Computing Science Group

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

Steve Bannerman and Andrew Martin

CS-RR-10-03

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford, OX1 3QD

Oxford University Computing Laboratory — Software Engineering
Technical Report: CS-RR-10-03

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

Steve Bannerman · Andrew Martin

Submitted: April 2010

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 define 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 this 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 simplify their product can benefit from practicing TWD.

Keywords Multiple Comparative Study · Test-With Development · Team Speed · Product Quality

1 Introduction

Over the last decade, Kent Beck and his colleagues have pioneered the Extreme Programming (XP) Method, which is an Agile Method within the Agile Model family. 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, 1999a). By taking best practices to extreme levels, teams

Steve Bannerman
University of Oxford, Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK
E-mail: steve.bannerman@comlab.ox.ac.uk

Andrew Martin
University of Oxford, Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK
E-mail: andrew.martin@comlab.ox.ac.uk

should be able to develop and deliver their product to their customers more quickly and at higher levels of quality (Beck, 1999b).

Test-Driven Development (TDD) is one of XP's best practices, and possibly *the* key practice (Beck, 1999a; Jeffries et al, 2001; Auer and Miller, 2002). When a team practices 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 automated product tests before they change their product, they practice the Test-First Development (TFD) step within TDD and XP.

In principle, teams executing TFD could develop their product more quickly and at a higher level of quality—it could focus them, removing excess from their activities and helping them leverage automated test assets to improve their internal and external product quality. This hypothesis 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 practicing 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 practicing TFD was the cause of, or correlated with, slightly decreased speed and slightly or moderately increased quality. To date, the relevant research has assumed that a team either practices TFD or they do not, rather than considering that a team may practice it to a degree. We present an overview of the TFD research in Section 2.

1.1 Test-With Development

When practicing TFD, developers evolve their automated product tests before making their fine-grained product changes (Beck, 2003), in contrast to practicing 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 practicing TLD, developers evolve most or all of their automated product tests after making a series of product changes, if they evolve their automated 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 and others 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 and Williams, 2003; Pancur et al, 2003; Erdogmus et al, 2005; Janzen and Saiedian, 2006; Sanchez et al, 2007). So,

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 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 recognize 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. To reinforce our definition of TWD, and to introduce some ideas that are pertinent to our research, we present a product development example in Section 3.

1.2 Multiple Comparative Study

Therefore, given the potential benefits of TWD, assumed from the potential benefits of the TDD practice and the TFD step, and some limitations of the TFD research, we were motivated to begin to answer this question: *What are the relationships between practicing TWD, to varying degrees, and some attributes related to team speed and some attributes related to product quality, when those practicing it are real-life software development teams, developing their product over the course of many years*?

If the effects on the attributes related to team speed and product quality are significantly different, depending on the degree to which a team has practiced TWD, then future teams may want to practice it (or not) and may want to practice it to a certain degree. Ultimately, knowledge of the differences could allow teams to work more quickly, or to deliver products of higher quality, or both. Given the resources that we allocate to buying and developing software, and the importance of high quality software products to society, this is an important question.

As a result of our motivation to begin answering this question, we conducted a multiple comparative study to explore and describe these relationships. We rebuilt and studied the product and automated product test revisions of six long-term open-source projects and then performed statistical analyses on the resulting data to compare the differences between the effects of practicing TWD. We elaborate on our motivation, our plan, and our protocol for this study in Section 4. Then, we present and discuss its results in Section 5.

In summary, our results indicate that, on average, the degree of TWD that developers practiced when making product changes resulted in some significant differences between the effects of those changes. In Section 6, we present our conclusions, compare them with others' prior research results, discuss their implications and limitations, and identify potential future work.

2 Test-First Development Research

Although several researchers have performed studies to investigate the effects of TFD on, or correlations of TFD with, specific attributes of team speed or product quality, the results of their studies have been mixed. Some have found that attributes of team speed increase or stay roughly the same when individuals or a team execute TFD. Others have found that attributes of team speed decrease, if only slightly. However, some have not found any effect on attributes of team speed. Further, some have found that attributes of either the internal or external product quality increase, to a varying degree. Finally, some have been unable to determine the effect of TFD on attributes of product quality.

We have summarized their studies and results in four tables. Table 1 summarizes the studies where students were their subjects. Table 2 summarizes the studies where professionals were their subjects. Table 3 summarizes the attributes of team speed and product quality that they analyzed in each study. Table 4 summarizes the development durations for each study's subjects. We refer the reader to the relevant publications for more specific details on these studies.

Publication	*Type*	*Speed*	*Quality*
Muller and 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 and Saiedian (2006)	Controlled	Faster	No difference
Gupta and Jalote (2007)	Controlled	Faster	No difference

Table 1: Effects of Test-First development–Students

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

Table 2: Effects of/Correlations with Test-First development–Professionals

Publication	*Team Speed*	*Product Quality*
Muller and Hagner (2002)	total time	reliability
George and Williams (2003)	productivity (time)	acceptance tests
Maximilien and 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 and Nagappan (2006)	time estimates	defect density
Canfora et al (2006)	time to complete	–
Damm and Lundberg (2006)	–	fault rate
Janzen and Saiedian (2006)	total effort	internal metrics
Gupta and Jalote (2007)	effort, productivity	acceptance tests passed
Sanchez et al (2007)	–	defect density, complexity
Siniaalto and Abrahamsson (2007)	–	program design, code coverage
Nagappan et al (2008)	time estimates	defect density

Table 3: Studied Attributes of Team Speed and Product Quality

Publication	*Development Duration*
Muller and Hagner (2002)	2 months
George and Williams (2003)	1 day (or less)
Maximilien and 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 and Nagappan (2006)	7-12 months
Canfora et al (2006)	1 day (or less)
Damm and Lundberg (2006)	12-18 months
Janzen and Saiedian (2006)	1 day (or less)
Gupta and Jalote (2007)	3 weeks
Sanchez et al (2007)	5 years
Siniaalto and Abrahamsson (2007)	9 weeks
Nagappan et al (2008)	7-12 months

Table 4: Development Durations for Study Subjects

Overall, 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, we noticed a few things about these studies that we wished to improve upon in our own. Most of these studies have been relatively short in duration: days and weeks. And, all of them have assumed that a team practices TFD or that they do not, rather than considering that a team may practice it to a degree. 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.

While planning and executing our study, we sought to improve upon these limitations. We studied projects and their products, after they had been completed, which were developed over the course of months and years, rather than studying projects in a controlled environment, or in their context, over the course of a day or weeks. As well, we have observed that developers and teams practice TWD (which includes TFD), and that they practice it to varying degrees. And, we assume that within a project, the confounding effects of contextual variables (Basili et al, 1986, 1999) will be dampened, therefore allowing for more reasonable comparisons. That is, within a project, its product, its process, its technology, and to a certain extent its people will vary less than they would across two or more similar projects.

3 Product Development Example

In this section, we present a product development example to introduce our software change model, or our frame of reference, which we developed to support our empirical study. The model itself, which we do not include in this section because of its length, is presented in Section A.

Consider a contrived software product, written in Java. Further, consider a single product class within that product, Amount, which could be used to model the attributes and behavior for an amount of money. Listing 1 shows a single product method for this class.

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

Listing 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.

Now, assuming the team is maintaining an automated suite of product tests, a test class named AmountTest might look like Listing 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" scenario.

```
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 2: Product Tests Revision 1

Because this test method will exercise the product method when it is compiled and executed, we consider the product method to be exercised by the product tests. Further, if the team is practicing 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.

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).

```
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: Product Revision 2

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 practiced TWD to a degree.

In making the changes from product revision 1 (Listing 1) to product revision 2 (Listing 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.

Finally, 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 4).

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

```
    return new Amount(getValue() + exchangeValue);
  }
}
```

Listing 4: Product Revision N

Now, if we reconsider the Amount class at revision 1 (Listing 1), we notice that the signature of the "add" method has remained unchanged, with respect to revision N (Listing 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.

Similarly, if we reconsider the Amount class at revision 2 (Listing 3), we notice that the team has deleted the second method, with respect to revision N (Listing 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.

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: its number of potential bugs and its average method complexity. Findbugs is a static analysis tool that can be executed against a compiled class to determine the number of potential bugs within it—this tool does not find the number of defects that will actually be encountered by somebody executing the product; rather, it finds a number of potential defects within the product. Similarly, cyvis is a static analysis tool that can be executed against a compiled class to determine the cyclomatic complexity for each of its methods.

4 Study Design

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

4.1 Motivation

In addition to being motivated to study the potential benefits of TWD (Section 1), and being motivated to improve upon some of the limitations of previous TFD studies (Section 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 organizations 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).

4.2 Study Plan

4.2.1 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 and 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.

4.2.2 Research Question

Our research question, as introduced in Section 1, is: *What are the relationships between practicing 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 practicing 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 Section 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.

4.2.3 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—in this precise set-theory model, we characterize changes to product and automated product test classes, methods, and compiled instructions. We introduced the model's central ideas in the

Java example in Section 3, but do not present the model in its entirety, in this section. However, we present it in Section A.

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).

4.2.4 Subject Definition and Selection Strategy

As illustrated in the Java example in Section 3, the subjects of our study are products as they change over time. And, although subjects should be selected intentionally (Runeson and Host, 2009), we selected ours based on availability, as do many studies (Benbasat et al, 1987) and experiments (Hannay 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 cases 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.

4.2.5 Data Collection Methods

Because we were motivated to study multiple products and thousands of product revisions, we planned on utilizing 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 and 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 practiced, as well as its initial net size, discards over time, change in the number of potential bugs, and change in average method complexity.

4.3 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 analyze the results.

For these needs, we utilized 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 analyze the product revisions of the products that we studied.

4.3.1 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 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.

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 analyze 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 (Section 3), and defined in our software change model, we assume that the degree to which a developer practices 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). It leverages the Byte-Code Engineering Library (BCEL) (http://jakarta.apache.org/bcel) to model and compare revisions of compiled classes, methods, and instructions of Java code. 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.

If all of the changed product methods had automated test methods related to them, then we categorized the change set as "all." Likewise, if none of the changed product methods had automated test methods related to them, then we categorized the

change set as "none." However, if some of the changed product methods had automated test methods related to them, then we categorized the change set as "some." These categories provided the basis for our static analysis in the present study; our future work section describes an alternative and more accurate approach for a dynamic analysis.

Zaidman et al used an alternative static approach to determine whether or not a team was practicing TDD, in addition to analyzing 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 practiced 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 organizational changes and allows us to focus on behavioral changes. That is, organizational changes such as alphabetizing methods within a class source file, commenting source lines, 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 organizational changes rather than behavioral changes. We want our measure to recognize 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.

Nonetheless, we realize 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 (Section 3), and defined in our software change model, 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—we also performed an auxiliary study (Section B) which indicated that 80–90% of discards over time occur within 150 days of the initial change and that few discards occur after that[1]. 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.

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

[1] Our auxiliary study also analyzed data from six projects, two of which were also studied in our main study.

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.

4.3.2 Analysis Procedures

Before analyzing 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 utilized 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 analyze 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 centered 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 reorganizations of the product (for example, namespace changes).
- Merging two streams of product code (for example, incorporating "sandbox" 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 practices, and since all of the extreme outliers we due to one of these causes, we excluded the extreme outliers.

