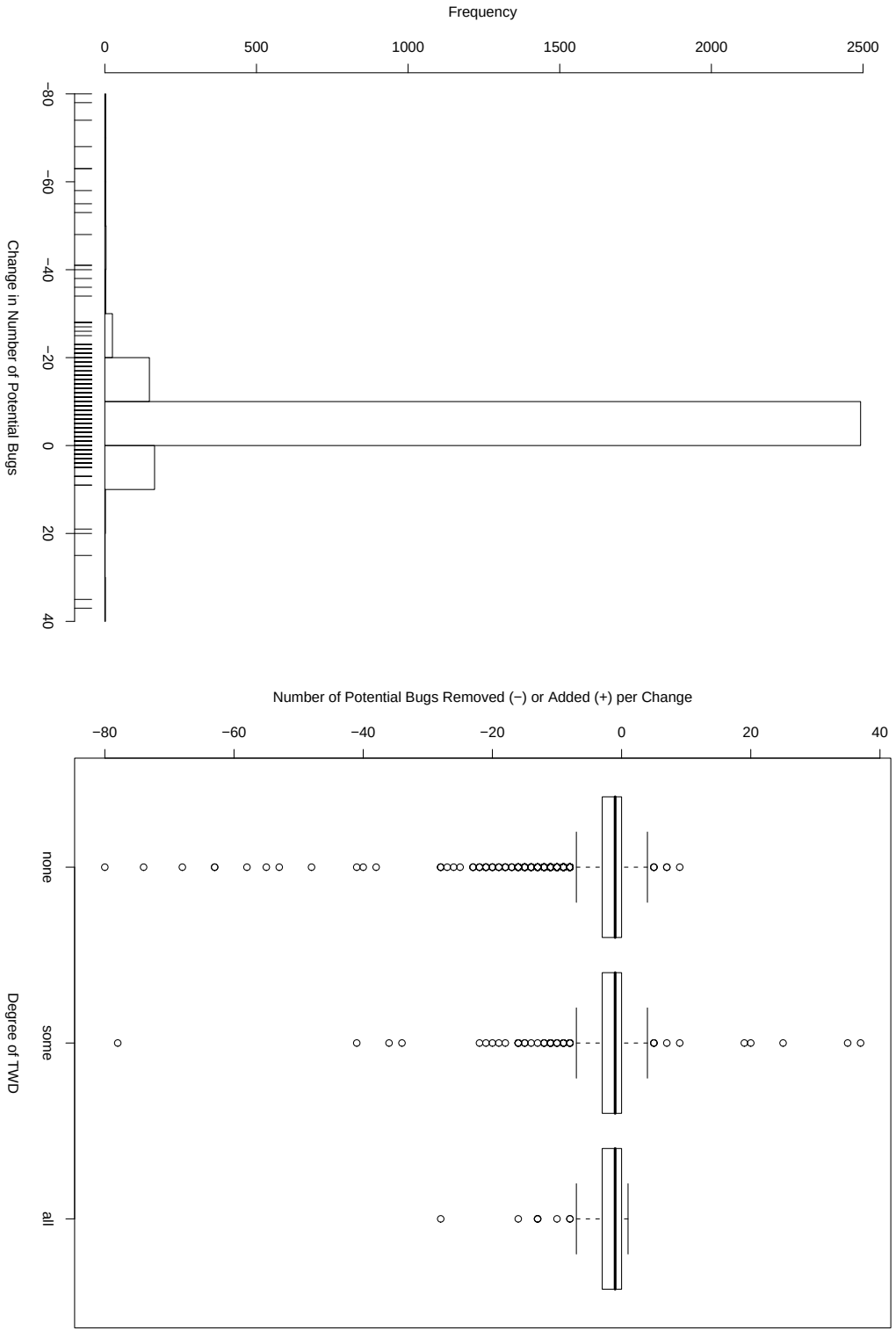


Figure F.3: ant—Histogram and boxplot for changes in potential bugs

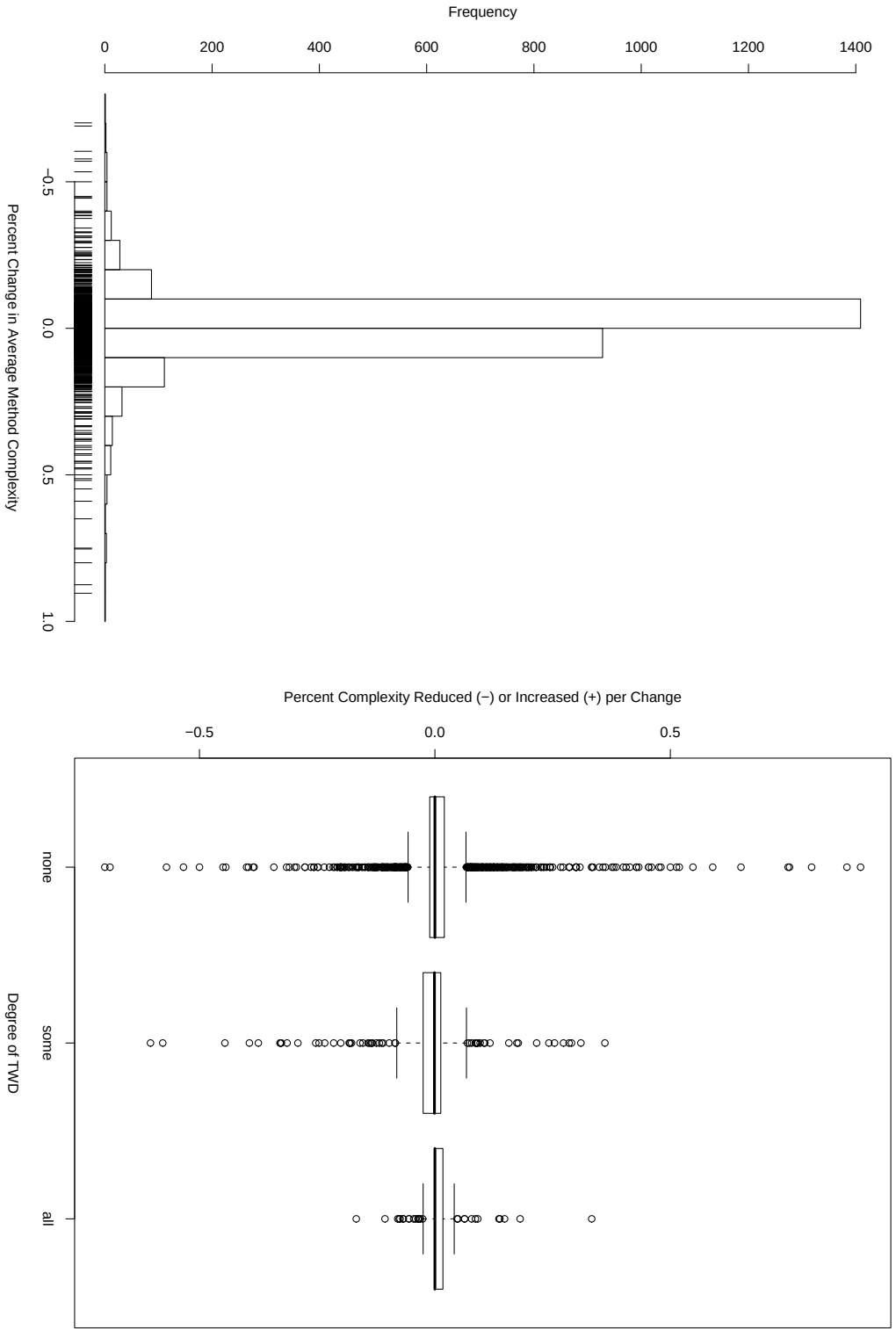


Figure F.4: ant—Histogram and boxplot for changes in average method complexity

F.2 cayenne

Figure F.5 presents the overall distribution and segregated distributions of the initial net size measures. Figure F.6 presents the overall distribution and segregated distributions of the discards over time measures. Figure F.7 presents the overall distribution and segregated distributions of the change in potential bugs measures. Figure F.8 presents the overall distribution and segregated distributions of the change in average method complexity measures.