To compare and analyze 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.

4.4 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 and Host, 2009). We settled on exploring and describing the degree of TWD practiced 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.

5 Study Results

As described in Section 4, we rebuilt the revisions of the software products for our subject projects, and then measured the degree of TWD and the attributes related to team speed and the attributes related to product quality, for each set of changes that developers had made to those products. Then, we correlated the measurements and analyzed the differences between the means, as well as the statistical significances of the differences. In this section, we summarize and interpret the analysis results, discuss the results per degree of TWD, and then examine the validity of the results. In the next section, we present our conclusions.

5.1 Summaries

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.

To keep this results section focused (Robson, 2002), we have not included the scatterplot, histogram, and boxplot diagrams of the measurements. For the same reason, we have not included the tabular summaries for each of the projects; they are available in Section C. Instead, we present lattice plots to summarize the 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). Finally, we present the percent differences for some of the comparisons, to highlight significant differences.

5.1.1 Initial Net Size

Figure 1 summarizes 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 2 summarizes the confidence intervals for the differences between the none-some and the none-all initial net size measurement comparisons. It also indicates the significances of the differences, with shading, according to the vertical color 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 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 3 summarizes the percentage differences between the averages for the none-some and the none-all 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.

5.1.2 Discards Over Time

Figure 4 summarizes 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 practiced. When these developers exercised some of their changes with tests, sometimes more and sometimes less of their changes were discarded over time, as

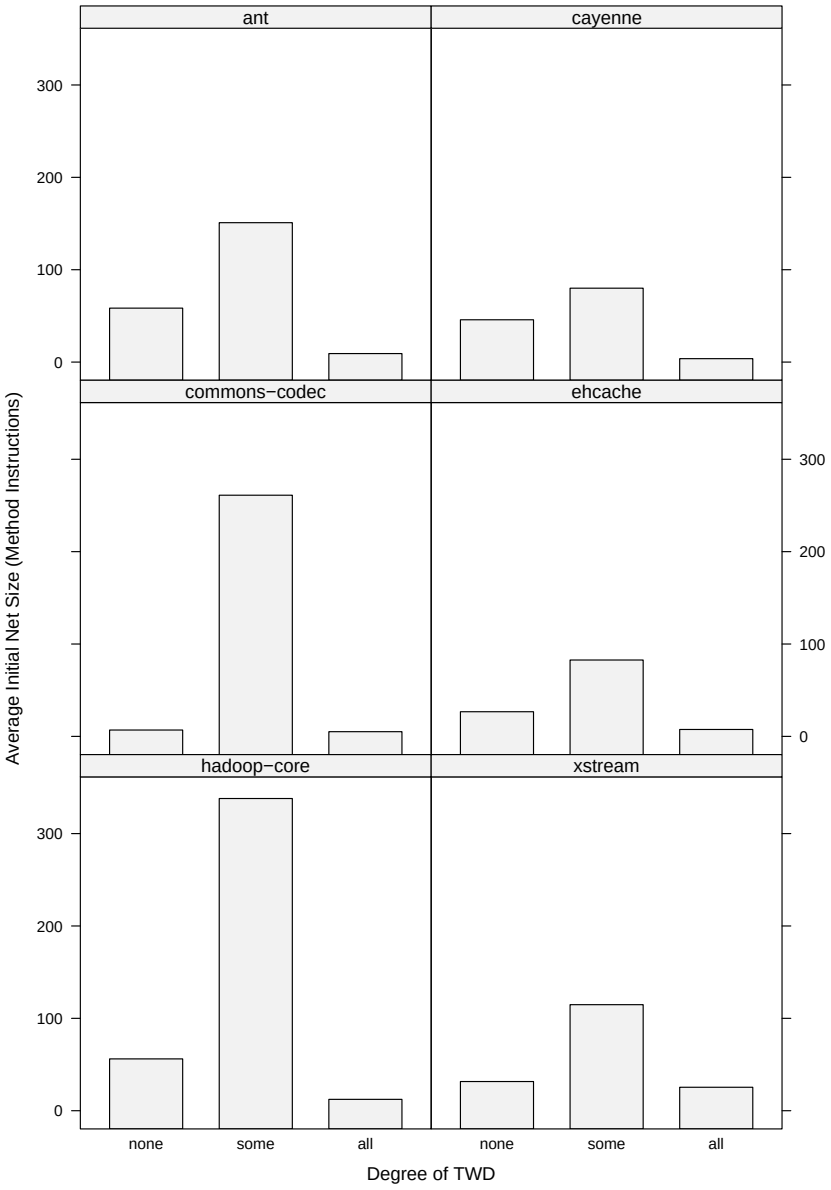

Fig. 1: Average Initial Net Size

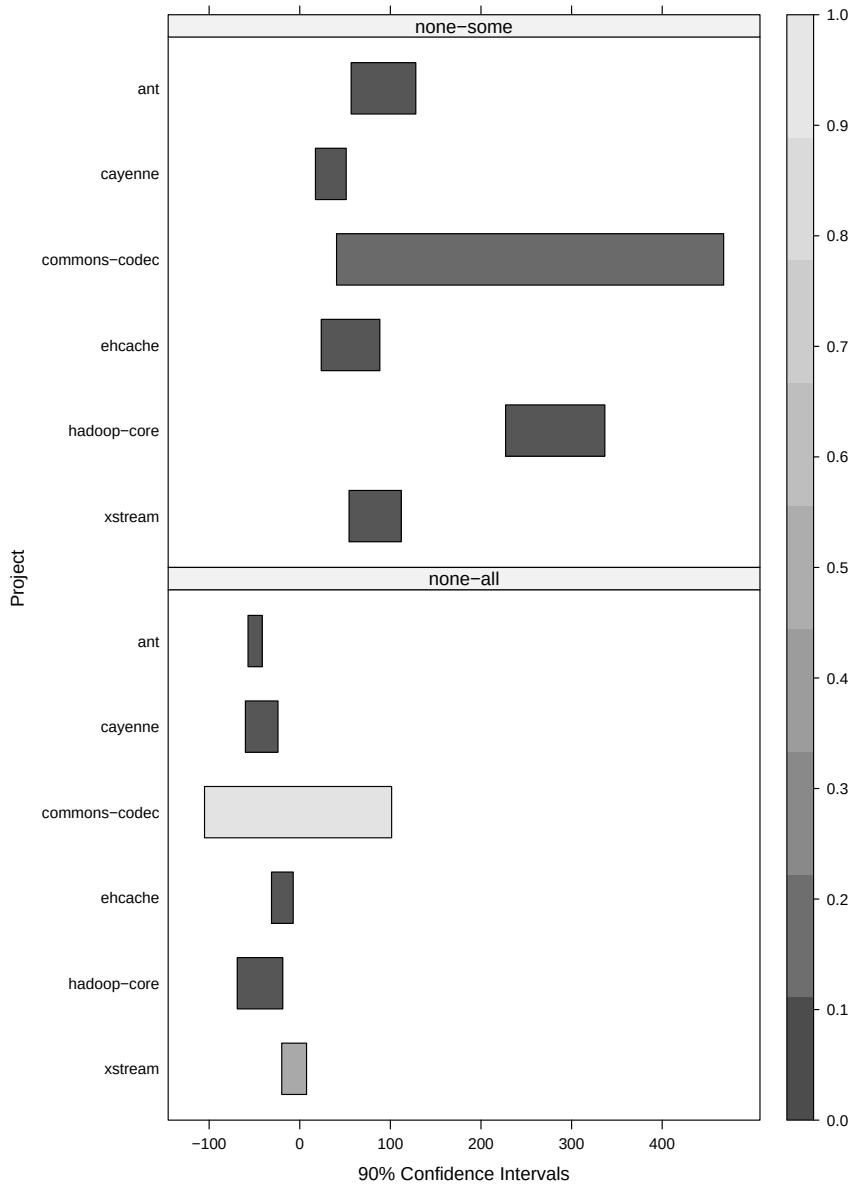

Fig. 2: Confidence Intervals for Initial Net Size Comparisons

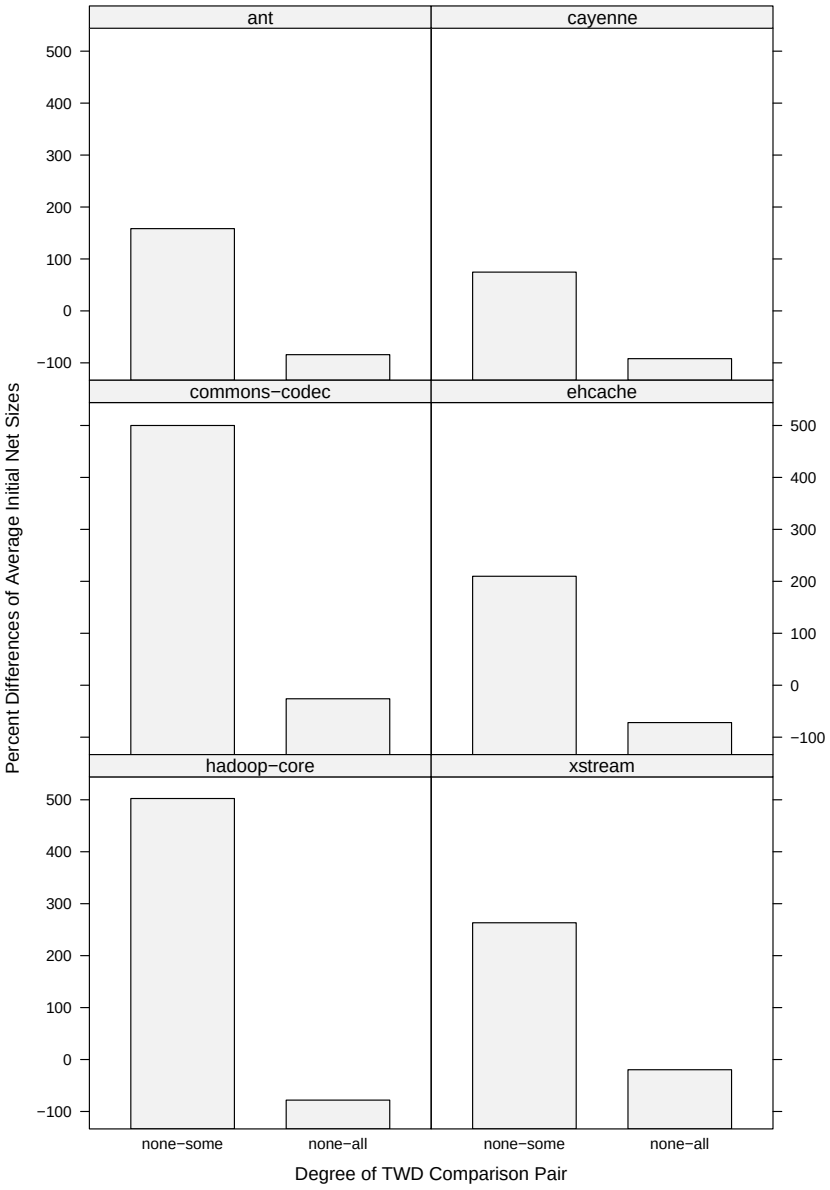

Fig. 3: Percentage Differences for Average Initial Net Size Comparisons

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 5 summarizes the confidence intervals for the differences between the none-some and the none-all discards over time measurement comparisons. It also indicates the significances of the differences, with shading, according to the vertical color key on the far right of the figure. All but one of the confidence intervals are significant, indicating that there were significant differences between the groups that were compared. However, in both the none-some and none-all comparison groups, there are cases where developers exercised their changes with tests and fewer of their changes were discarded over time (cayenne in the none-all comparison group) and cases where developers exercised their changes with tests and more of their changes were discarded over time (hadoop-core in the none-some comparison group). Additionally, these results show that the differences between the discards over time typically varied between +/-10%, for the projects that we studied.

5.1.3 Net Size Over Time

Figure 6 summarizes 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 7 summarizes the percentage differences between the derived values for the none-some and the none-all 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.

5.1.4 Potential Bugs

Figure 8 summarizes 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 practiced 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 practiced TWD, to some degree or fully, and removed more or less potential bugs than when they did not practice it.

Figure 9 summarizes the confidence intervals for the differences between the none-some and the none-all change in potential bugs measurement comparisons. It also indicates the significances of the differences, with shading, according to the vertical color 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

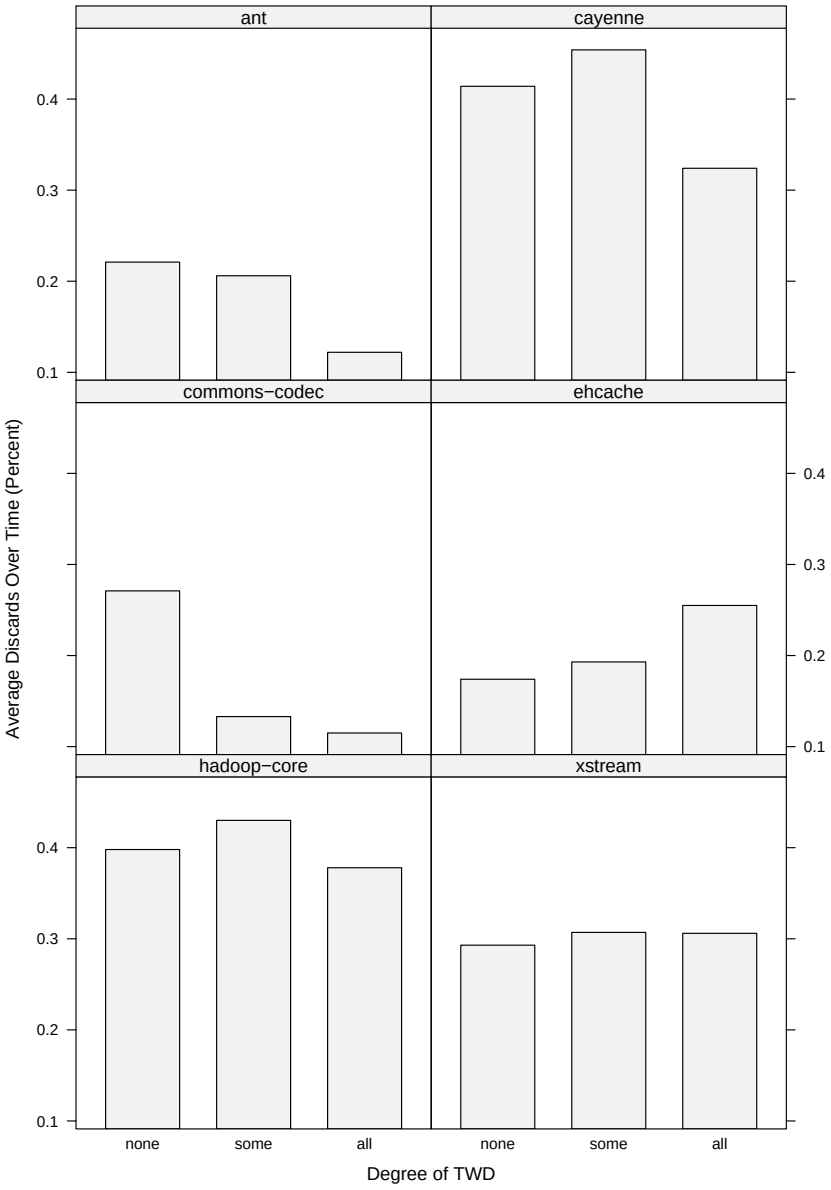

Fig. 4: Average Discards Over Time

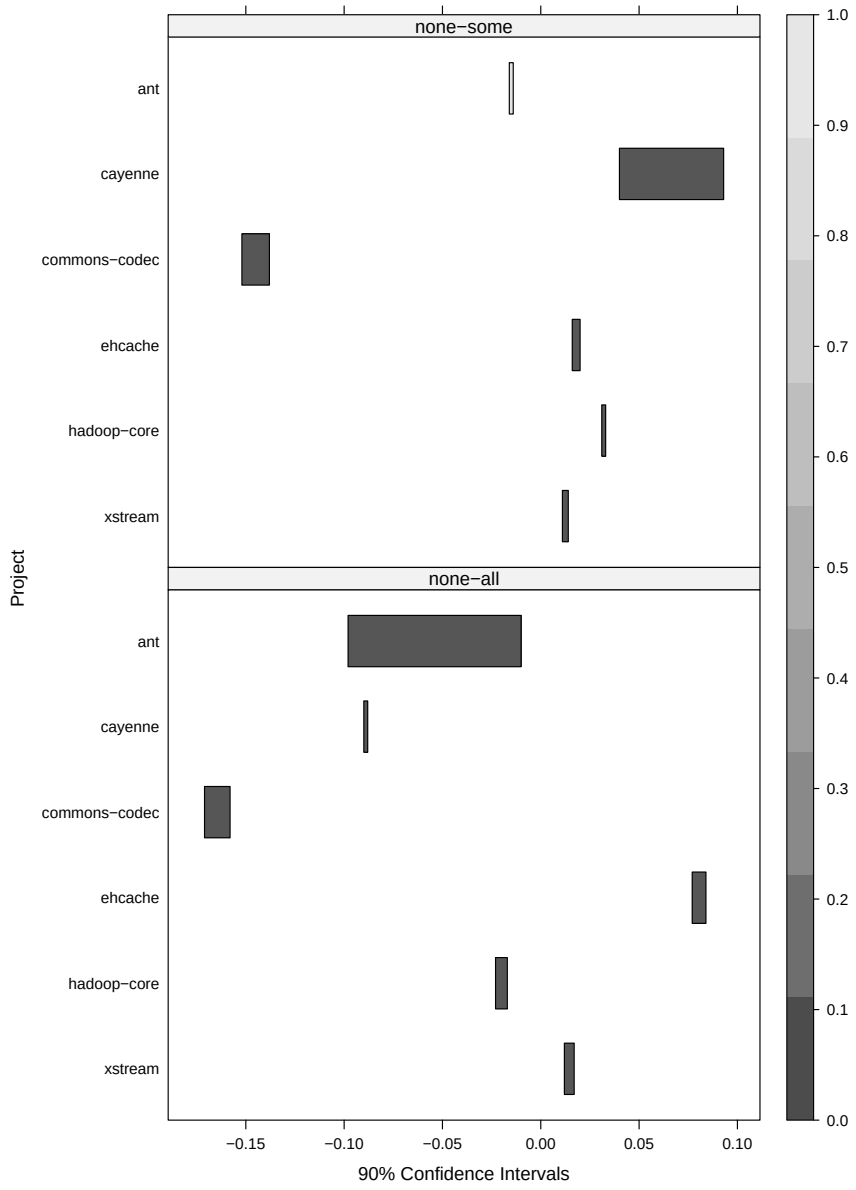

Fig. 5: Confidence Intervals for Discards Over Time Comparisons

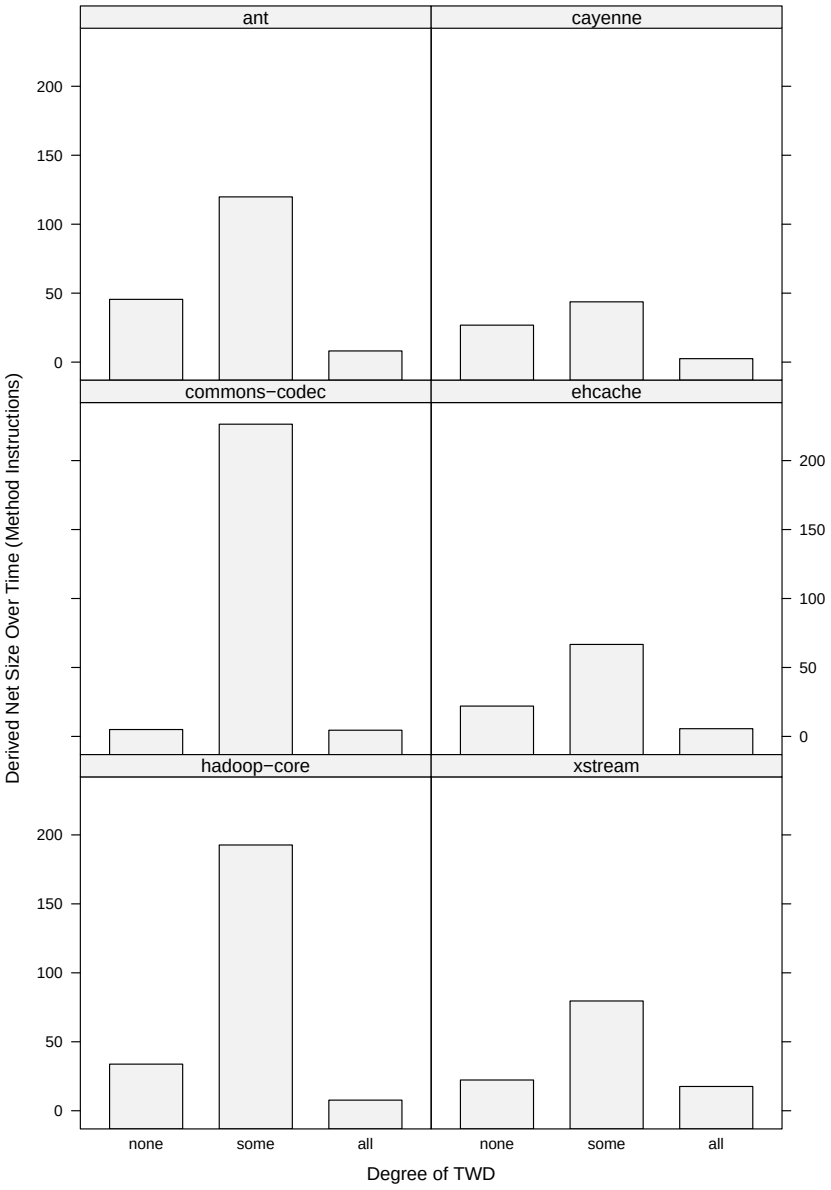

Fig. 6: Derived Net Size Over Time

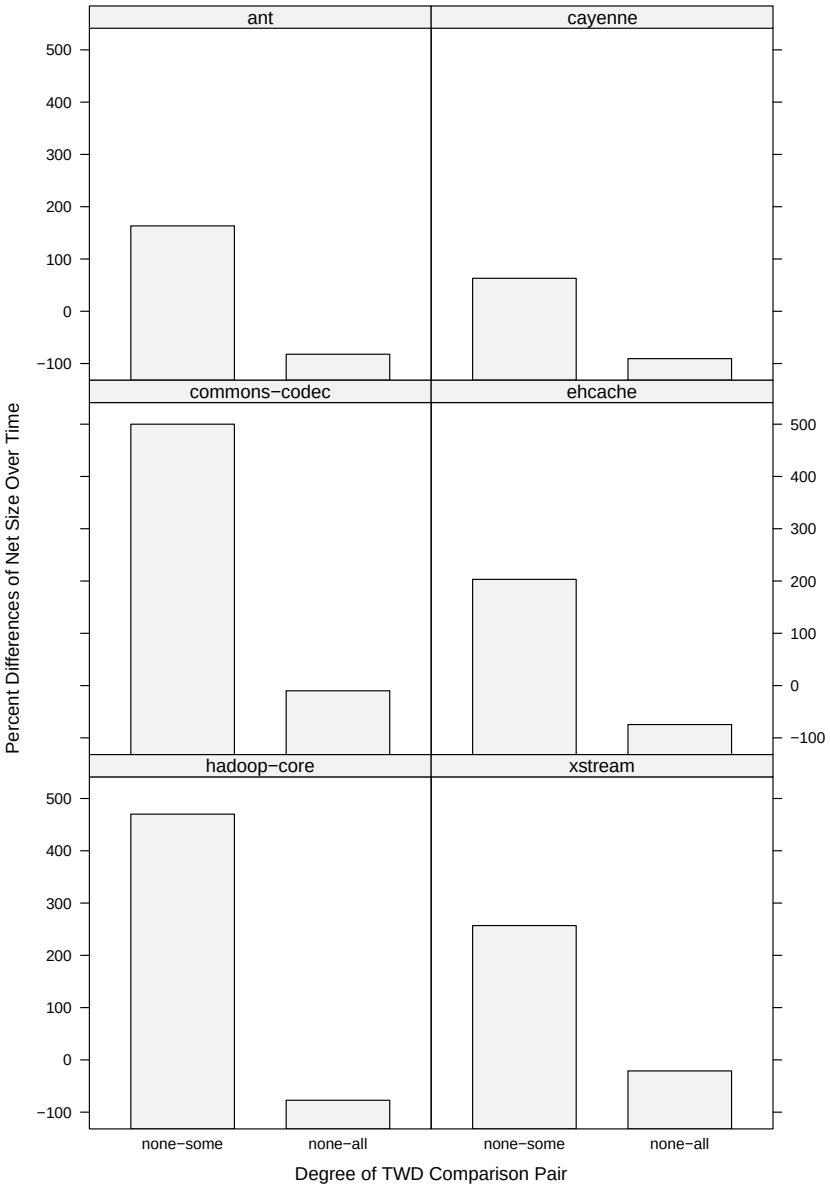

Fig. 7: Percentage Differences for Net Size Over Time Comparisons