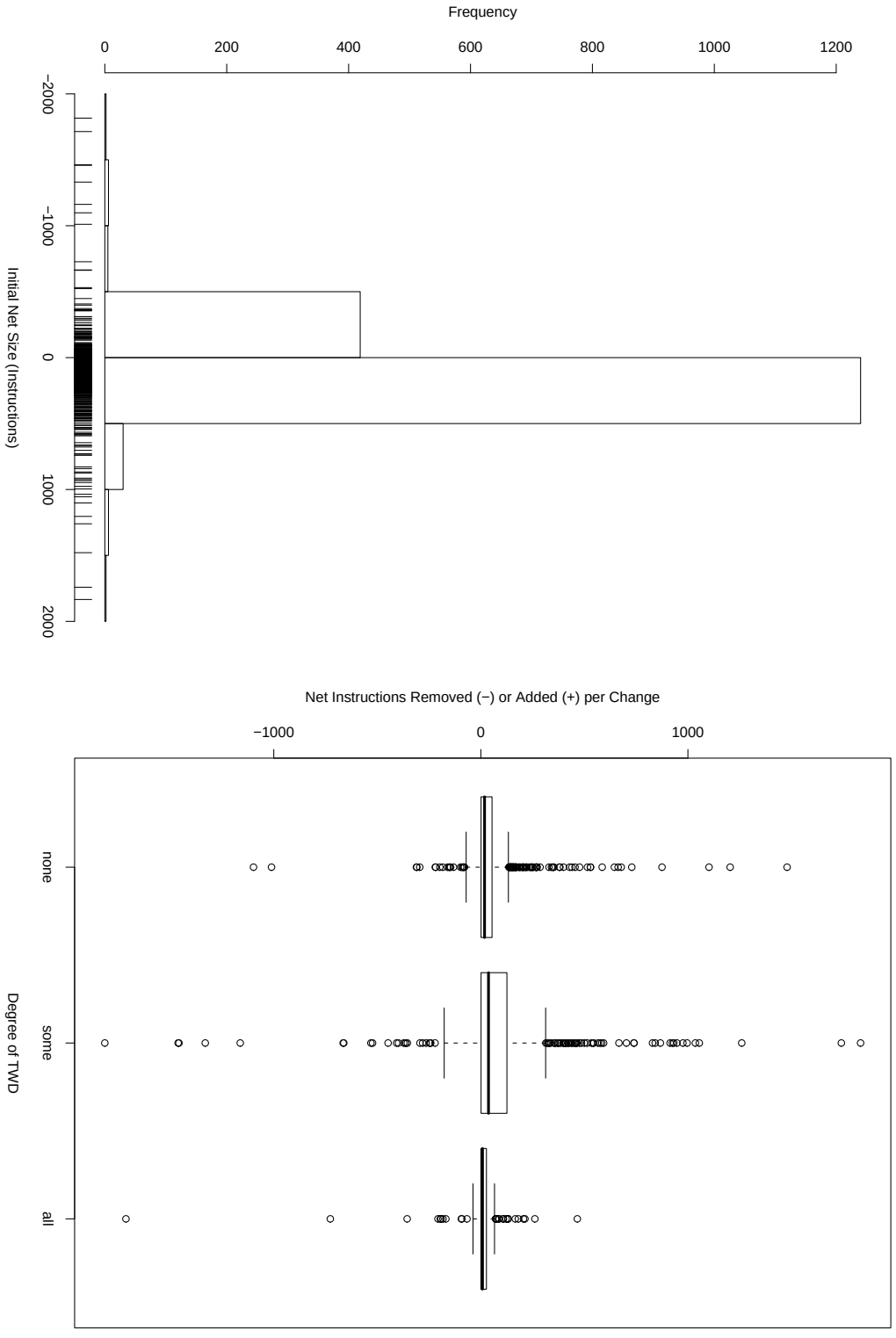


Figure F.5: cayenne—Histogram and boxplot for initial net size

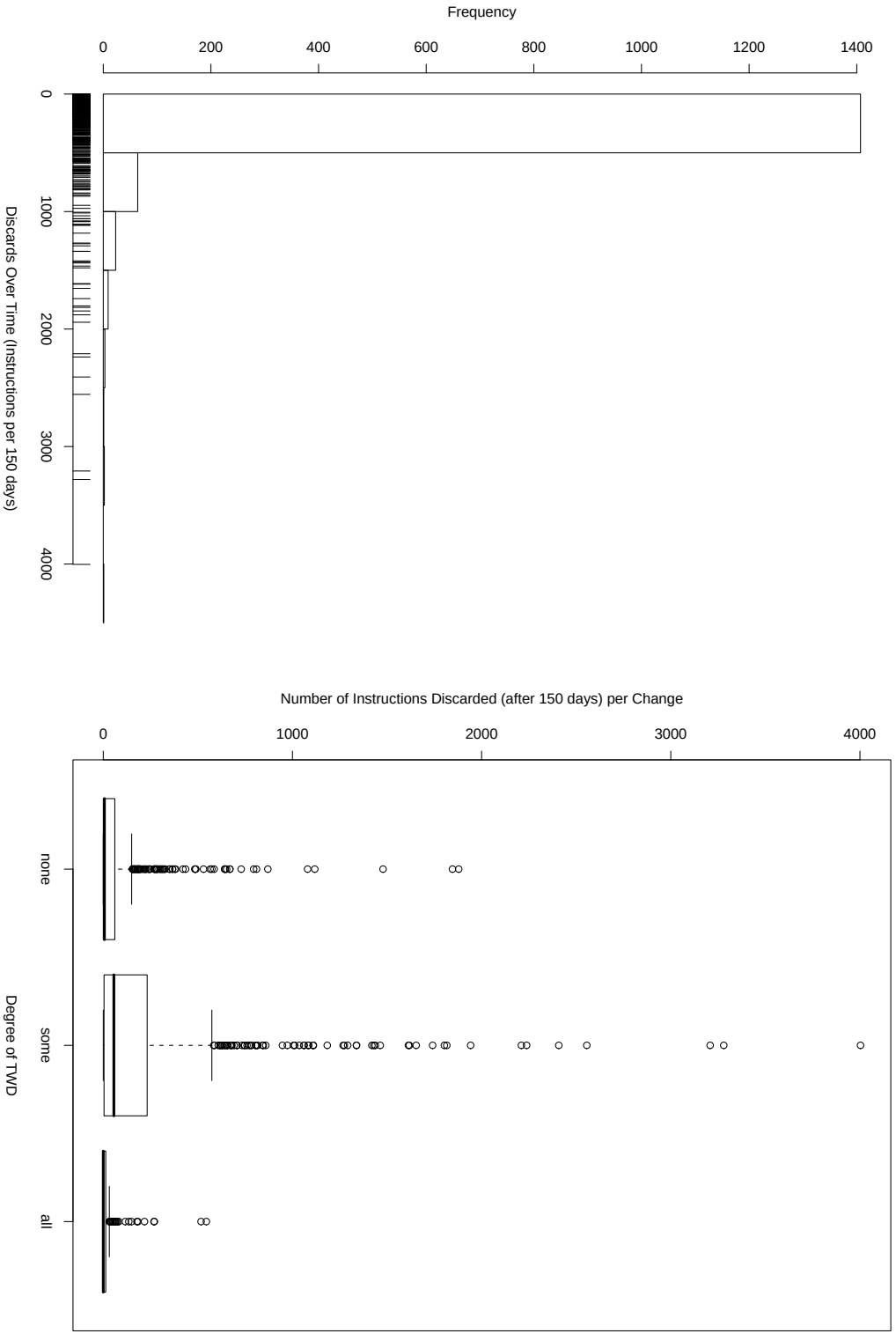


Figure F.6: cayenne—Histogram and boxplot for discards over time

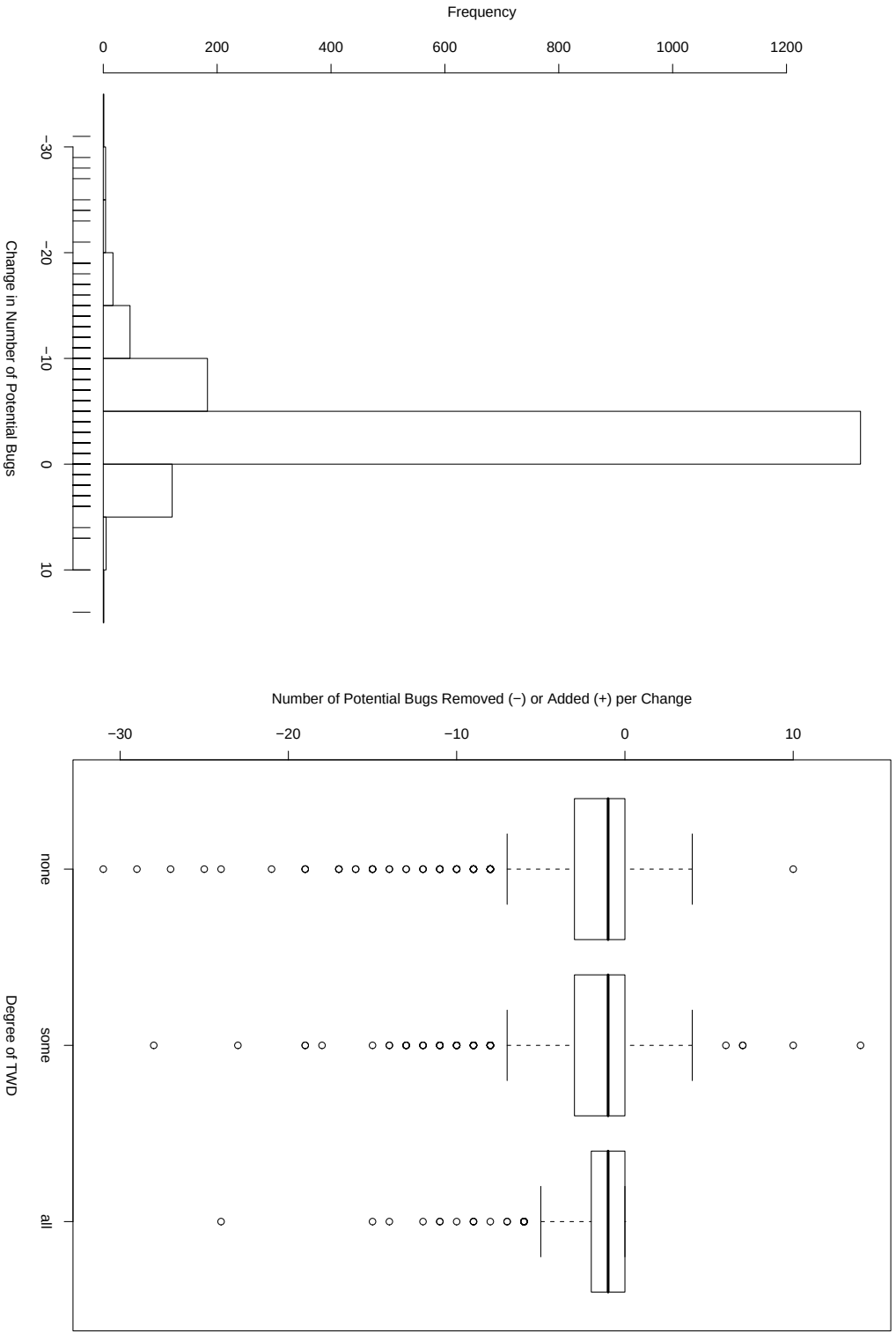


Figure F.7: cayenne—Histogram and boxplot for changes in potential bugs

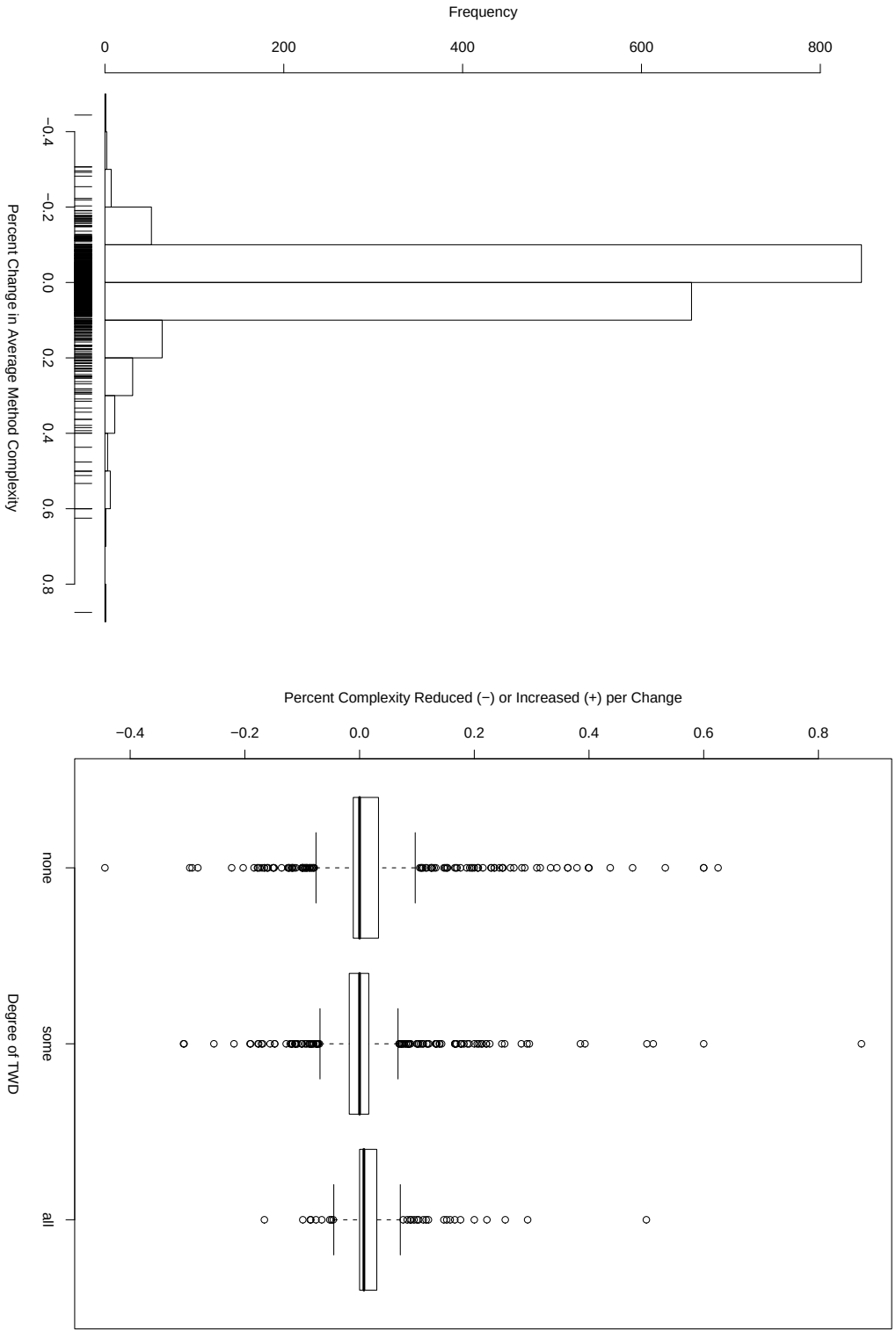


Figure F.8: cayenne—Histogram and boxplot for changes in average method complexity

F.3 commons-codec

Figure F.9 presents the overall distribution and segregated distributions of the initial net size measures. Figure F.10 presents the overall distribution and segregated distributions of the discards over time measures. Figure F.11 presents the overall distribution and segregated distributions of the change in potential bugs measures. Figure F.12 presents the overall distribution and segregated distributions of the change in average method complexity measures.