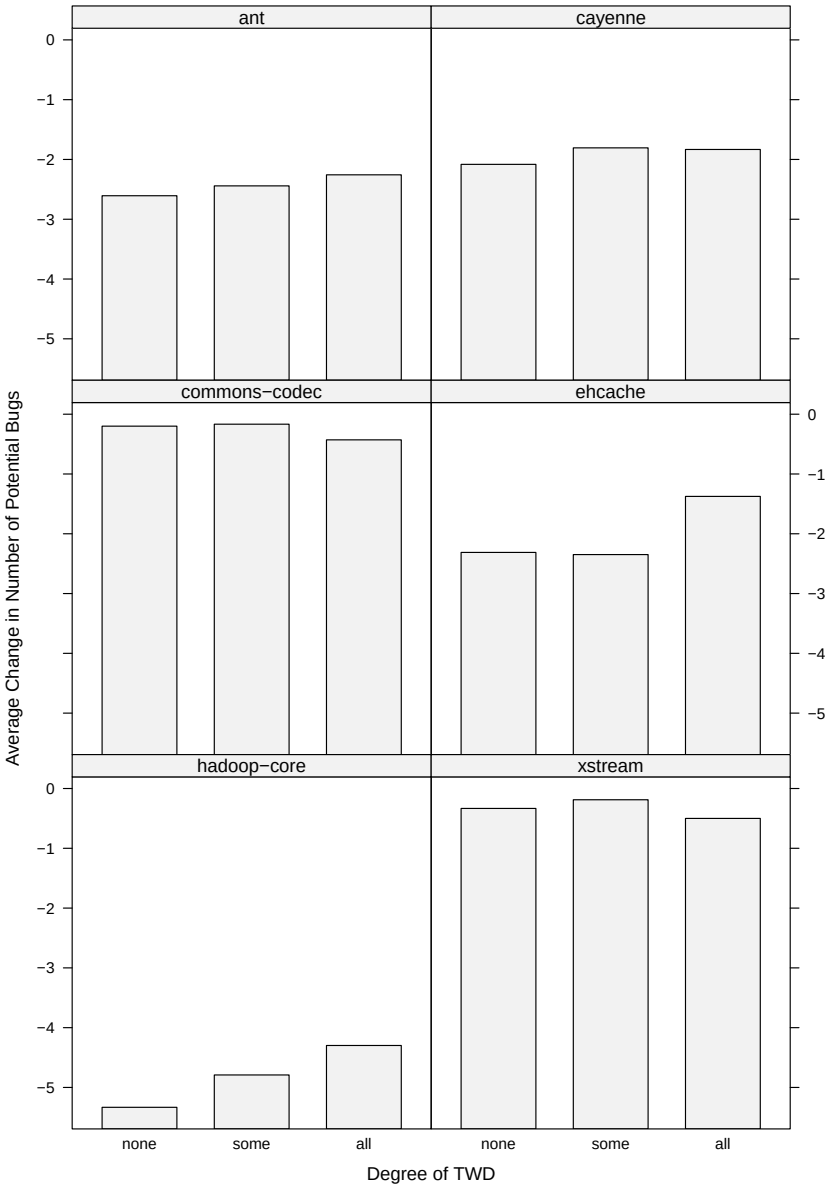

Fig. 8: Average Change in Potential Bugs

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

5.1.5 Average Method Complexity

Figure 10 summarizes 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 changes with tests.

Figure 11 summarizes the confidence intervals for the differences between the none-some and the none-all change in average method complexity measurement comparisons. It also indicates the significances of the differences, with shading, according to the vertical color key on the far right of the figure. Three significant confidence intervals and three other consistent confidence intervals in the none-some 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 none-all comparison, these intervals seem to confirm the neutrality of developers making changes when they practice TWD fully, as compared to not practicing it at all; sometimes they increase the average method complexity of the methods they change and sometimes they decrease it.

Finally, Figure 12 summarizes the percentage differences between the averages for the none-some and the none-all 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.

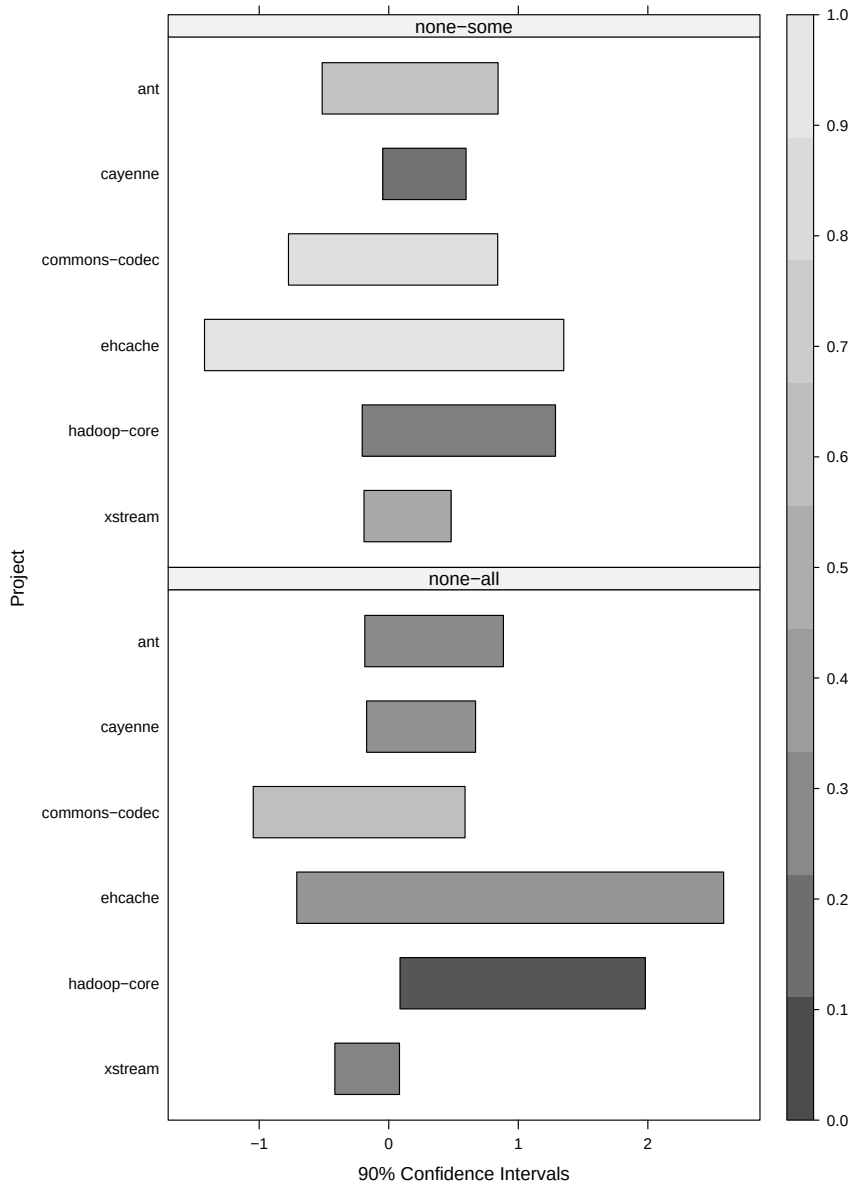

Fig. 9: Confidence Intervals for Change in Potential Bugs Comparisons

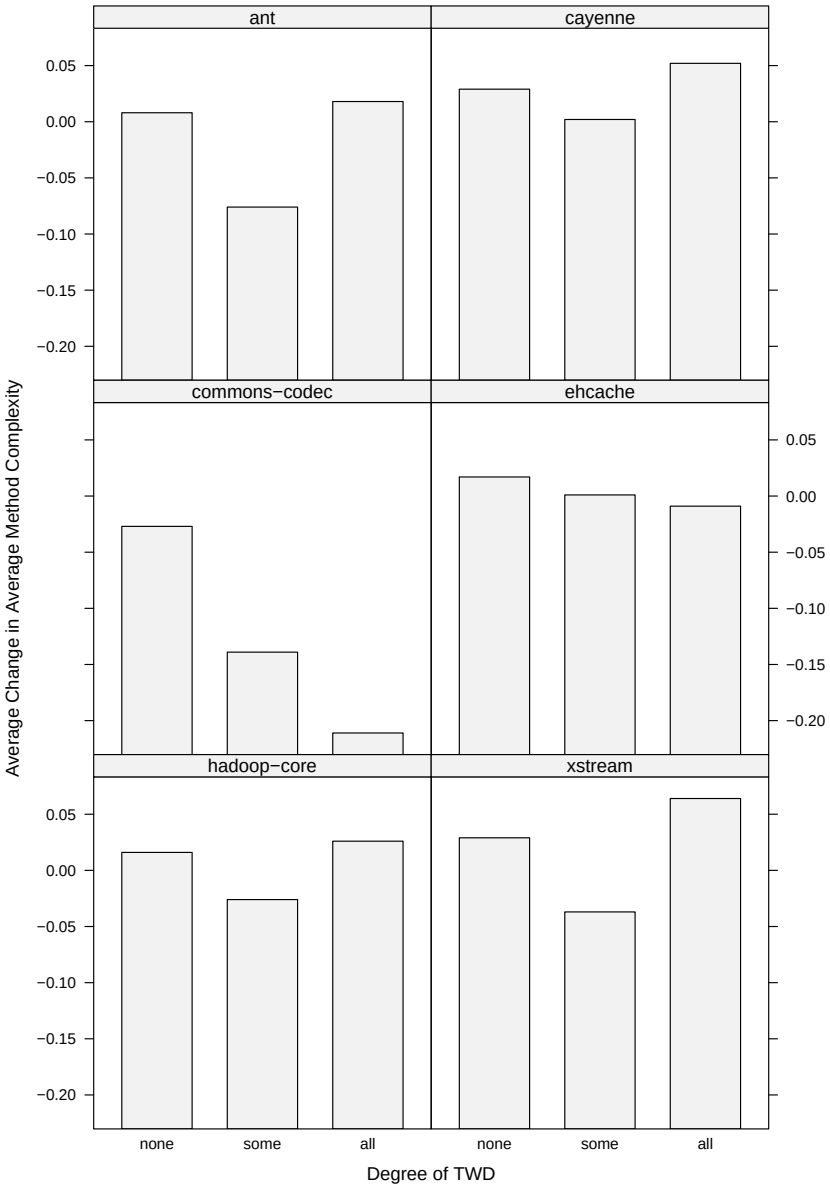

Fig. 10: Average Change in Average Method Complexity

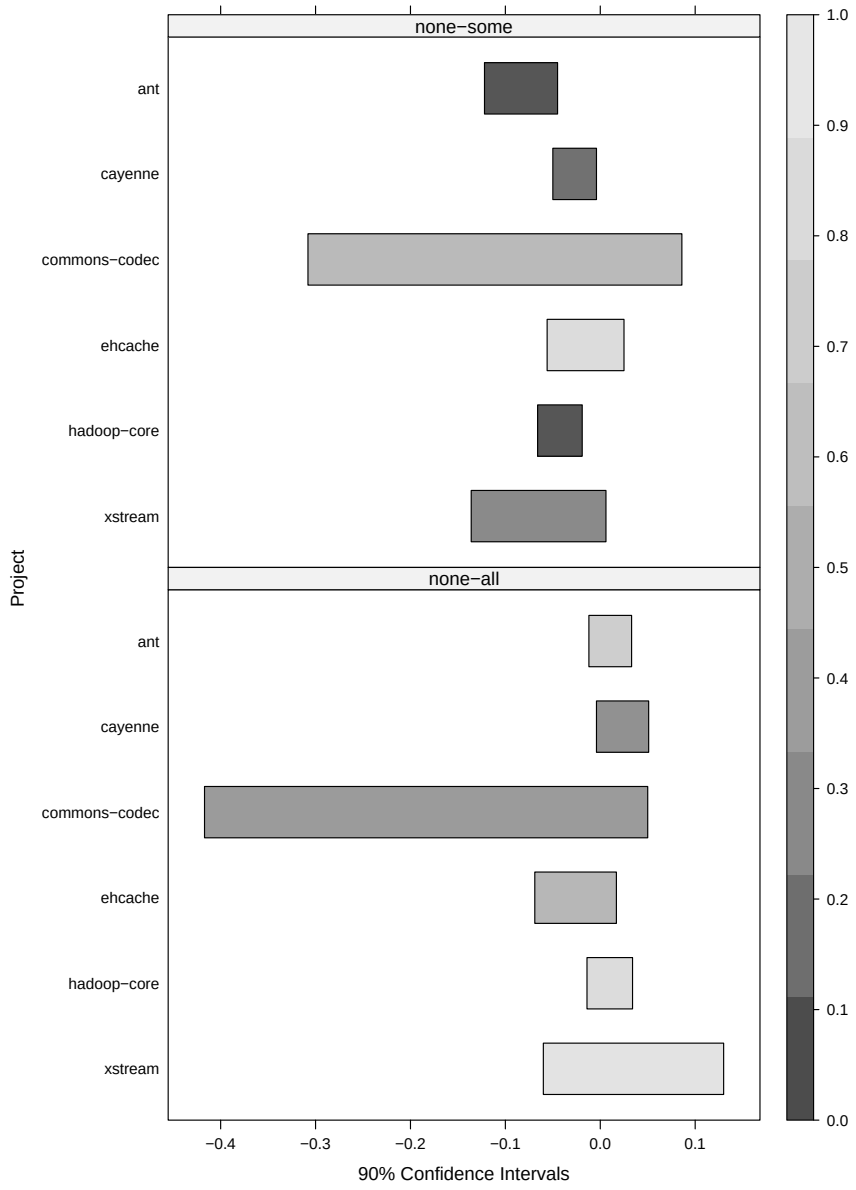

Fig. 11: Confidence Intervals for Change in Average Method Complexity Comparisons

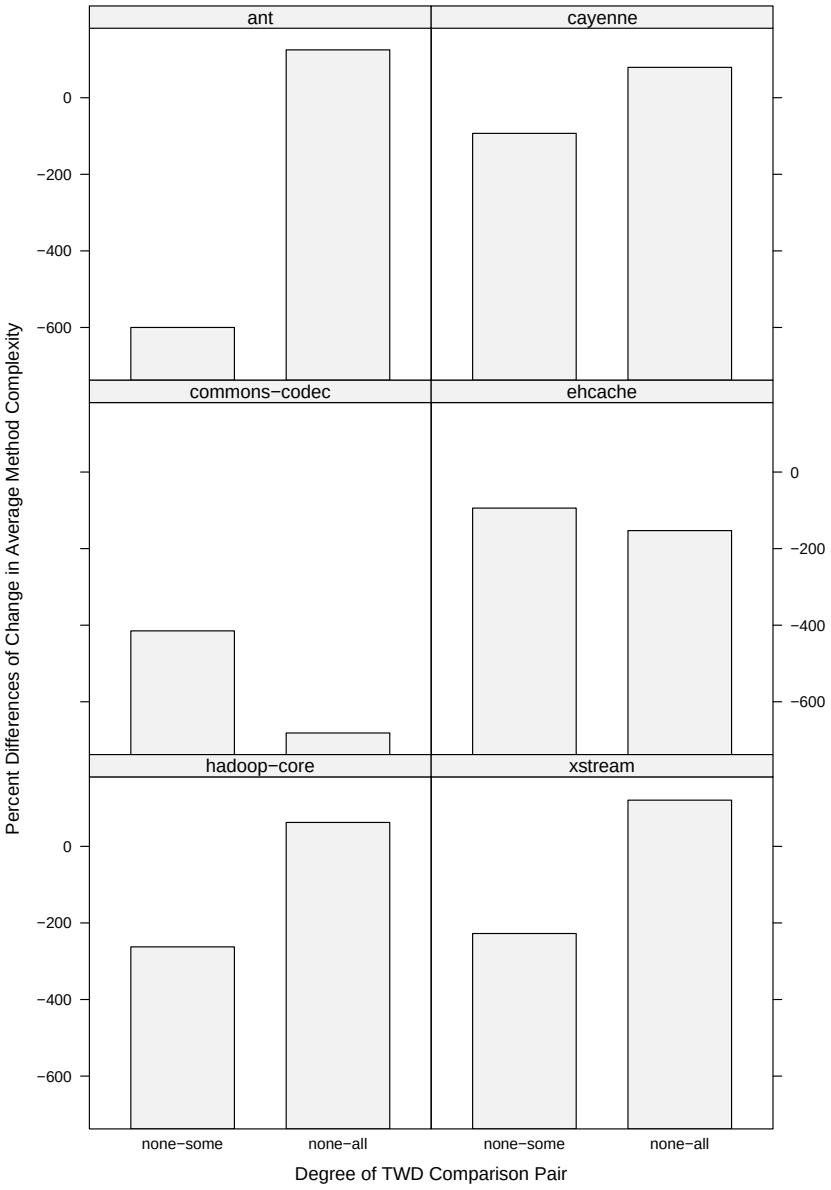

Fig. 12: Percentage Differences for Change in Average Method Complexity Comparisons

5.2 Discussion of Some TWD Results

As summarized above, when the developers practiced 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 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 practiced TWD to a degree. After analyzing 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 12 we only showed 600% so that the other comparisons would not be overwhelmed. After analyzing a sampling of the source code changes, we conclude that the team's commitment to refactoring and simplifying their design, when they practiced TWD to a degree, resulted in this extreme value.

5.3 Discussion of All TWD Results

Further, as summarized above, when the developers practiced 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 analyzing 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 practicing TWD for substantial changes; therefore, the smaller changes they made when they practiced TWD fully were more similar in size. But, for the xstream project, the developers added/changed more automated tests per change when they practiced 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 practiced TWD fully. After analyzing 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 practiced TWD fully; in these projects, the developers took the extra steps to refactor and simplify, which resulted in lower average method complexity.

5.4 Validity Discussion

5.4.1 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 practicing 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 Section 4.3, when the data from these revisions was outside of the normal distribution, it was removed.

5.4.2 Static TWD Analysis

Another threat to the validity of our data is our measurement for the degree of TWD that a developer practiced while making a product revision. 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 practiced 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 practiced 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 practice, the first problem with static analysis is nearly moot; after sampling the automated tests written by the developers of the projects we studied, nearly all of the relationships from automated test methods to product methods are direct. This is 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 have manually analyzed samples of the changes that were made, for each project, to see if we correctly classified them with respect to the degree of TWD that the developer actually practiced: none, some, or all. Again, for nearly all of these samples, our static analysis classification and our manual classification match. Nonetheless, as identified in the future work of Section 6, we think that the degree of TWD that a developer practiced 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 present study.

5.4.3 Tool Bias

A third threat to the validity of our data is tool bias, which could result from developers using additional tools 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 practicing TWD and its effects on the number of potential bugs.

To investigate this threat, we downloaded and searched the mail archives[2] 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 practicing TWD fully resulted in between 0 and 2 more potential bugs, on average. Therefore, since some developers may have been using findbugs, this result should be accepted with caution.