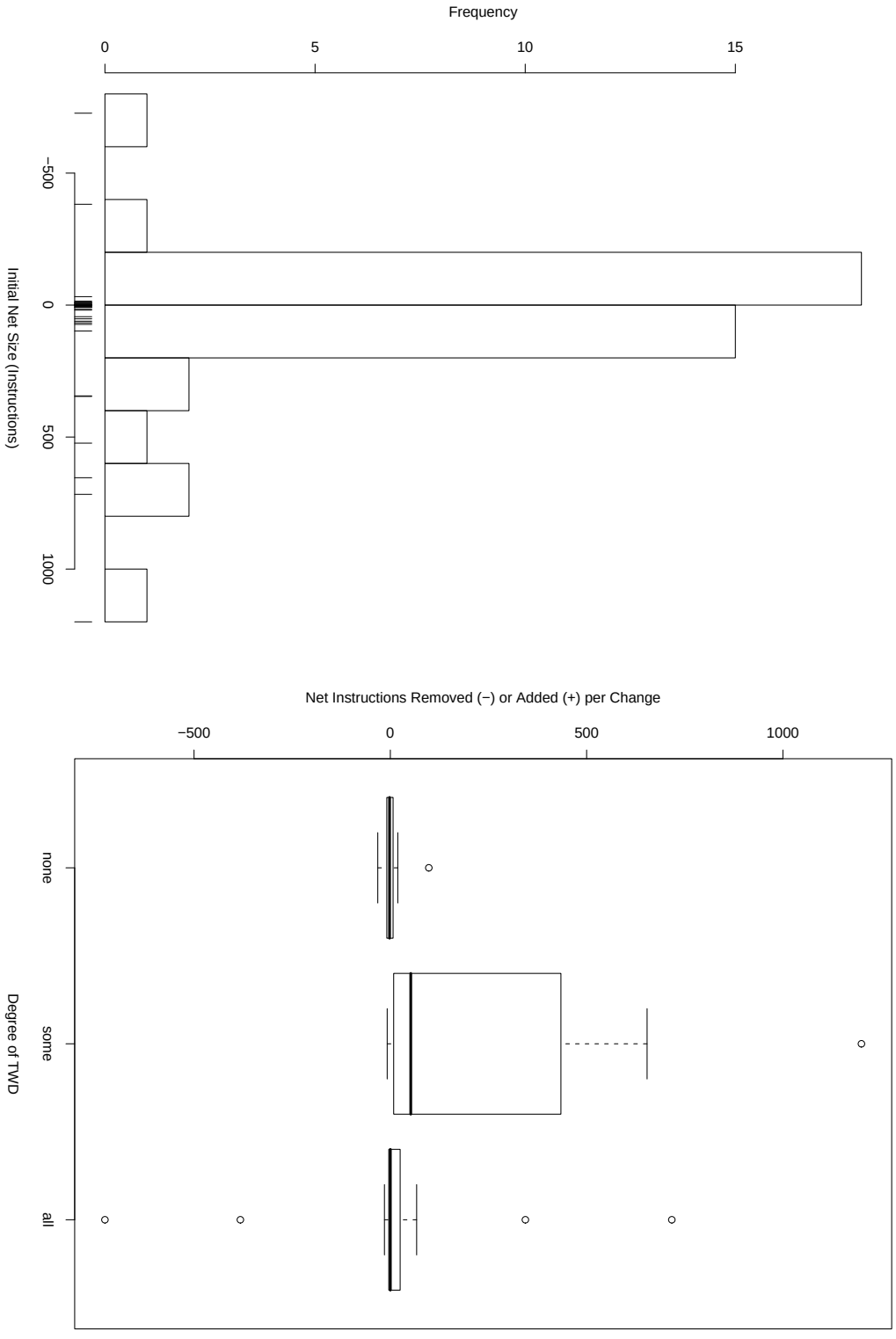


Figure F.9: commons-codec—Histogram and boxplot for initial net size

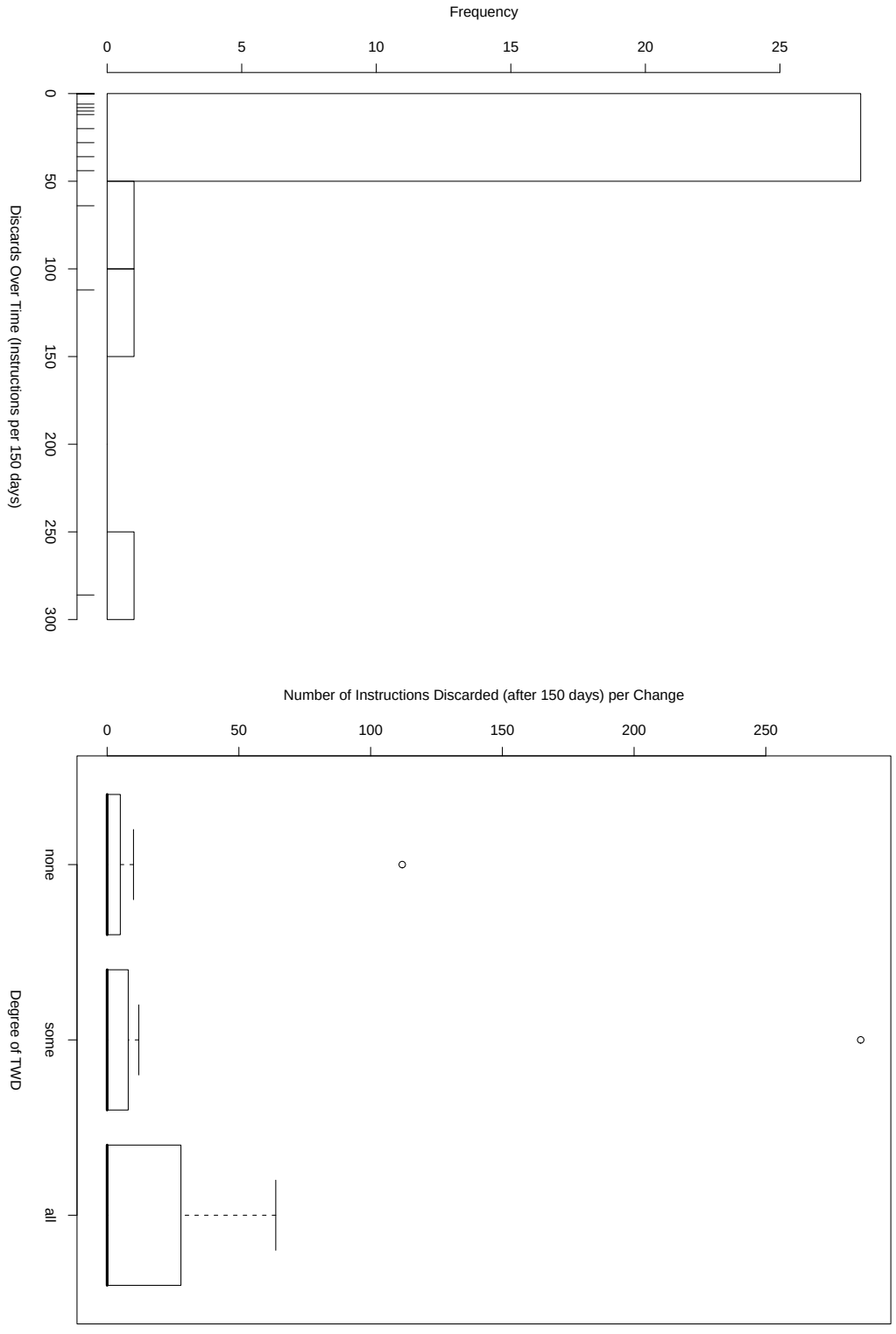


Figure F.10: commons-codec—Histogram and boxplot for discards over time

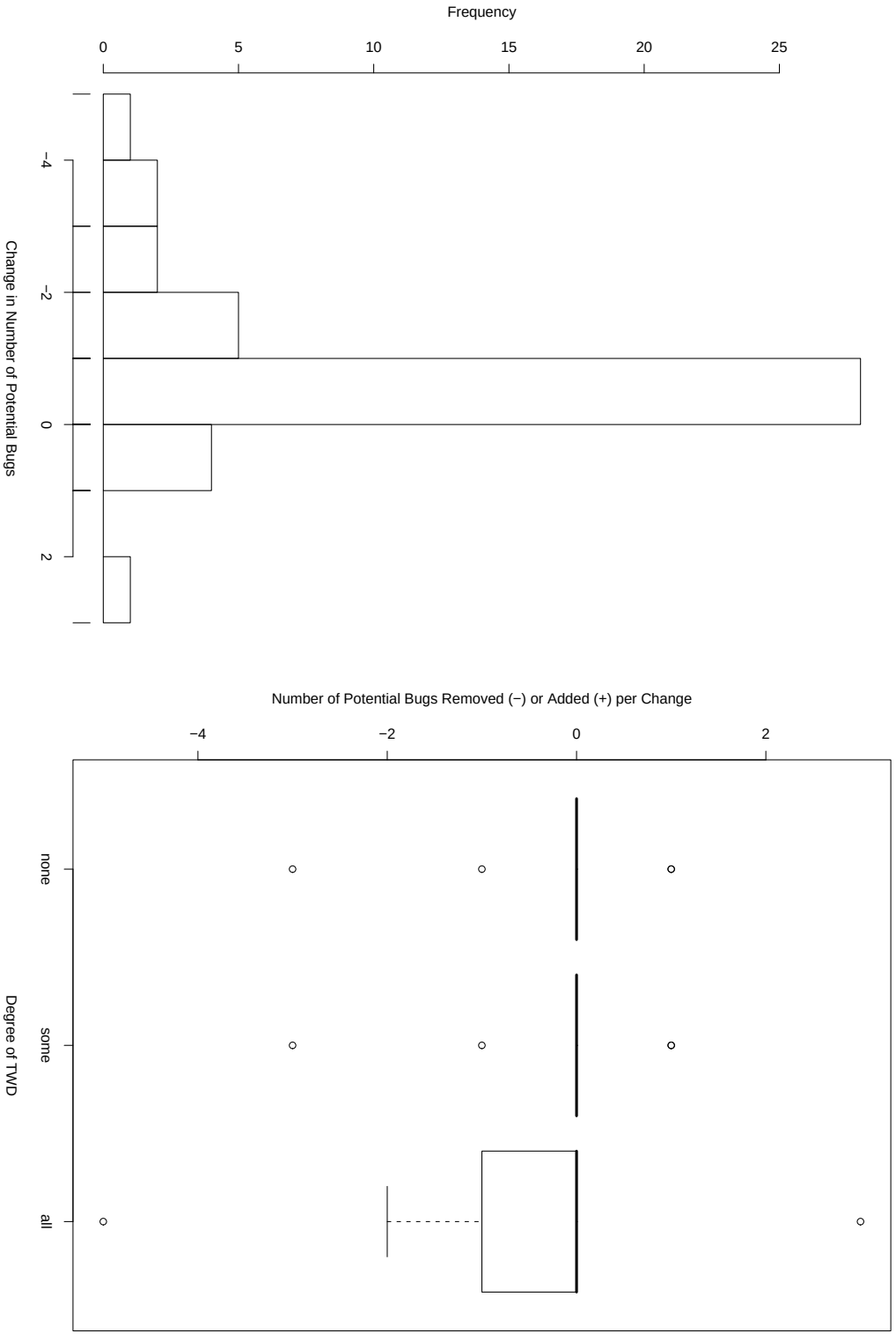


Figure F.11: commons-codec—Histogram and boxplot for changes in potential bugs

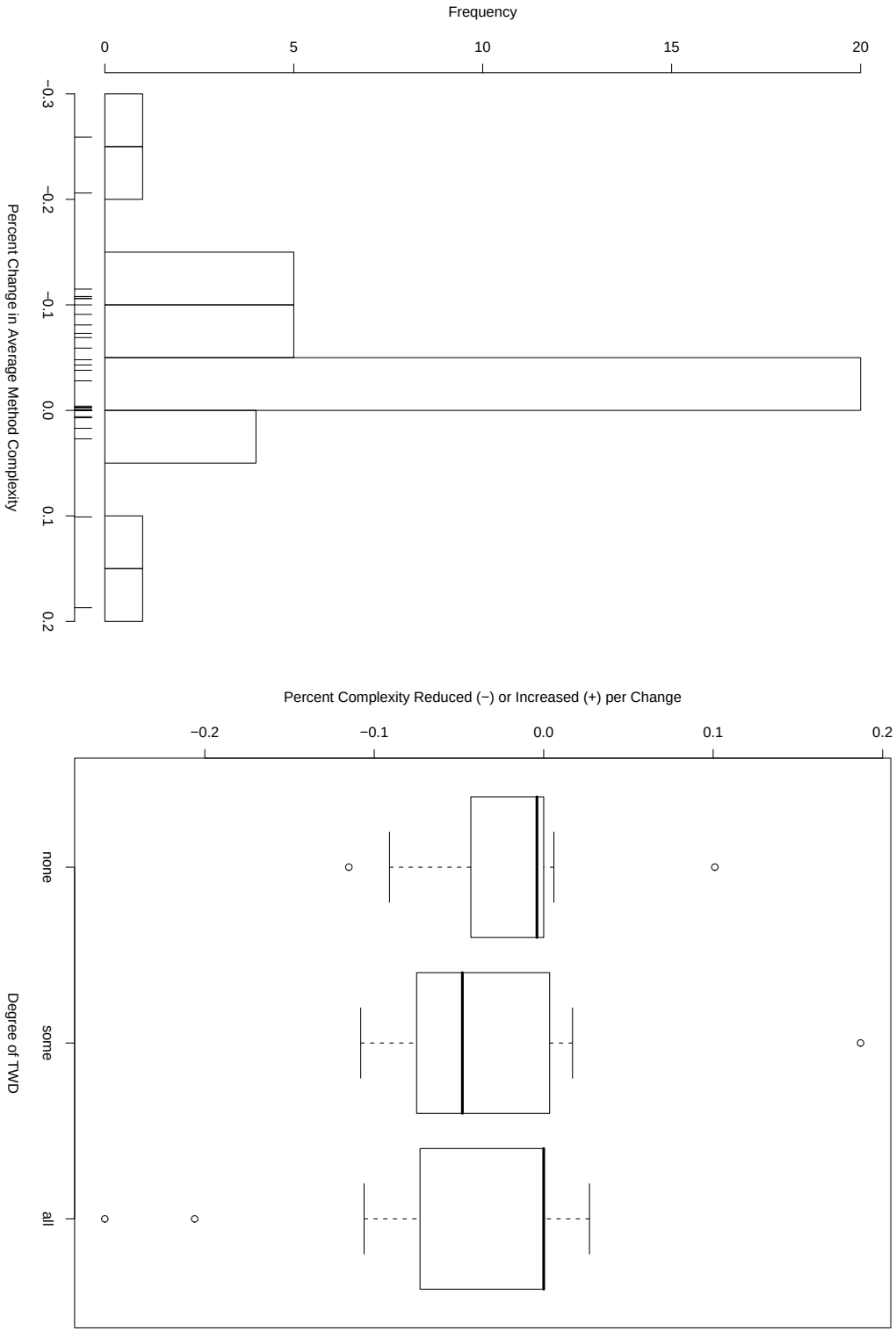


Figure F.12: commons-codec—Histogram and boxplot for changes in average method complexity

F.4 ehcache

Figure F.13 presents the overall distribution and segregated distributions of the initial net size measures. Figure F.14 presents the overall distribution and segregated distributions of the discards over time measures. Figure F.15 presents the overall distribution and segregated distributions of the change in potential bugs measures. Figure F.16 presents the overall distribution and segregated distributions of the change in average method complexity measures.

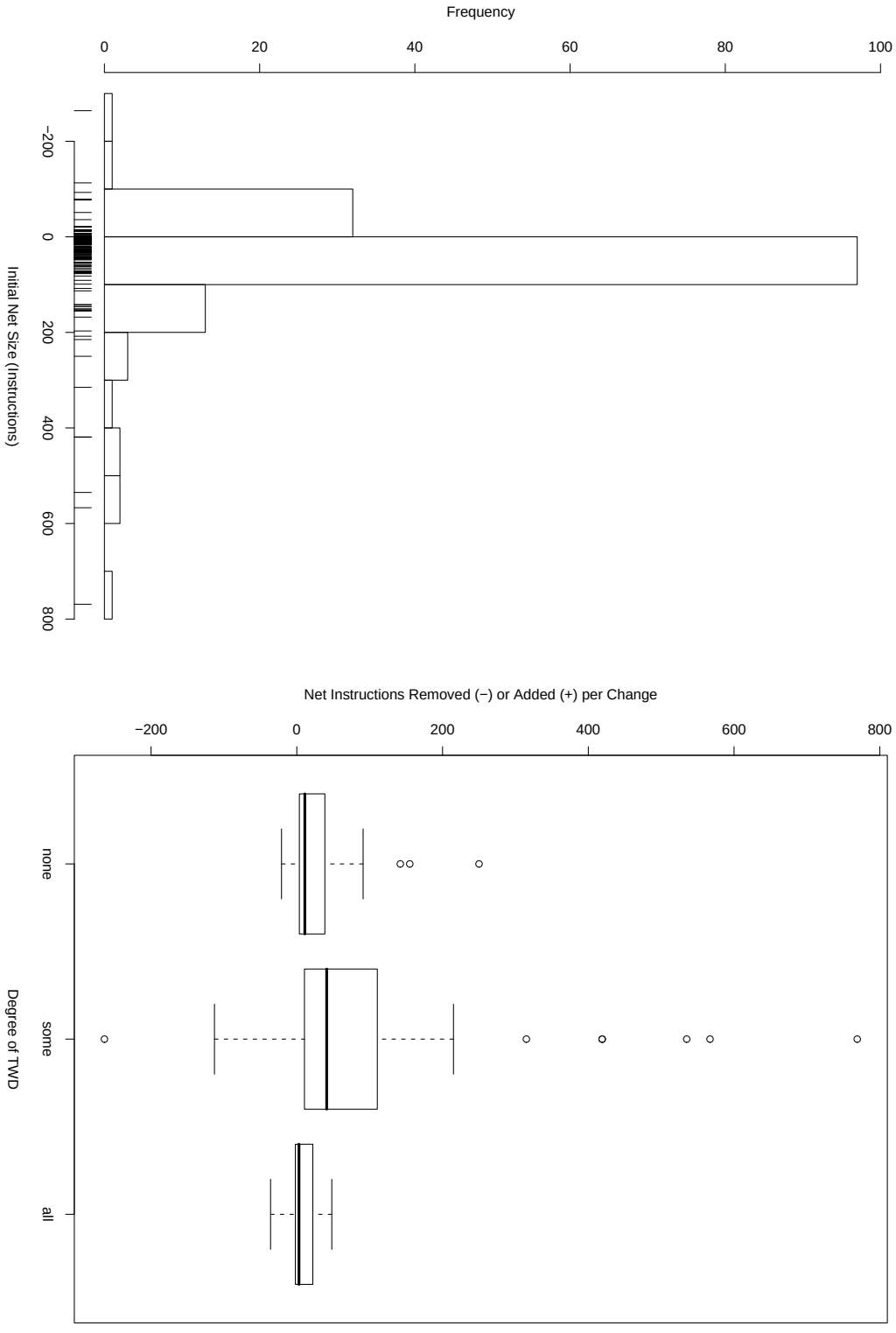


Figure F.13: ehcache—Histogram and boxplot for initial net size

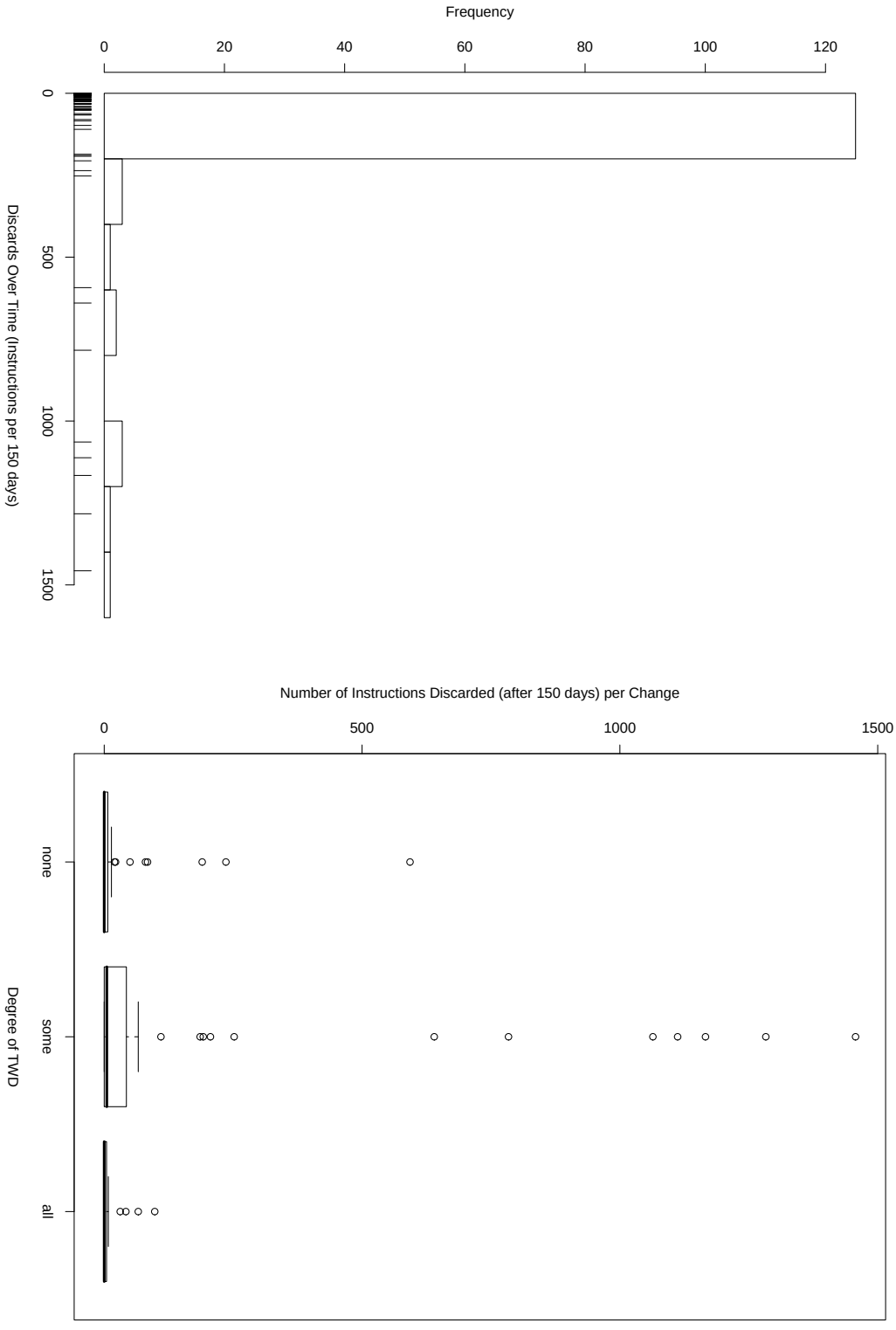


Figure F.14: ehcache—Histogram and boxplot for discards over time

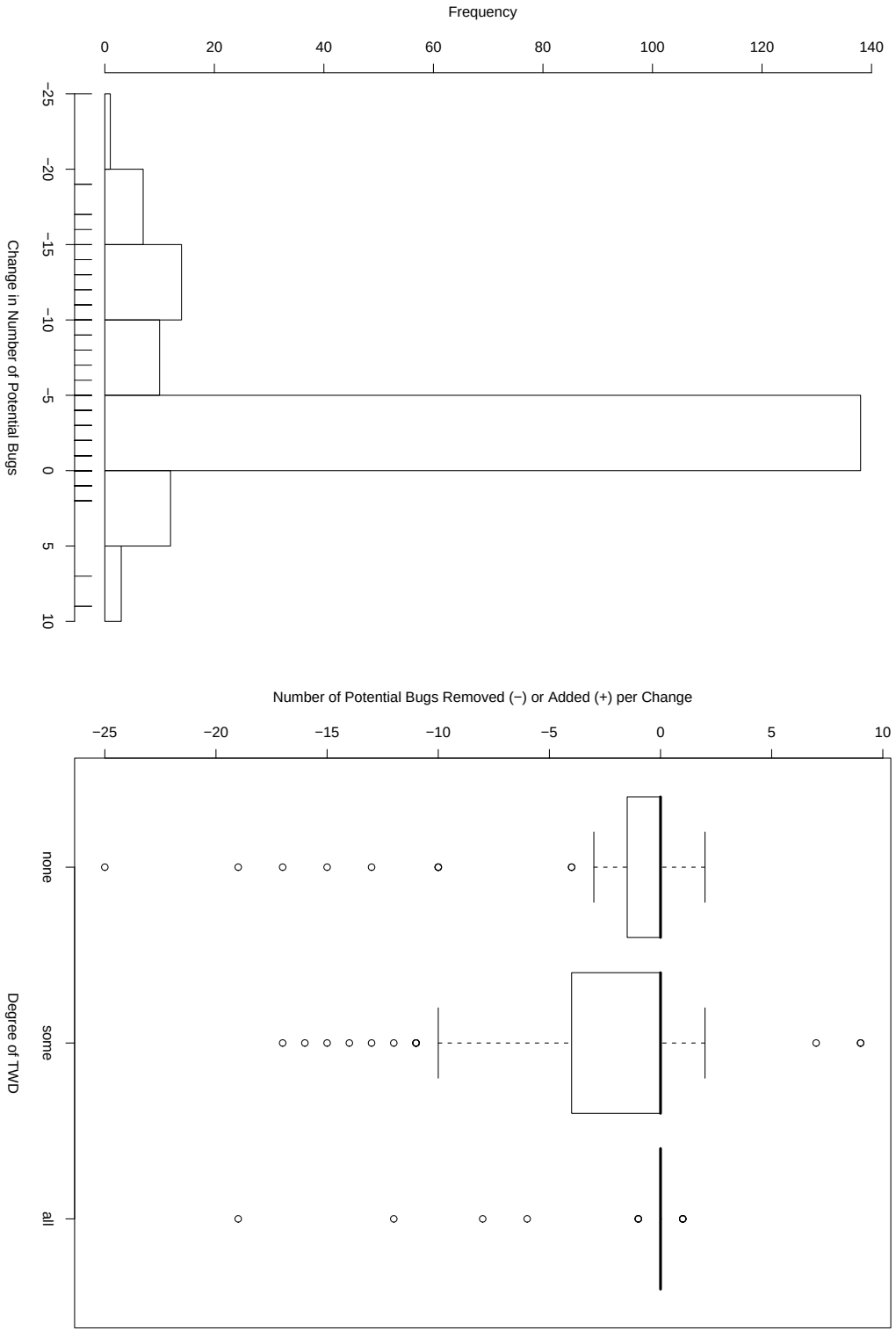


Figure F.15: ehcache—Histogram and boxplot for changes in potential bugs

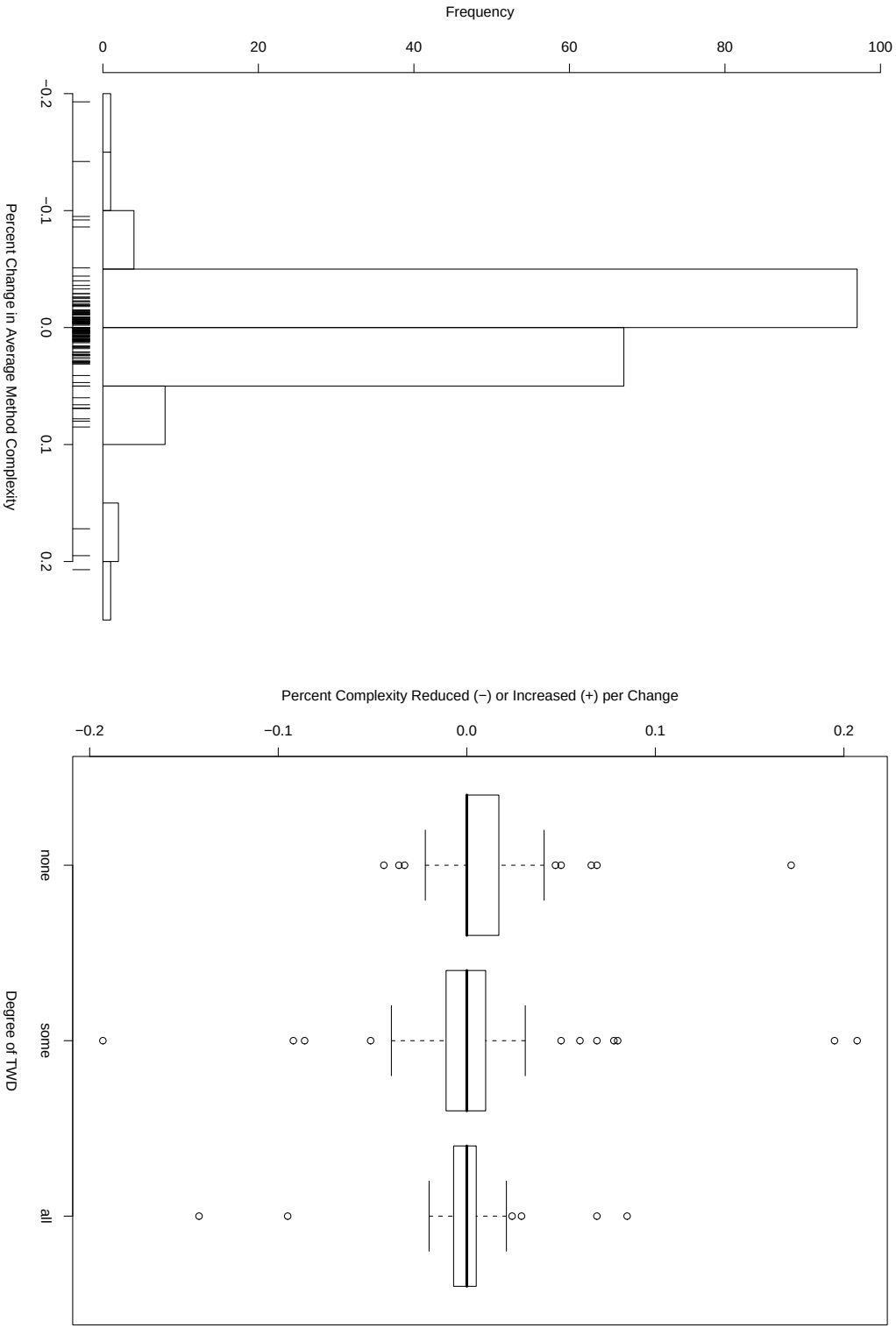


Figure F.16: ehcache—Histogram and boxplot for changes in average method complexity

F.5 hadoop-core

Figure F.17 presents the overall distribution and segregated distributions of the initial net size measures. Figure F.18 presents the overall distribution and segregated distributions of the discards over time measures. Figure F.19 presents the overall distribution and segregated distributions of the change in potential bugs measures. Figure F.20 presents the overall distribution and segregated distributions of the change in average method complexity measures.