6 Conclusion

In this section, we present our conclusions, drawn from the study results in Section 5 and our interpretation of them. Additionally, we relate our results to the results of the previous studies on TFD, which we presented in Section 2. 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.

6.1 Conclusions

Based on our study design (Section 4) and our study results (Section 5), when a developer practiced 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 practiced 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 practiced.

These results agree in part and disagree in part with our initial hypotheses, in Section 4. The results for developers practicing TWD to some degree agree with our initial hypotheses: bigger changes over time and less average method complexity. However, the

[2] We used case-insensitive grep to search the archives

results for developers practicing 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 practicing 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, 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. 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.

Therefore, based on these interpretations, we conclude that teams can benefit from practicing TWD. In particular, we conclude that they can simplify their product more quickly when they practice TWD, incrementally adding automated tests to support its simplification.

6.2 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 Section 2, Table 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 practicing TWD to some degree match the increase in attributes related to speed from the results of the student studies (Section 2, Table 1) and the increase in attributes related to quality from the results of the professional studies (Section 2, Table 2). In contrast our results for developers practicing TWD fully match the decrease in attributes related to speed from the results of the professional studies.

6.3 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, practicing 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 practicing 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 generalizing 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.

6.4 Future Work

The most compelling question raised by this study is whether practicing 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. Therefore, we submit that this is an important area of future work.

As well, as described in Section 5, 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 practicing TWD, which may or may not require practicing TFD. Therefore, 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? Additionally, is there a profile with which a team could optimize the potential benefits of TWD by practicing 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?

6.5 Summary

We began our study with this general research question: *What are the relationships between practicing 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 practicing it are real-life software development teams, developing their product over the course of many years*?

In preparation for answering this question, we developed a precise software change model (Section A) so that we could determine the degree of TWD, the initial net size, and the discards over time for a particular developer's change. Then, we calibrated that model by performing an auxiliary study (Section B) to determine how far into the future we needed to "look" to estimate the effect on one attribute related to a team's speed: discards over time—150 days was the answer. After that, we planned and designed our main study to explore and describe the relationships between practicing TWD when making a change and the effects of the change on the two attributes related to team speed and two attributes related to product quality (Section 4). Finally, we executed our main study, assembled its results, and analyzed its validity (Section 5).

Our main study results indicate that there were significant differences between developers practicing TWD when making a change, and the resulting effects on the attributes that we studied. When the developers exercised some of their changes with tests, they made significantly larger changes over time while significantly reducing their product's average method complexity. And, when they exercised all of their changes with tests, they made significantly smaller changes over time.

We interpret these results to indicate that developers refactored their product more when they developed their tests with their product changes, reducing the complexity of their product faster. Therefore, we conclude that teams that wish to reduce the complexity of their product over time, and to do so more quickly, can benefit from practicing TWD.

A Software Change Model

Because of the mixed results from previous studies and some of the study limitations, we were motivated to study the effects of TWD. However, before we could design and perform a study, we needed a model to support some of the measurements for it, an approach that has since been recommended by Hannay et al (2007).

Ideally, we wanted a model to support direct measurements on versions of the 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 normalizing 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've 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'd have to convert the index card and issue measurements to a normalized measure for the number of features. Assuming that we had access to the information, this normalization task would require some effort for each speed variant that we wanted to study. Further, we would have to relate that normalized

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.

Therefore, 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 versions of products. We defined a model which allows us to make direct measurements for the degree of TWD that a developer practiced 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 direct measurements for two attributes related to product quality: average method complexity and the number of potential bugs. By doing so, we removed the normalization and association overheads illustrated by our example.

A.1 Specific Example

In the specific example that we presented in Section 3, we introduced our software change model. Products are composed of methods, which may be exercised by product tests. And, for each change, a developer practices 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 average method complexity and the number of potential bugs within the product.

In the following subsections, we define these concepts (except for the product quality concepts) formally by presenting them using set notation; since we adopt the product quality models and tools, we simply refer the reader to them. And, although we considered using the Unified Modeling Language (UML) (OMG, 2007) to communicate our model, we feel that set notation is both more precise and concise for our particular needs.

A.2 Product Changes

A.2.1 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.

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

A.2.2 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.

Therefore, in a build, we have a relation between the product methods and the test instructions. 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.

$$testInstructions : MP \leftrightarrow IT$$
$$MPE = \{\, x \in MP \mid testInstructions(x) \neq \emptyset \,\}$$
$$MPU = \{\, x \in MP \mid testInstructions(x) = \emptyset \,\}$$

A.2.3 Product Method Changes

To describe the degree to which a team practiced TWD for a particular change set, we first need to know which product methods changed. We can do so by recognizing 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 is a member of MPR.

$$MPA_{ij} = \{\, x \in MP_j \mid (\nexists\, y \in MP_i \mid sig(x) = sig(y)) \,\}$$
$$MPD_{ij} = \{\, x \in MP_i \mid (\nexists\, y \in MP_j \mid sig(x) = sig(y)) \,\}$$
$$MPR_{ij} = \{\, (x, y) : x \in MP_i, y \in MP_j \mid sig(x) = sig(y) \,\}$$

Within the retained product methods, MPR, each method 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.

$$MPM^{old}_{ij} = \{\, x : (x, y) \in MPR_{ij} \mid impl(x) \neq impl(y) \,\}$$
$$MPM^{new}_{ij} = \{\, y : (x, y) \in MPR_{ij} \mid impl(x) \neq impl(y) \,\}$$
$$MPI_{ij} = \{\, y : (x, y) \in MPR_{ij} \mid impl(x) = impl(y) \,\}$$

A.3 Degree of Test-With Development

We can now define the degree of TWD that a developer practiced when making a change. We define it as the number of product methods that have been added or modified and are "exercised" by tests, divided by the number of product methods that have been added or modified (Equation 1).

$$DegreeOfTWD_{ij} = \frac{\#((MPA_{ij} \cup MPM^{new}_{ij}) \cap MPE_{ij})}{\#(MPA_{ij} \cup MPM^{new}_{ij})} \tag{1}$$

A.4 Team Speed

A.4.1 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 both, IPM_i and IPM_j.

$methodInstructions : MP \rightarrow IP$
$$IPA_{ij} = \{\, x \in MPA_{ij} \mid methodInstructions(x) \,\}$$
$$IPD_{ij} = \{\, x \in MPD_{ij} \mid methodInstructions(x) \,\}$$
$$IPM_i = \{\, x \in MPM^{old}_{ij} \mid methodInstructions(x) \,\}$$
$$IPM_j = \{\, x \in MPM^{new}_{ij} \mid methodInstructions(x) \,\}$$

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.

Therefore, 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.

$$InitialNetSize_{ij} = \#IPA_{ij} - \#IPD_{ij} + (\#IPM_j - \#IPM_i) \tag{2}$$

A.4.2 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).

$$\Delta D_{ij} = \Delta_{ij} \setminus \Delta P_{ij} \tag{3}$$

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 4).

$$\Delta P_{ij} = \Delta_{iN} \setminus \Delta_{jN} \tag{4}$$

Hence, by substituting for ΔP_{ij}, from Equation 4 and into Equation 3, we arrive at a revised definition for discards for build j (Equation 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.

$$\Delta D_{ij} = \Delta_{ij} \setminus (\Delta_{iN} \setminus \Delta_{jN}) \tag{5}$$

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 allows us to define the discards over time for the changes made to build i in build j (Equation 6).

$$DiscardsOverTime_{ij} = \frac{\Delta D_{ij}}{\Delta t_{jN}} \tag{6}$$

A.4.3 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 2). And, some of those changes will be discarded over time (Equation 6). So, we define the net size over time as the adjustment of the initial net size by the discards over time (Equation 7).

$$NetSizeOverTime_{ij} = InitialNetSize_{ij} - DiscardsOverTime_{ij} \tag{7}$$

A.5 Product Quality

A.5.1 Average Method Complexity

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.

A.5.2 Number of Potential Bugs

Similarly, 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.

A.6 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 practiced 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 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 practiced, when making a product change, is equal to the number of changed and exercised methods divided by the number of changed methods (Equation 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 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 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 7). Finally, it adopts two pre-existing models and tools to define and measure the average method complexity and the number of potential bugs for a change.

B Auxiliary Study

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 (Section A). 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.

B.1 Study Design

B.1.1 Motivation

Within our software change model (Section A), 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.

B.1.2 Plan

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. And, we wished to describe a common discard pattern for all of the projects, if one existed. Our hope was that we could identify a common discard pattern which we could then use to make reasonable estimates of discards over time for other projects.

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 reasonable estimates regarding the discards over time for that product change*?

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

Although study subjects should be selected intentionally (Runeson and Host, 2009), we selected ours based on availability, as do many studies (Benbasat et al, 1987) and experiments (Hannay 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.

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

B.1.3 Protocol: Preparation

First, we selected and reconstructed the revisions for six projects. 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 the following projects, by identifying the revisions that developers had made, and then assembling, building, and storing each revision that could be built (without compilation errors).

- 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.

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, 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, if all permutations are considered. Therefore, 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, utilizing 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.