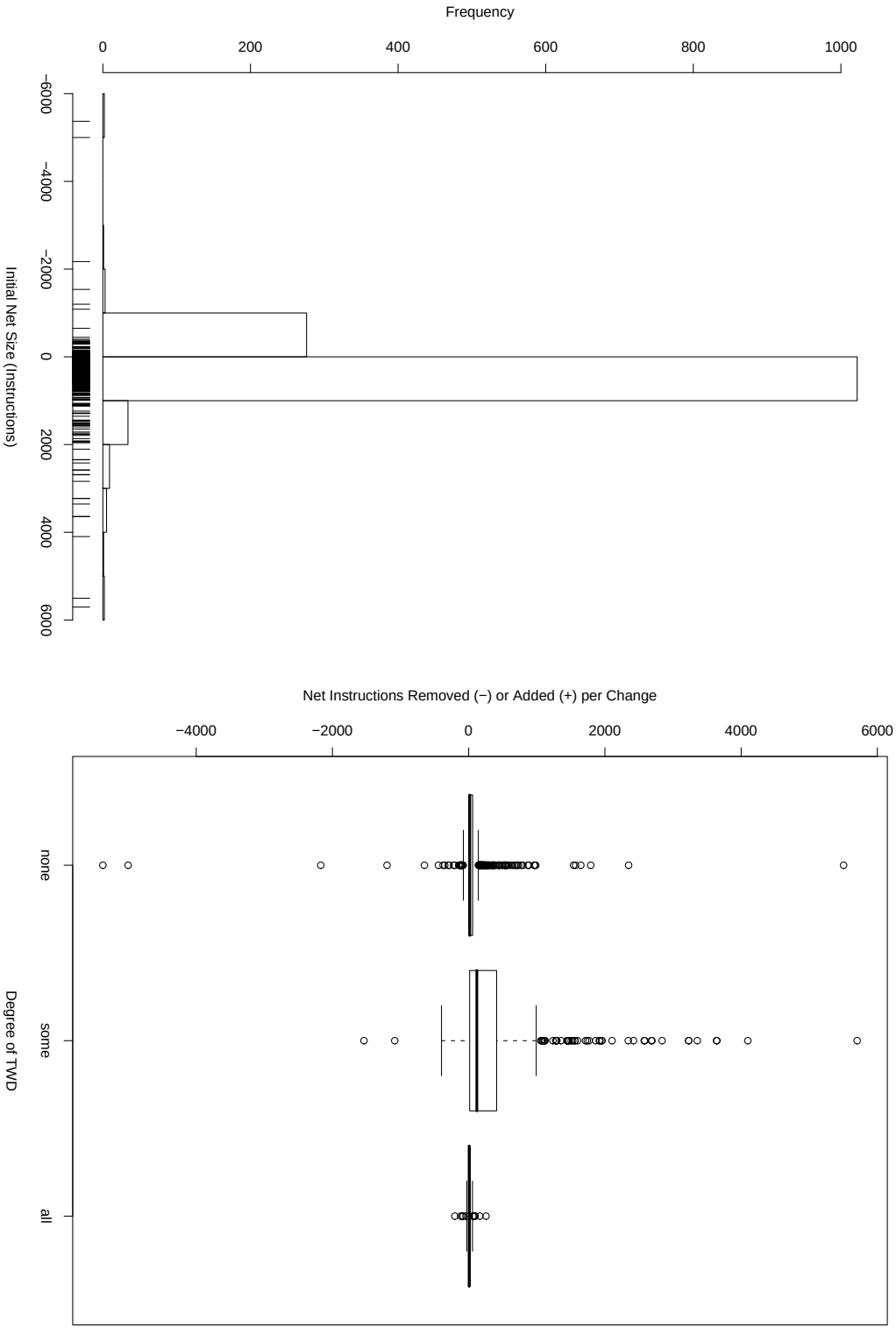


Figure F.17: hadoop-core—Histogram and boxplot for initial net size

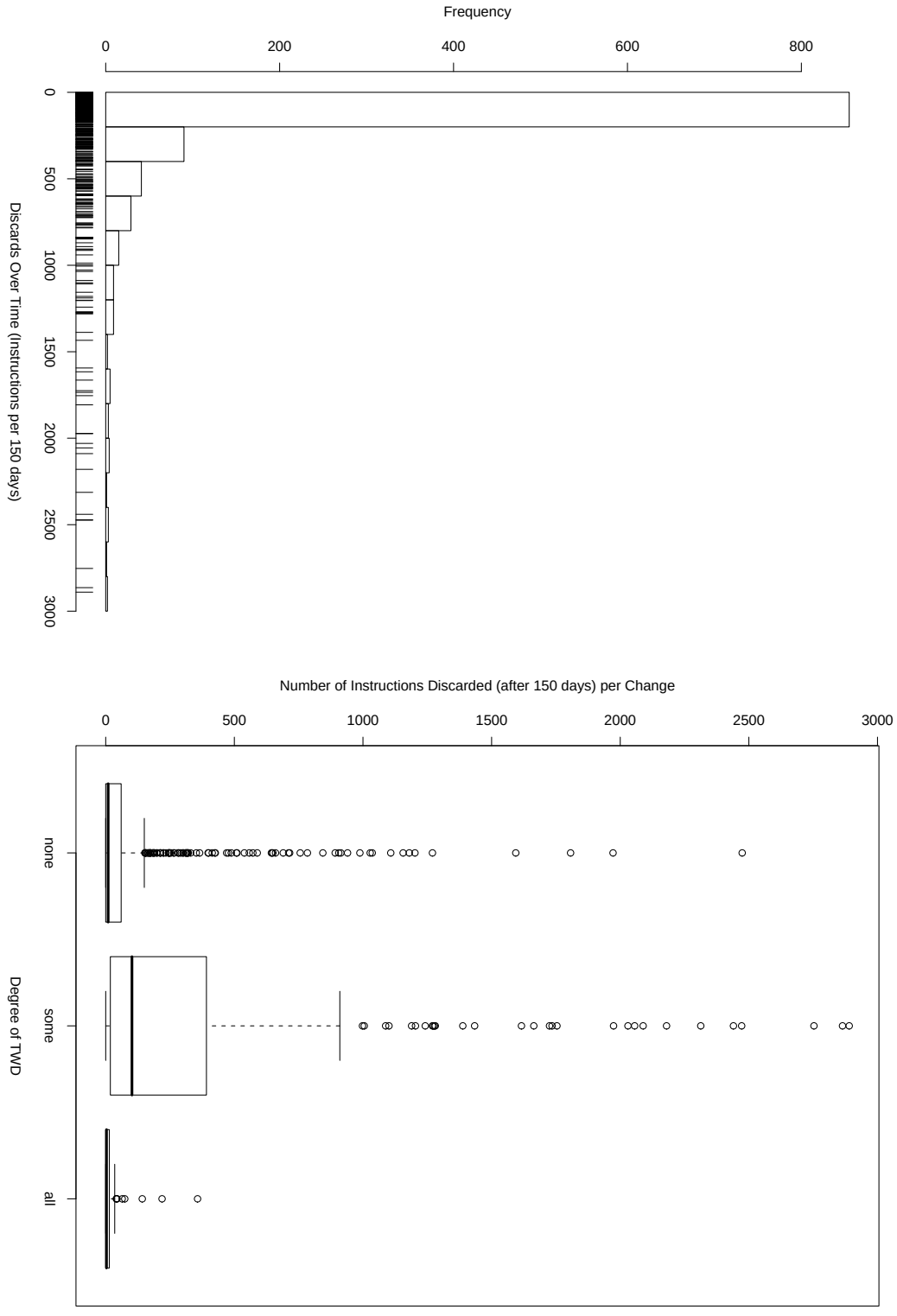


Figure F.18: hadoop-core—Histogram and boxplot for discards over time

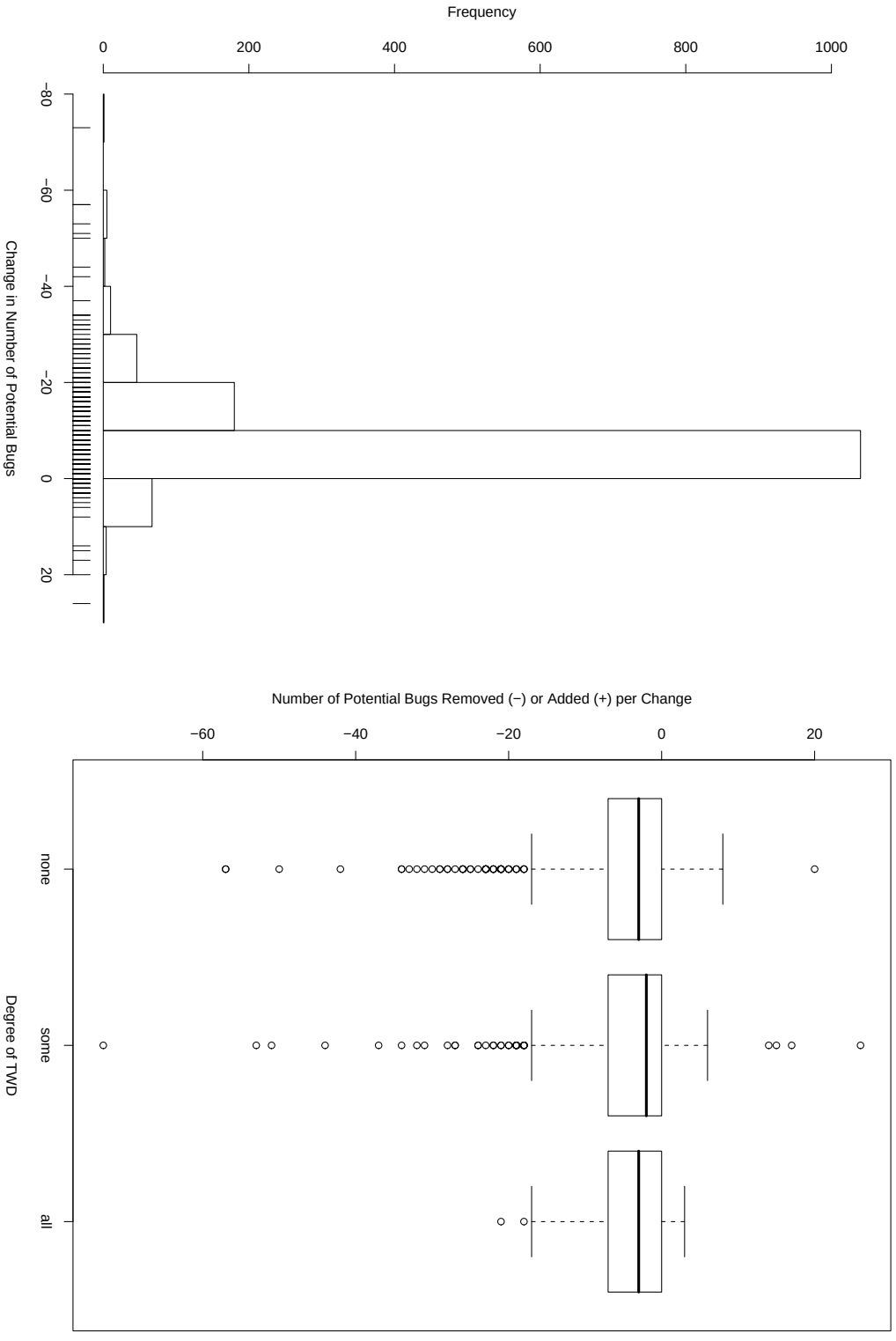


Figure F.19: hadoop-core—Histogram and boxplot for changes in potential bugs

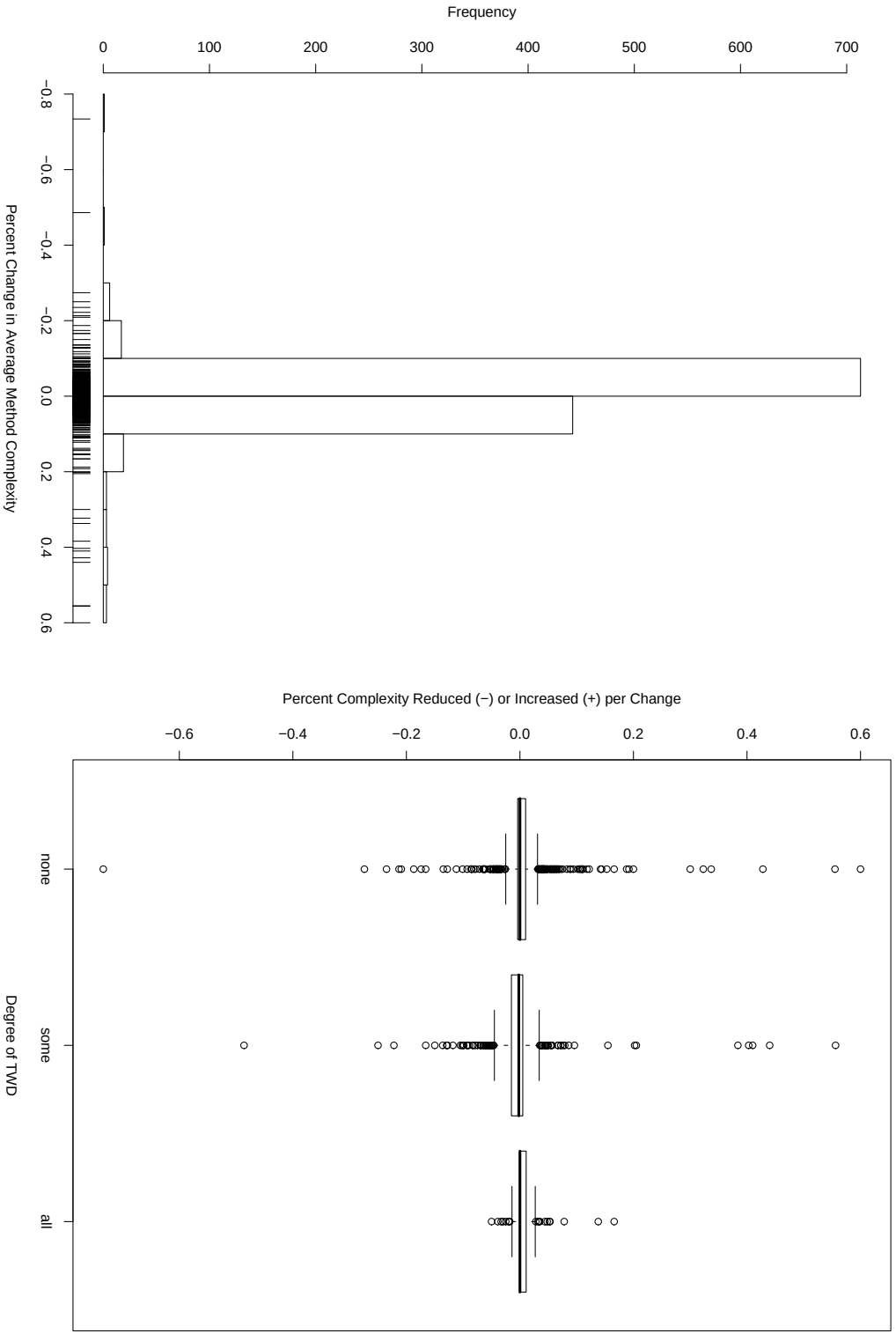


Figure F.20: hadoop-core—Histogram and boxplot for changes in average method complexity

F.6 xstream

Figure F.21 presents the overall distribution and segregated distributions of the initial net size measures. Figure F.22 presents the overall distribution and segregated distributions of the discards over time measures. Figure F.23 presents the overall distribution and segregated distributions of the change in potential bugs measures. Figure F.24 presents the overall distribution and segregated distributions of the change in average method complexity measures.

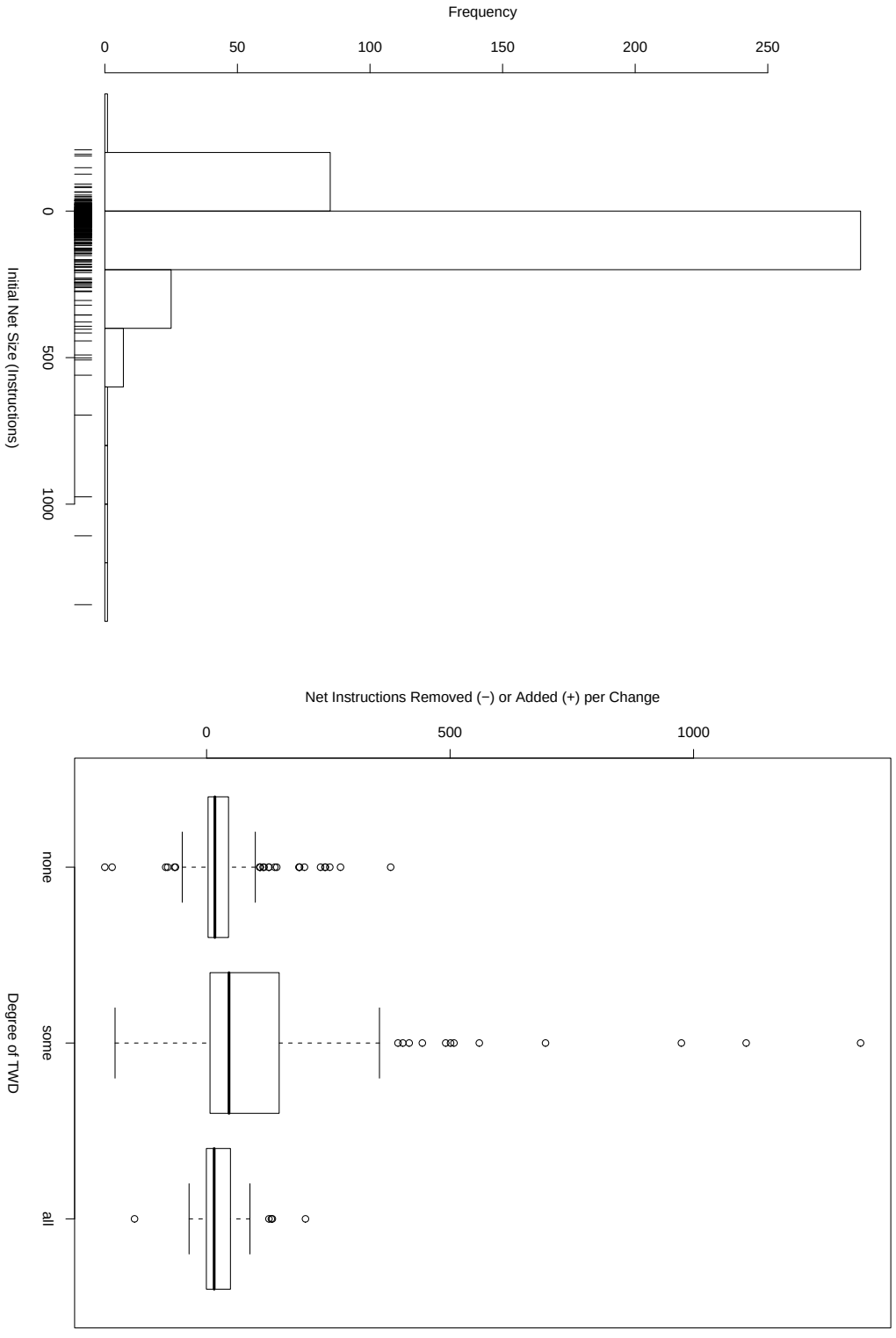


Figure F.21: xstream—Histogram and boxplot for initial net size

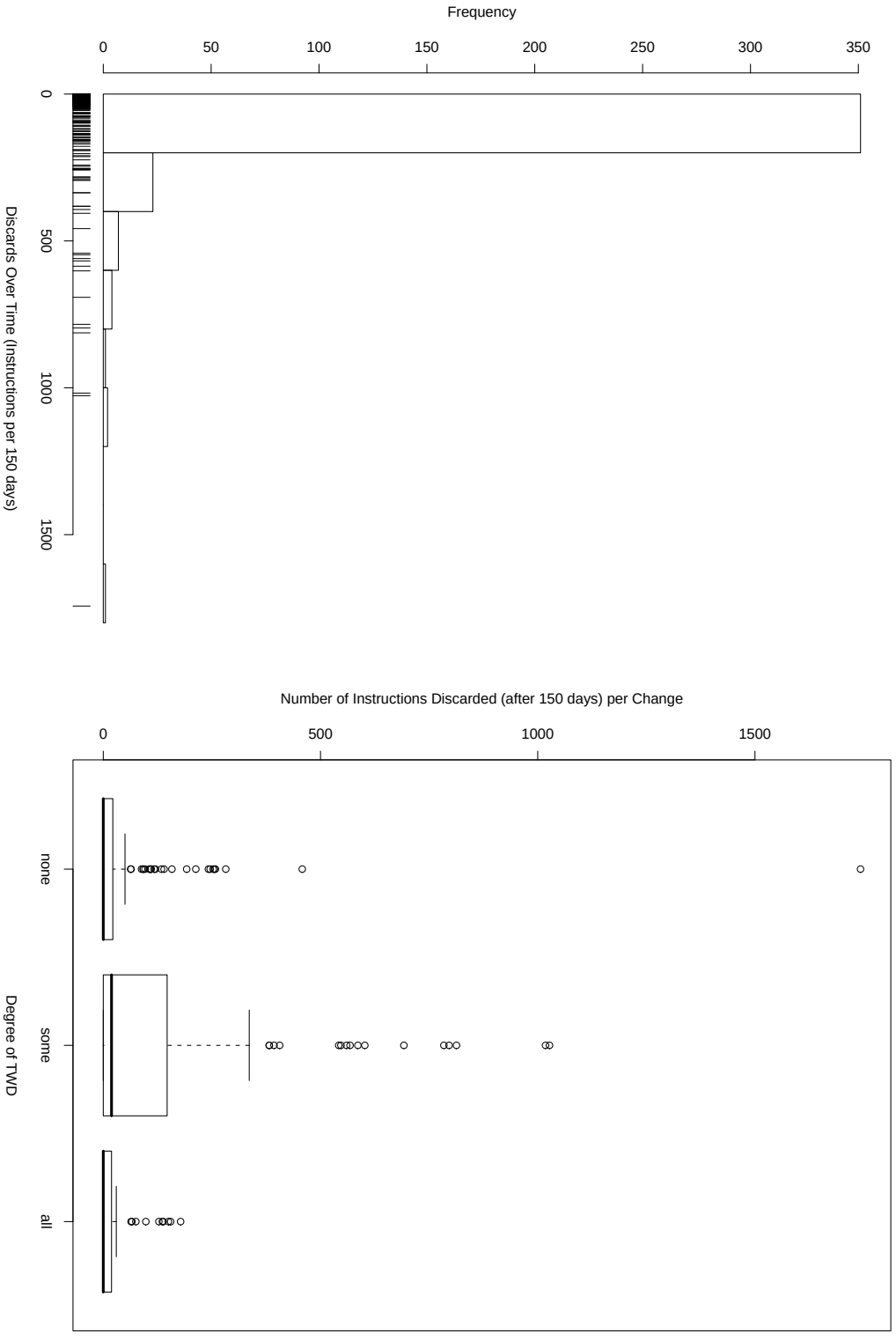


Figure F.22: xstream—Histogram and boxplot for discards over time

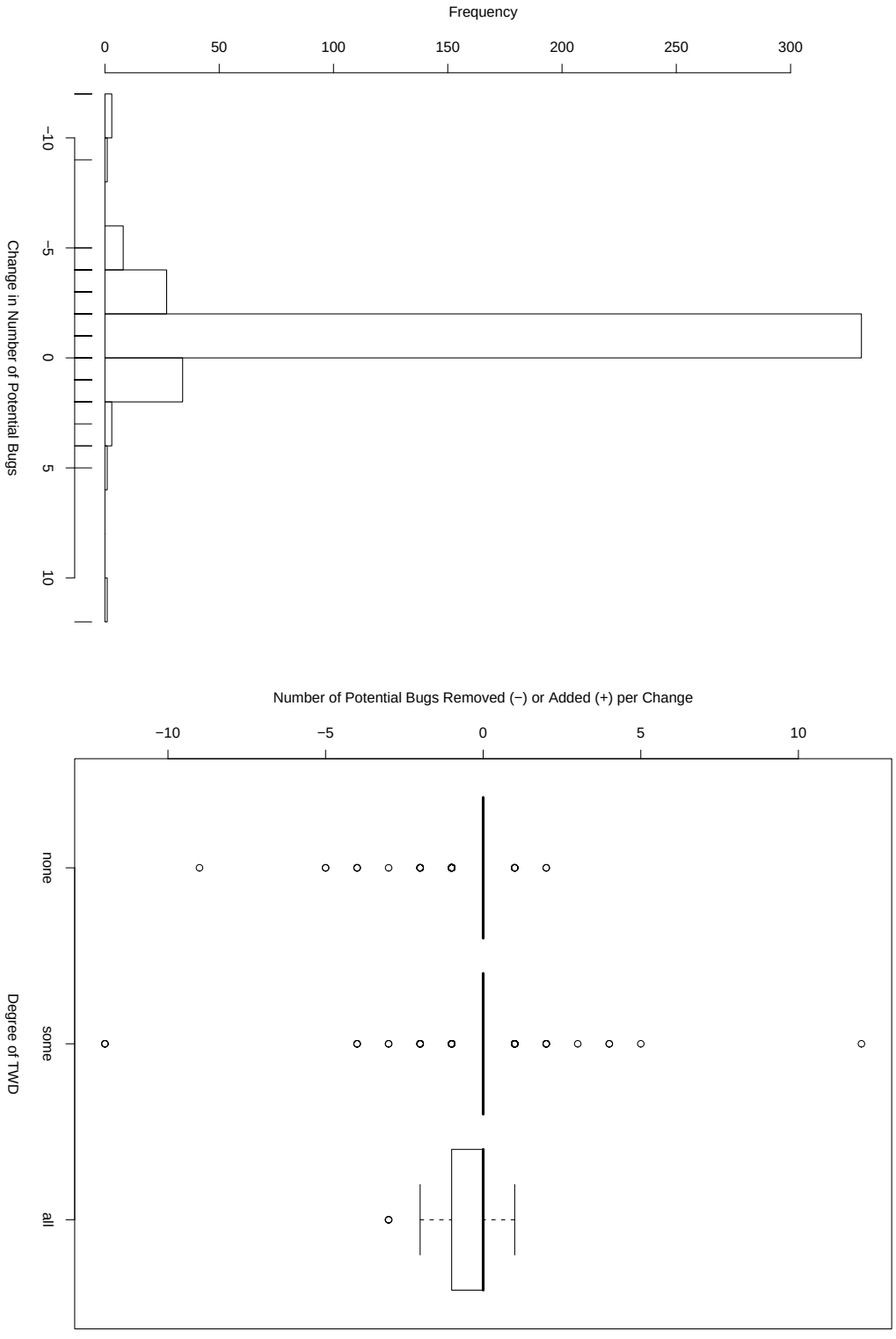


Figure F.23: xstream—Histogram and boxplot for changes in potential bugs

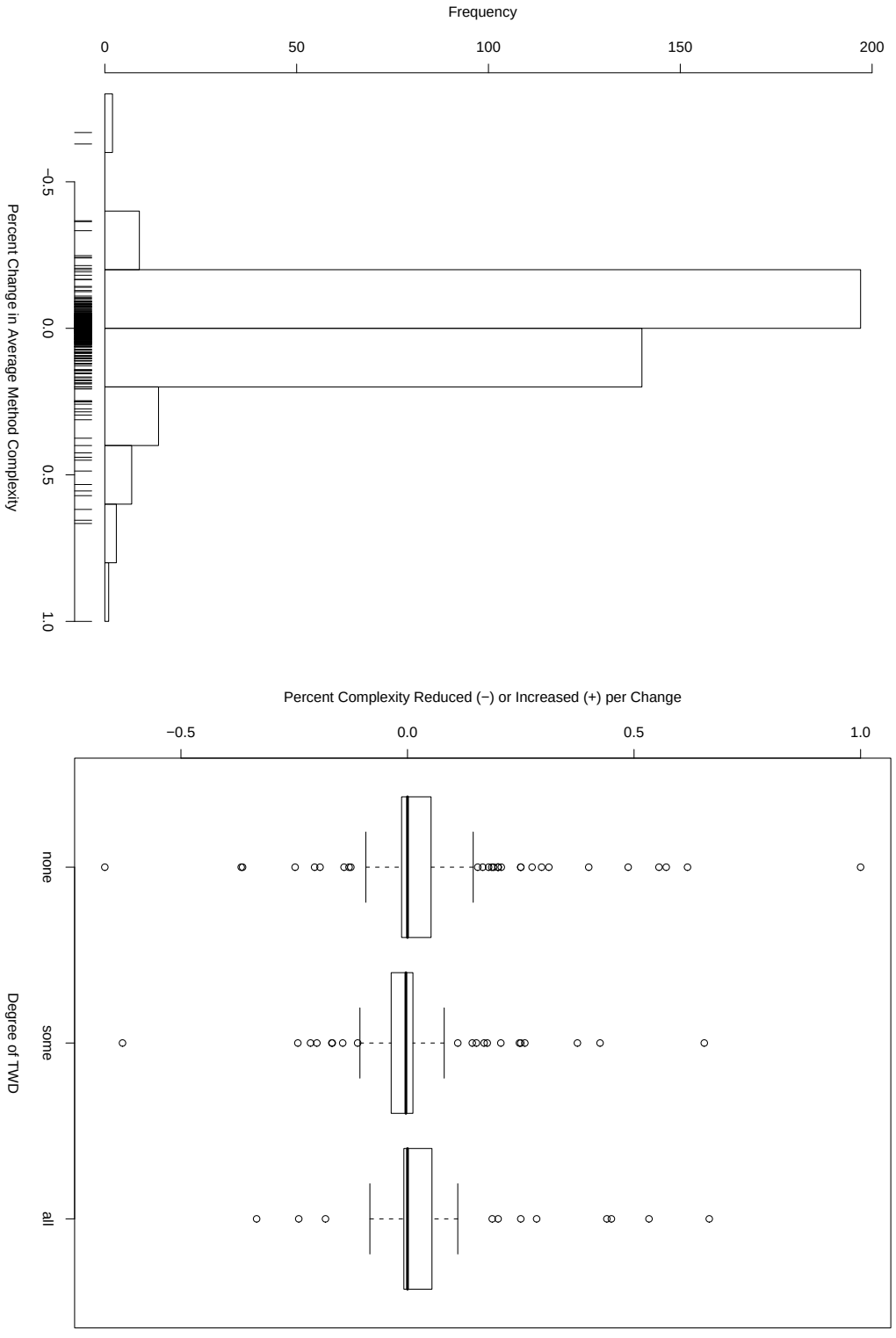


Figure F.24: xstream—Histogram and boxplot for changes in average method complexity

Appendix G

Electronic Resources

A set of electronic resources is also available with this thesis. There are three main types of resource: product, measurement, and analysis. The product resources include the product revisions for all of the main study cases and all of the scripts that were used to build them. The measurement resources include the measurement scripts and the resulting correlated measurements. Finally, the analysis resources include the scripts that were used to analyse the correlated measurements and to generate .ps, .pdf, or Latex output.

G.1 Product Resources

Directory	*Size*	*Description*
bin	250 KB	scripts to build a revision, revisions, month, months
etc	11.9 MB	case specific build scripts for ranges of revisions
util	400 KB	scripts to prepare for builds

Table G.1: Scripts to build product revisions

Directory	*Size*	*Description*
svn	24 MB	Subversion information for each case
src	770 MB	Source directory for each case (latest only)
dst	17.8 GB	Compiled products and product-tests for each case

Table G.2: Product revisions

G.2 Measurement Resources

Directory	*Size*	*Description*
bin	2.6 MB	jeanda compiled classes
lib	2.5 MB	jeanda dependency libraries
java	300 KB	shell scripts to utilize jeanda
overall	2.8 MB	shell scripts to measure/correlate cases

Table G.3: Scripts to measure product revisions

Directory	*Size*	*Description*
dotwd	180 MB	Degree of TWD measurements
speed-size	382 MB	Net size measurements
speed-discard	50 MB	Discard measurements
quality-findbugs	880 MB	Bugs measurements
quality-cyvis	1.1 GB	Complexity measurements

Table G.4: Raw product revision measurements

G.3 Analysis Resources

Directory	*Size*	*Description*
bin	150 KB	filter, analyse data and generate .ps, .pdf, .tex
template	360 KB	template scripts for analyzing each case

Table G.5: Scripts to analyse product revision measurements

Directory	*Size*	*Description*
ant	2.5 MB	ant correlated analyses
cayenne	1.7 MB	cayenne correlated analyses
commons-codec	472 KB	commons-codec correlated analyses
ehcache	528 KB	ehcache correlated analyses
hadoop-core	1.4 MB	hadoop-core correlated analyses
xstream	672 KB	xstream correlated analyses

Table G.6: Correlated product revision analyses

Glossary

Average Method Complexity The average of the method complexities, as measured by cyvis, for the methods involved in a developer's change to their product. We measure this value before and after the change.

Degree of TWD The degree to which a developer has practised TWD, as measured by jeanda, when they made a change to their product. For our present work, the possible degrees are: none, some, or all.

Discards over Time The fraction of a developer's change to their product, as measured by jeanda, that is discarded over time. For our present work, we measured the discarded fraction after 150 days.

Initial Net Size The net size of a developer's change to their product, as measured by jeanda, when they initially make the change and before any of it is discarded.

Net Size over Time The net size of a developer's change to their product, as measured by jeanda, after a fraction of the change has been discarded over time.

Number of Potential Bugs The number of potential bugs, as measured by findbugs, for the methods involved in a developer's change to their product. We measure this number before and after the change.

Software Change Model Our software change model uses set-theory to define the degree of TWD, the initial net size, the discards over time, and the net size over time for a developer's change. As part of the model, we also adopted definitions for potential bugs and average method complexity.

TDD Test-Driven Development is a practise, within the Extreme Programming Method, where developers write automated product tests for a fine-grained feature, ensure they fail, and then add to or change their product until the tests pass.

TFD Test-First Development is a step, within the TDD practise, where developers write automated product tests for a fine-grained feature *before* they add to or change their product.

TLD Test-Last Development is the traditional step where developers (or specialised testers) write automated or manual product tests for large-grained features, after the developers add to or change their product and share their changes, if they ever write product tests at all.