B.1.4 Protocol: Comparison

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 5). Then, we directed the custom toolset to make the class comparison(s) with a command (Listing 6); 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 something like Listing 7 and their summary looks something like Listing 8

```
<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 5: Comparison Specification

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

Listing 6: 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 7: 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 8: Comparison Summary

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 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 9 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 9: 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 10 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 10: 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 11 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 11: Comparison Summary Δ_{iN}

B.1.5 Protocol: Calculation

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 12 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 12: Persisted Summary ΔP_{ij}

Second, we divided the relevant portions of the persisted changes results by the original results corresponding to Δ_{ij} (Listing 9), in order 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 13 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 13: Persisted Percentages

Third, we subtracted the percentages of the persisted changes from unity, in order to calculate the discarded changes. Listing 14 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 14: 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.

B.2 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 13 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 visualize the patterns in a curve group. Finally, these curves present the preserved compiled instructions for each of the projects.

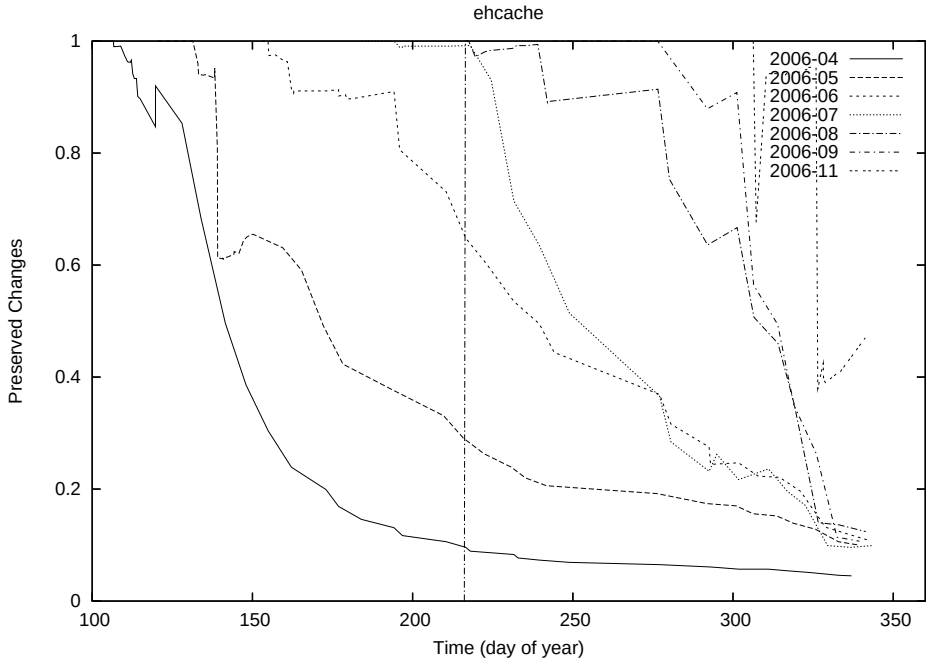

Fig. 13: ehcache—Preserved Changes over Time (Absolute)

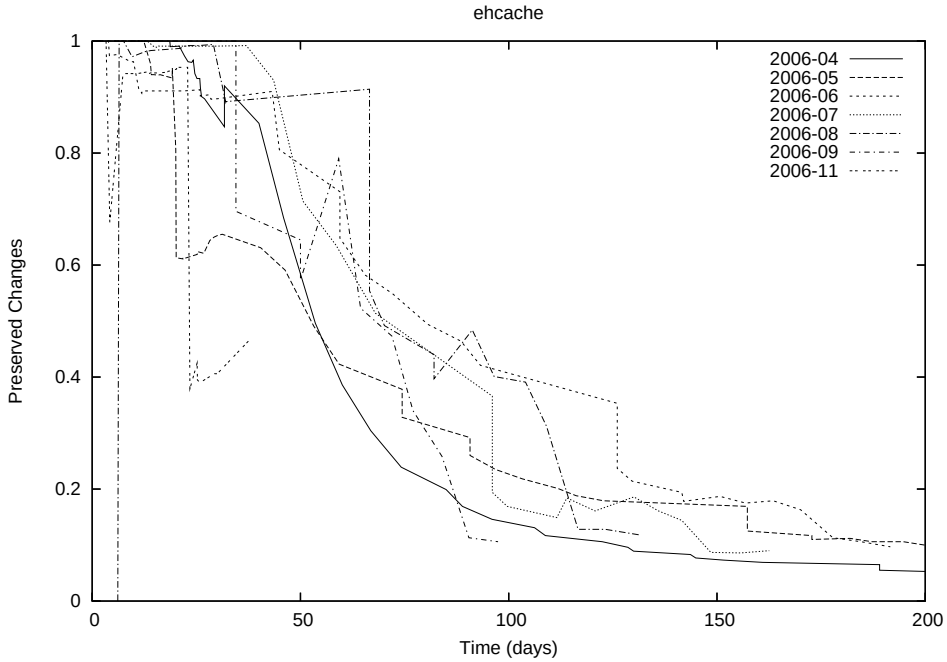

Fig. 14: ehcache—Preserved Changes over Time (Relative)

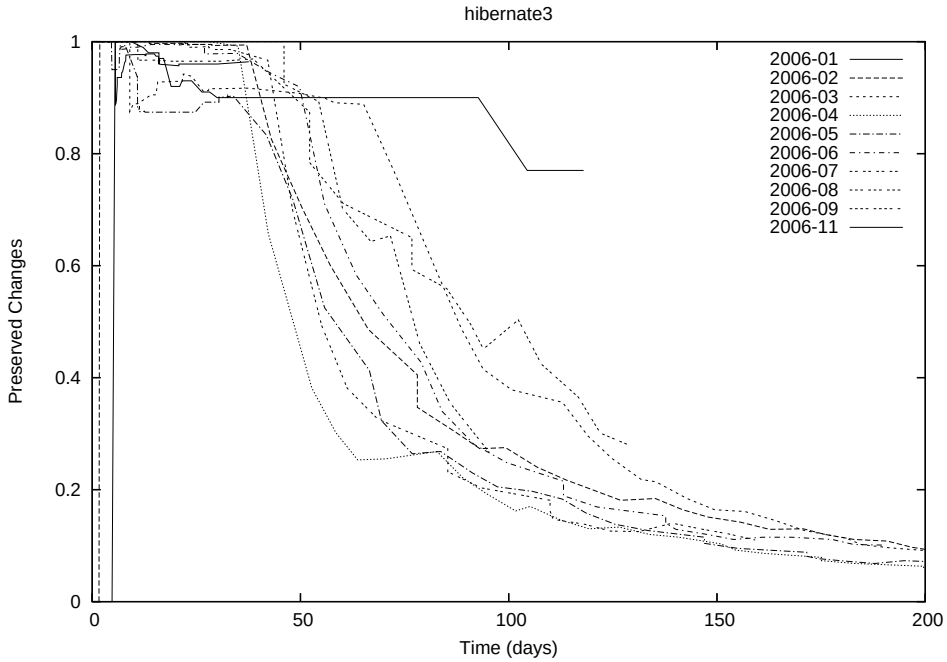

Fig. 15: hibernate3—Preserved Changes over Time (Relative)

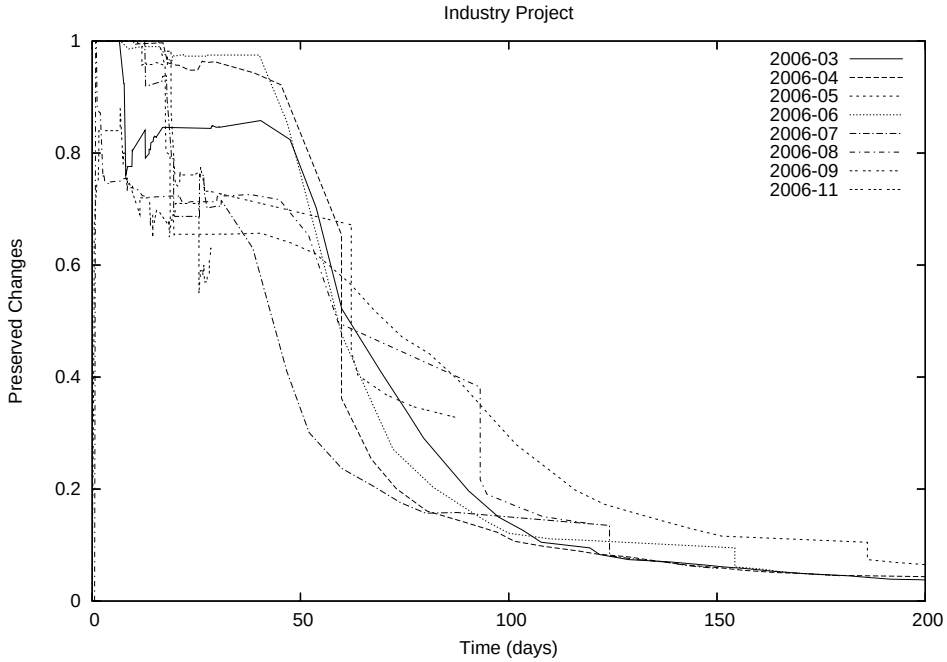

Fig. 16: industry—Preserved Changes over Time (Relative)

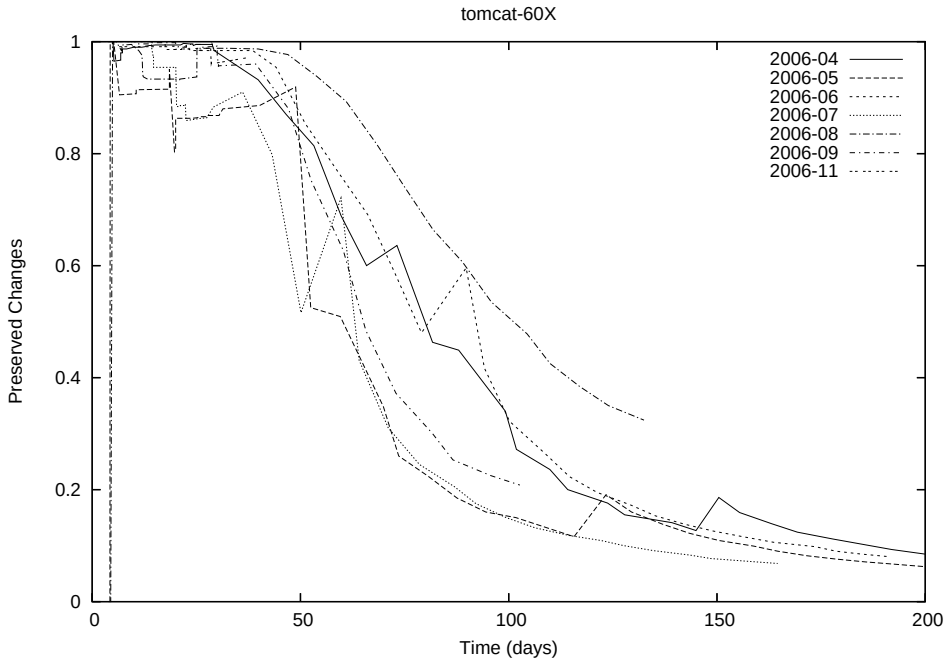

Fig. 17: tomcat-60X—Preserved Changes over Time (Relative)

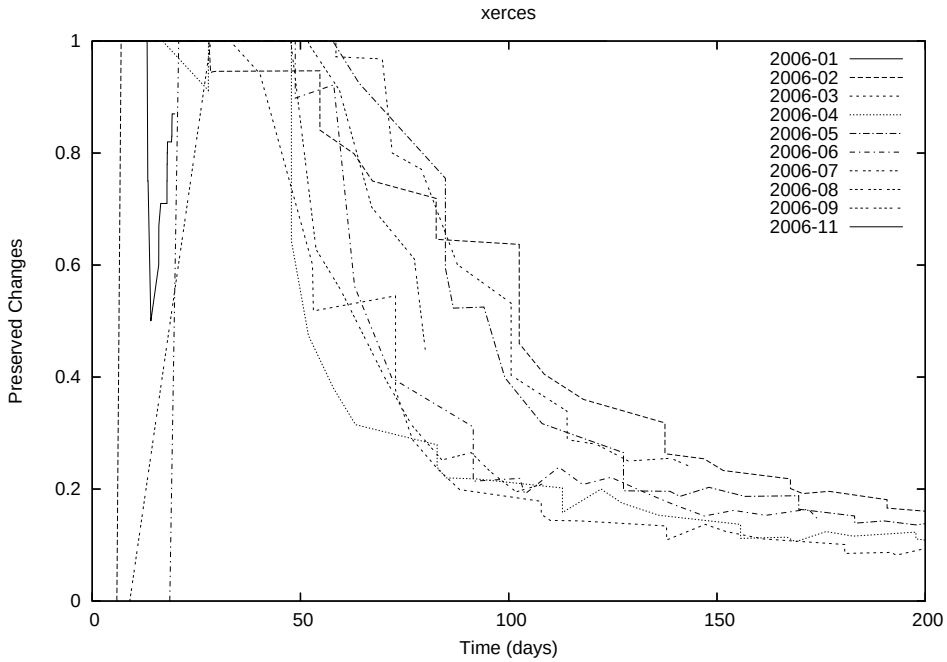

Fig. 18: xerces—Preserved Changes over Time (Relative)

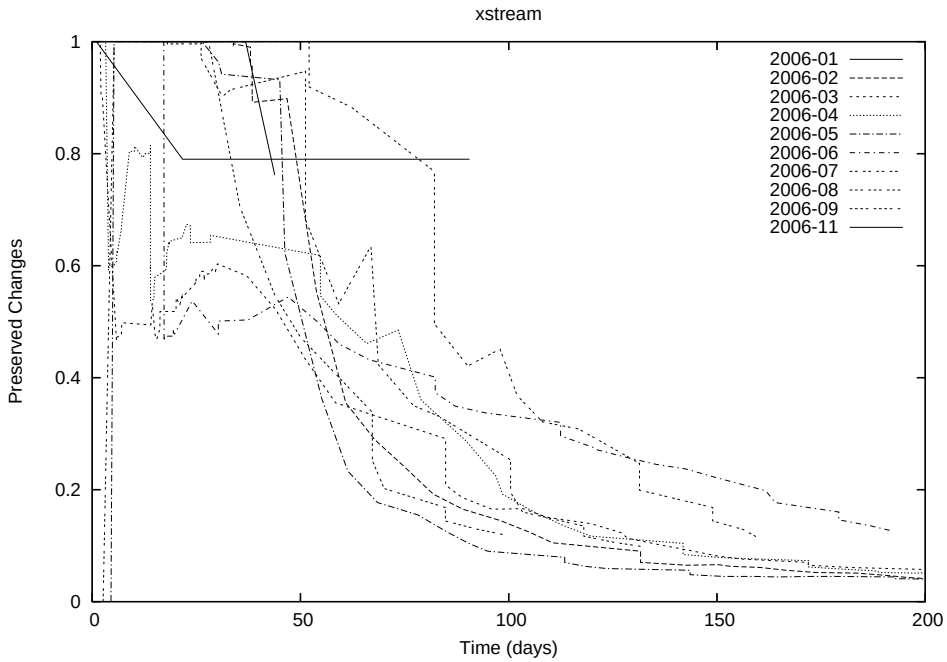

Fig. 19: xstream—Preserved Changes over Time (Relative)

B.2.1 Analysis

Based on our preserved/discarded changes over time measurements, we observed three patterns across certain periods of time: stabilization, decay, and constant. During the stabilization 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.

Our first observation was that there appeared to be a "stabilization 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 characterized by "erratic" or random fluctuations.

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 characterized 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.

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.

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 stabilization period or the decay period.

B.2.2 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.

B.3 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 Section A, 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: stabilization, 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 stabilization 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.

C Individual Project Results

C.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 practiced 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 analyzed: (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 5 presents a summary of the measurement means for each group and overall, in addition to the overall standard deviation. Table 6 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: average method complexity	0.008	-0.076	0.018	-0.003	0.325

Table 5: 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: average method complexity	none,some	0.045	0.122	< 0.01
quality: average method complexity	none,all	-0.033	0.012	0.439

Table 6: ant—Summary of Welch Two Sample t-test

Consider the results presented in Table 5. These represent the mean values of the dependent variables, categorized by the degree to which the developer of each change practiced 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, we can see that in the first four rows of Table 6 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 6, 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 6, 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); note that 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, we can see that in the last four rows of Table 6 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 practiced 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 5). And, on average, they reduced the average method complexity by 1050% ($(-0.076 - 0.008)/0.008 = 1050\%$, also from Table 5).

But, when they practiced 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 5).

C.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 practiced 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 analyzed: (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 7 presents a summary of the measurement means for each group and overall, in addition to the overall standard deviation. Table 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	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: average method complexity	0.029	0.002	0.052	0.020	0.259

Table 7: 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: average method complexity	none,some	0.004	0.050	0.058
quality: average method complexity	none,all	-0.051	0.004	0.162

Table 8: cayenne—Summary of Welch Two Sample t-test

In summary, assuming we interpret the results as we did in Section C.1, when the cayenne developers practiced 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 practiced 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 ($(2.5 - 26.8)/26.8 = 91\%$).

C.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 behavior 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 practiced 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 analyzed: (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 9 presents a summary of the measurement means for each group and overall, in addition to the overall standard deviation. Table 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	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: average method complexity	-0.027	-0.139	-0.211	-0.147	0.367

Table 9: 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: average method complexity	none,some	-0.086	0.308	0.341
quality: average method complexity	none,all	-0.050	0.417	0.192

Table 10: commons-codec—Summary of Welch Two Sample t-test

In summary, assuming we interpret the results as we did in Section C.1, when the commons-codec developers practiced 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, 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 practiced 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.

C.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 practiced 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 analyzed: (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 11 presents a summary of the measurement means for each group and overall, in addition to the overall standard deviation. Table 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	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: average method complexity	0.017	0.001	-0.009	0.004	0.139