TWD Test-With Development is a more general step than TFD, as it includes TFD, and as it allows that developers may write automated product tests for fine-grained features while or after they add to or change their product, as long as they do so before they share their changes.

Bibliography

Abrahamsson, P., Hanhineva, A. & Jaalinoja, J. (2005), Improving business agility through technical solutions: A case study on test-driven development in mobile software development, *in* 'IFIP 2005: Business Agility and Information Technology Diffusion', pp. 1–17.

Ambler, S. (2002), *Agile Modeling*, J. Wiley, London.

Archer, T. (2001), *Inside C*, Microsoft Press, Redmond.

Auer, K. & Miller, R. (2002), *Extreme Programming Applied*, Addison-Wesley, Boston.

Ayewah, N., Hovemeyer, D., Morgenthaler, J. D., Penix, J. & Pugh, W. (2008), 'Using static analysis to find bugs', *IEEE Software* **25**(5), 22–29.

Bannerman, S. & Martin, A. (2010), 'A multiple comparative study of test-with development product changes and their effects on team speed and product quality', *Empirical Software Engineering* pp. 1–34. 10.1007/s10664-010-9137-5.

URL: *http://dx.doi.org/10.1007/s10664-010-9137-5*

Basili, V. R., Selby, R. W. & Hutchens, D. H. (1986), 'Experimentation in software engineering', *IEEE Transactions on Software Engineering* **12**(7), 733–743.

Basili, V. R., Shull, F. & Lanubile, F. (1999), 'Building knowledge through fam-

ilies of experiments', *IEEE Transactions on Software Engineering* **25**(4), 456–473.

Basili, V. R. & Turner, A. J. (1975), 'Iterative enhancement: A practical technique for software development', *IEEE Transactions on Software Engineering* **1**(4), 390–396.

Beck, K. (1999*a*), 'Embracing change with extreme programming', *IEEE Computer* **32**(10), 70–77.

Beck, K. (1999*b*), *Extreme Programming Explained: Embrace Change*, Addison-Wesley.

Beck, K. (2001), 'Aim, fire', *IEEE Software* **18**(5), 87–89.

Beck, K. (2003), *Test-Driven Development*, Addison-Wesley, Boston.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J. & Thomas, D. (2001), 'Manifesto for agile software development'.
URL: *http://agilemanifesto.org*

Benbasat, I., Goldstein, D. K. & Mead, M. (1987), 'The case research strategy in studies of information systems', *JSTOR MIS Quarterly* **11**(3), 369–386.

Benington, H. D. (1983), 'Production of large computer programs', *IEEE Annals of the History of Computing* **05**(4), 350–361.

Benington, H. D. (1987), Production of large computer programs, *in* 'ICSE '87: Proceedings of the 9th international conference on Software Engineering', IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 299–310.

Bhat, T. & Nagappan, N. (2006), Evaluating the efficacy of test-driven development: Industrial case studies, *in* 'ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering', ACM, New York, NY, USA, pp. 356–363.

Boehm, B. (1981), *Software Engineering Economics*, Prentice Hall, Englewood Cliffs.

Boehm, B. (1986), 'A spiral model of software development and enhancement', *SIGSOFT Software Engineering Notes* **11**(4), 14–24.

Boehm, B. W. (1988), 'A spiral model of software development and enhancement', *IEEE Computer* **21**(5), 61–72.

Boehm, B. W., Abts, C., Brown, A. W., Chulani, S., Clark, B. K., Horowitz, E., Madachy, R., Reifer, D. J. & Steece, B. (2000), *Software Cost Estimation with Cocomo II*, Prentice Hall, Englewood Cliffs.

Böhm, C. & Jacopini, G. (1966), 'Flow diagrams, turing machines and languages with only two formation rules', *Communications of the ACM* **9**(5), 366–371.

Canfora, G., Cimitile, A., Garcia, F., Piattini, M. & Visaggio, C. A. (2006), Evaluating advantages of test driven development: a controlled experiment with professionals, *in* 'ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering', ACM, New York, NY, USA, pp. 364–371.

Card, D. N., McGarry, F. E. & Page, G. T. (1987), 'Evaluating software engineering technologies', *IEEE Transactions on Software Engineering* **13**(7), 845–851.

Champy, J. (1996), *Reengineering Management*, HarperBusiness, New York.

Cobb, R. H. & Mills, H. D. (1990), 'Engineering software under statistical quality control', *IEEE Software* **7**(6), 44–54.

Cook, J. E., Votta, L. G. & Wolf, A. L. (1998), 'Cost-effective analysis of in-place software processes', *IEEE Transactions on Software Engineering* **24**(8), 650–663.

Damm, L.-O. & Lundberg, L. (2006), 'Results from introducing component-level test automation and test-driven development', *J. Syst. Softw.* **79**(7), 1001–1014.

Davenport, T. H. & Short, J. E. (1990), 'The new industrial engineering: Information technology and business process redesign', *MIT Sloan Management Review* **31**(4), 11–27.

Davies, J. & Woodcock, J. (1996), *Using Z: Specification, Refinement, and Proof*, Prentice Hall.

Degrace, P. & Stahl, L. H. (1990), *Wicked Problems, Righteous Solutions*, Yourdon Press, Englewood Cliffs.

Deming, W. (1986), *Out of the Crisis*, Massachusetts Institute of Technology, Center for Advanced Engineering Study, Cambridge.

Diehl, S., Gall, H., Pinzger, M. & Hassan, A. E. (2006), Msr 2006: the 3rd international workshop on mining software repositories, *in* 'ICSE '06: Proceedings of the 28th international conference on Software engineering', ACM, New York, NY, USA, pp. 1021–1021.

Diehl, S., Hassan, A. E. & Holt, R. C. (2005), 'Report on msr 2005: international workshop on mining software repositories', *SIGSOFT Softw. Eng. Notes* **30**(5), 1–3.

Dijkstra, E. W. (1968), 'Letters to the editor. go to statement considered harmful', *Communications of the ACM* **11**(3), 147–148.

Duvall, P. (2007), *Continuous Integration: Improving Software Quality and Reducing Risk*, Addison-Wesley Professional, Reading.

Ellims, M., Bridges, J. & Ince, D. C. (2004), Unit testing in practice, *in* 'ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering', IEEE Computer Society, Washington, DC, USA, pp. 3–13.

Ellims, M., Bridges, J. & Ince, D. C. (2006), 'The economics of unit testing', *Empirical Software Engineering* **11**(1), 5–31.

Emam, K. E. & Koru, A. G. (2008), 'A replicated survey of it software project failures', *IEEE Software* **25**(5), 84–90.

Emerson, H. (2003), *The Twelve Principles of Efficiency*, Kessinger Publishing.

Erdogmus, H., Morisio, M. & Torchiano, M. (2005), 'On the effectiveness of the test-first approach to programming', *IEEE Transactions on Software Engineering* **31**(3), 226–237.

Fenton, N. & Pfleeger, S. (1997), *Software Metrics: A Rigorous and Practical Approach*, PWS Publishing Company, Boston.

Gabor, A. (1992), *The Man Who Discovered Quality: How W. Edwards Deming Brought the Quality Revolution to America*, Penguin Books, New York.

Gall, H., Lanza, M. & Zimmermann, T. (2007), 4th international workshop on mining software repositories (msr 2007), *in* 'ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering', IEEE Computer Society, Washington, DC, USA, pp. 107–108.

George, B. & Williams, L. (2003), An initial investigation of test driven development in industry, *in* 'SAC '03: Proceedings of the 2003 ACM symposium on Applied computing', ACM, New York, NY, USA, pp. 1135–1139.

Geras, A., Smith, M. & Miller, J. (2004), A prototype empirical evaluation of test driven development, *in* 'METRICS '04: Proceedings of the Software Metrics, 10th International Symposium', IEEE Computer Society, Washington, DC, USA, pp. 405–416.

Gilb, T. (1985), 'Evolutionary delivery versus the "waterfall model"', *SIGSOFT Software Engineering Notes* **10**(3), 49–61.

Goldberg, A. & Robson, D. (1989), *Smalltalk-80: The Language*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gosling, J. (2005), *The Java Language Specification*, Addison-Wesley, Boston.

Gupta, A. & Jalote, P. (2007), An experimental evaluation of the effectiveness and efficiency of the test driven development, *in* 'ESEM '07: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement', IEEE Computer Society, Washington, DC, USA, pp. 285–294.

Hammer, M. (1990), 'Reengineering work: Dont automate, obliterate', *Harvard Business Review* **68**(4), 104–112.

Hannay, J. E., Sjoberg, D. I. K. & Dyba, T. (2007), 'A systematic review of theory use in software engineering experiments', *IEEE Transactions on Software Engineering* **33**(2), 87–107.

Hassan, A. E., Holt, R. C. & Mockus, A. (2005), 'Report on msr 2004: International workshop on mining software repositories', *SIGSOFT Softw. Eng. Notes* **30**(1), 4.

Hindle, A., German, D. M. & Holt, R. (2008), What do large commits tell us? a taxonomical study of large commits, *in* 'MSR '08: Proceedings of the 2008 international working conference on Mining software repositories', ACM, New York, NY, USA, pp. 99–108.

Janzen, D. S. & Saiedian, H. (2006), On the influence of test-driven development on software design, *in* 'CSEET '06: Proceedings of the 19th Conference on Software Engineering Education & Training', IEEE Computer Society, Washington, DC, USA, pp. 141–148.

Jeffries, R., Anderson, A. & Hendrickson, C. (2001), *Extreme Programming Installed*, Addison-Wesley, Boston.

Juran, J. (1988), *Juran's Quality Control Handbook*, McGraw-Hill, New York.

Kernighan, B. (1988), *The C Programming Language*, Prentice Hall, Englewood Cliffs.

Kitchenham, B., Pfleeger, S., Pickard, L., Jones, P., Hoaglin, D., Emam, K. E. & Rosenberg, J. (2002), 'Preliminary guidelines for empirical research in software engineering', *IEEE Transactions on Software Engineering* **28**(8), 721–734.

Kitchenham, B., Pickard, L. & Pfleeger, S. L. (1995), 'Case studies for method and tool evaluation', *IEEE Software* **12**(4), 52–62.

Lanza, M., Godfrey, M. W. & Kim, S. (2008), Msr 2008 - 5th working conference on mining software repositories, *in* 'ICSE Companion '08: Companion of the 30th international conference on Software engineering', ACM, New York, NY, USA, pp. 1037–1038.

Larman, C. & Basili, V. R. (2003), 'Iterative and incremental development: A brief history', *IEEE Computer* **36**(6), 47–56.

Lethbridge, T. C., Sim, S. E. & Singer, J. (2005), 'Studying software engineers: Data collection techniques for software field studies', *Empirical Software Engineering* **10**(3), 311–341.

Madeyski, L. (2005), Preliminary analysis of the effects of pair programming and test-driven development on the external code quality, *in* 'Proceeding of the 2005 conference on Software Engineering: Evolution and Emerging Technologies', IOS Press, Amsterdam, The Netherlands, The Netherlands, pp. 113–123.

Malhotra, Y. (1998), 'Business process redesign: An overview', *IEEE Engineering Management Review* **26**(3).

Maximilien, E. M. & Williams, L. (2003), Assessing test-driven development at ibm, *in* 'ICSE '03: Proceedings of the 25th International Conference on Software Engineering', IEEE Computer Society, Washington, DC, USA, pp. 564–569.

McCabe, T. J. (1976), 'A complexity measure', *IEEE Transactions on Software Engineering* **2**(4), 308–320.

McConnell, S. (1993), *Code Complete*, Microsoft Press, Redmond.

McCracken, D. D. & Jackson, M. A. (1982), 'Lifecycle concept considered harmful', *SIGSOFT Software Engineering Notes* **7**(2), 29–32.

Mills, H. D. (1976), 'Software development', *IEEE Transactions on Software Engineering* **2**(4), 265–273.

Muller, M. & Hagner, O. (2002), 'Experiment about test-first programming', *Software, IEE Proceedings* **149**(5), 131–136.

Nagappan, N., Maximilien, E. M., Bhat, T. & Williams, L. (2008), 'Realizing quality improvement through test driven development: Results and experiences of four industrial teams', *Empirical Software Engineering* **13**(3), 289–302.

Naur, P. & Randell, B. (1969), Software engineering: Report of a conference sponsored by the nato science committee, garmisch, germany, 7-11 oct. 1968, *in* 'ICSE '68: Proceedings of the 1st International Conference on Software Engineering', Scientific Affairs Division, NATO, Brussels.

Nelson, D. (1992), *A Mental Revolution: Scientific Management since Taylor*, The Ohio State University Press.

Nygaard, K. & Dahl, O.-J. (1981), *The development of the SIMULA languages*, ACM, New York, NY, USA.

OMG (2007), *OMG Unified Modeling Language Specification*, OMG. Version 2.1.2.
URL: *http://www.omg.org/spec/UML/2.1.2*

O'Neill, D. (1980), 'The management of software engineering. part ii: Software engineering program', *IBM Systems Journal* **19**(4), 421–431.

Pancur, M., Ciglaric, M., Trampus, M. & Vidmar, T. (2003), Towards empirical evaluation of. test-driven development in a university environment, *in* 'EUROCON 2003: IEEE Region 8 Proceedings', Vol. 2, IEEE Press, pp. 83–86.

Parnas, D. L. (1972), 'On the criteria to be used in decomposing systems into modules', *Communications of the ACM* **15**(12), 1053–1058.

Parnas, D. L. & Clements, P. C. (1986), 'A rational design process: How and why to fake it', *IEEE Transactions on Software Engineering* **12**(2), 251–257.

Pesola, G. R. (2004), 'The reuse of subjects in clinical studies: Is a bias introduced? is there a threat to study validity?', *Journal of Asthma* **41**(3), 305–309.

URL: *http://informahealthcare.com/doi/abs/10.1081/JAS-120026087*

Pinsonneault, A. & Kraemer, K. L. (1993), 'Survey research methodology in management information systems: An assessment', *Journal of Management Information Systems* **10**(2), 75–105.

Randell, B. (1979), Software engineering in 1968, *in* 'ICSE '79: Proceedings of the 4th international conference on Software engineering', IEEE Press, Piscataway, NJ, USA, pp. 1–10.

Ratzinger, J., Sigmund, T. & Gall, H. C. (2008), On the relation of refactorings and software defect prediction, *in* 'MSR '08: Proceedings of the 2008 international working conference on Mining software repositories', ACM, New York, NY, USA, pp. 35–38.

Raymond, E. S. (1999), The magic cauldron, *in* 'The Cathedral & the Bazaar', O'Reilly & Associates, pp. 137–194.

Robson, C. (2002), *Real World Research*, Blackwell Publishers, Cambridge.

Royce, W. (1970), Managing the development of large software systems, *in* 'Proceedings of IEEE WESCON', pp. 1–9.

Royce, W. W. (1987), Managing the development of large software systems: Concepts and techniques, *in* 'ICSE '87: Proceedings of the 9th international conference on Software Engineering', IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 328–338.

Runeson, P. & Host, M. (2009), 'Guidelines for conducting and reporting case study research in software engineering', *Empirical Software Engineering* **14**(2), 131–164.

Sanchez, J. C., Williams, L. & Maximilien, E. M. (2007), On the sustained use of a test-driven development practice at ibm, *in* 'AGILE '07: Proceedings of the AGILE 2007', IEEE Computer Society, Washington, DC, USA, pp. 5–14.

Schwaber, K. & Beedle, M. (2001), *Agile Software Development with Scrum*, Prentice Hall PTR, Upper Saddle River, NJ, USA.

Sfetsos, P., Angelis, L. & Stamelos, I. (2006), 'Investigating the extreme programming system—an empirical study', *Empirical Software Engineering* **11**(2), 269–301.

Shaw, M. (1989), A twenty year retrospective of the nato software engineering conferences (panel session): remembrances of a graduate student, *in* 'ICSE '89: Proceedings of the 11th international conference on Software engineering', ACM, New York, NY, USA, pp. 99–100.

Shewhart, W. (1980), *Economic Control of Quality of Manufactured Product*, American Society for Quality Control, Milwaukee.

Shewhart, W. & Deming, W. E. (1986), *Statistical Method from the Viewpoint of Quality Control*, Dover Publishing.

Siniaalto, M. & Abrahamsson, P. (2007), A comparative case study on the impact of test-driven development on program design and test coverage, *in* 'ESEM '07: Proceedings of the First International Symposium on Empirical

Software Engineering and Measurement', IEEE Computer Society, Washington, DC, USA, pp. 275–284.

Sjoberg, D. I. K., Hannay, J. E., Hansen, O., Kampenes, V. B., Karahasanovic, A., Liborg, N. & Rekdal, A. C. (2005), 'A survey of controlled experiments in software engineering', *IEEE Transactions on Software Engineering* **31**(9), 733–753.

Stroustrup, B. (2000), *The C++ Programming Language*, Addison-Wesley, Boston.

Taylor, F. W. (1967), *The Principles of Scientific Management*, Norton, New York.

Williams, L., Maximilien, E. M. & Vouk, M. (2003), Test-driven development as a defect-reduction practice, *in* 'ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering', IEEE Computer Society, Washington, DC, USA, pp. 34–45.

Wirth, N. (1971), 'The programming language pascal', *Acta Informatica* **1**, 35–63.

Witzel, M. (2002), 'A short history of efficiency', *Business Strategy Review* **13**, 38–47.

Wood, M. C. & Wood, J. C. (2003), *Frank and Lillian Gilbreth: Critical Evaluations in Business and Management*, Routledge.

Wood, W. A. & Kleb, W. L. (2003), 'Exploring xp for scientific research', *IEEE Software* **20**(3), 30–36.

Zaidman, A., Van Rompaey, B., Demeyer, S. & van Deursen, A. (2008), Mining software repositories to study co-evolution of production & test code, *in* 'ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation', IEEE Computer Society, Washington, DC, USA, pp. 220–229.

Zurcher, F. W. & Randell, B. (1968), Iterative multi-level modeling - a methodology for computer system design, *in* 'IFIP Congress (2)', pp. 867–871.