Table 11: 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: average method complexity	none,some	-0.025	0.056	0.519
quality: average method complexity	none,all	-0.017	0.069	0.318

Table 12: ehcache—Summary of Welch Two Sample t-test

In summary, assuming we interpret the results as we did in Section C.1, when the ehcache developers practiced 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 practiced 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 ($(5.6 - 22.0)/22.0 = 75\%$).

C.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 practiced 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 analyzed: (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 13 presents a summary of the measurement means for each group and overall, in addition to the overall standard deviation. Table 14 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: average method complexity	0.016	-0.026	0.026	0.001	0.235

Table 13: 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: average method complexity	none,some	0.019	0.066	< 0.01
quality: average method complexity	none,all	-0.034	0.014	0.510

Table 14: hadoop-core—Summary of Welch Two Sample t-test

In summary, assuming we interpret the results as we did in Section C.1, when the hadoop-core developers practiced 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 practiced 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 ($(-4.298 - -5.332)/-5.332 = 19\%$).

C.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 practiced 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 analyzed: (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 15 presents a summary of the measurement means for each group and overall, in addition to the overall standard deviation. Table 16 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: average method complexity	0.029	-0.037	0.064	0.010	0.392

Table 15: xstream—Summary of descriptive statistics

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: average method complexity	none,some	-0.006	0.136	0.134
quality: average method complexity	none,all	-0.130	0.060	0.545

Table 16: xstream—Summary of Welch Two Sample t-test

In summary, assuming we interpret the results as we did in Section C.1, when the xstream developers practiced 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 (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 practiced 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.

References

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

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

Ayewah N, Hovemeyer D, Morgenthaler JD, Penix J, Pugh W (2008) Using static analysis to find bugs. IEEE Software 25(5):22–29, DOI http://dx.doi.org/10.1109/MS.2008.130

Basili VR, Selby RW, Hutchens DH (1986) Experimentation in software engineering. IEEE Transactions on Software Engineering 12(7):733–743

Basili VR, Shull F, Lanubile F (1999) Building knowledge through families of experiments. IEEE Transactions on Software Engineering 25(4):456–473, DOI http://dx.doi.org/10.1109/32.799939

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

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

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

Benbasat I, Goldstein DK, Mead M (1987) The case research strategy in studies of information systems. JSTOR MIS Quarterly 11(3):369–386, DOI http://dx.doi.org/10.2307/248684

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, DOI http://doi.acm.org/10.1145/1159733.1159787

Boehm BW, Abts C, Brown AW, Chulani S, Clark BK, Horowitz E, Madachy R, Reifer DJ, Steece B (2000) Software Cost Estimation with Cocomo II. Prentice Hall, Englewood Cliffs

Canfora G, Cimitile A, Garcia F, Piattini M, Visaggio CA (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, DOI http://doi.acm.org/10.1145/1159733.1159788

Card DN, McGarry FE, Page GT (1987) Evaluating software engineering technologies. IEEE Transactions on Software Engineering 13(7):845–851, DOI http://dx.doi.org/10.1109/TSE.1987.233495

Cook JE, Votta LG, Wolf AL (1998) Cost-effective analysis of in-place software processes. IEEE Transactions on Software Engineering 24(8):650–663, DOI http://dx.doi.org/10.1109/32.707700

Damm LO, Lundberg L (2006) Results from introducing component-level test automation and test-driven development. J Syst Softw 79(7):1001–1014, DOI http://dx.doi.org/10.1016/j.jss.2005.10.015

Diehl S, Hassan AE, Holt RC (2005) Report on msr 2005: international workshop on mining software repositories. SIGSOFT Softw Eng Notes 30(5):1–3, DOI http://doi.acm.org/10.1145/1095430.1095433

Diehl S, Gall H, Pinzger M, Hassan AE (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, DOI http://doi.acm.org/10.1145/1134285.1134483

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, DOI http://dx.doi.org/10.1109/TSE.2005.37

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

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, DOI http://dx.doi.org/10.1109/ICSECOMPANION.2007.8

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, DOI http://doi.acm.org/10.1145/952532.952753

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, DOI http://dx.doi.org/10.1109/METRICS.2004.2

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, DOI http://dx.doi.org/10.1109/ESEM.2007.20

Hannay JE, Hansen O, By Kampenes V, Karahasanovic A, Liborg NK, C Rekdal A (2005) A survey of controlled experiments in software engineering. IEEE Transactions on Software Engineering 31(9):733–753, DOI http://dx.doi.org/10.1109/TSE.2005.97

Hannay JE, K DI, Dyba T (2007) A systematic review of theory use in software engineering experiments. IEEE Transactions on Software Engineering 33(2):87–107, DOI http://dx.doi.org/10.1109/TSE.2007.12

Hassan AE, Holt RC, Mockus A (2005) Report on msr 2004: International workshop on mining software repositories. SIGSOFT Softw Eng Notes 30(1):4, DOI http://doi.acm.org/10.1145/1039174.1039188

Janzen DS, 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, DOI http://dx.doi.org/10.1109/CSEET.2006.25

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

Kitchenham B, Pickard L, Pfleeger SL (1995) Case studies for method and tool evaluation. IEEE Software 12(4):52–62, DOI http://dx.doi.org/10.1109/52.391832

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

Lanza M, Godfrey MW, 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, DOI http://doi.acm.org/10.1145/1370175.1370235

Lethbridge TC, Sim SE, Singer J (2005) Studying software engineers: Data collection techniques for software field studies. Empirical Software Engineering 10(3):311–341, DOI http://dx.doi.org/10.1007/s10664-005-1290-x

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

Maximilien EM, 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 TJ (1976) A complexity measure. IEEE Transactions on Software Engineering 2(4):308–320

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

Nagappan N, Maximilien EM, 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, DOI http://dx.doi.org/10.1007/s10664-008-9062-z

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

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, IEEE Press, vol 2, pp 83–86

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

Raymond ES (1999) The magic cauldron. In: The Cathedral & the Bazaar, O'Reilly & Associates, pp 137–194

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

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

Sanchez JC, Williams L, Maximilien EM (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, DOI http://dx.doi.org/10.1109/AGILE.2007.43
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, DOI http://dx.doi.org/10.1109/ESEM.2007.2
Williams L, Maximilien EM, 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
Zaidman A, Rompaey BV, Demeyer S, Deursen Av (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, DOI http://dx.doi.org/10.1109/ICST.2008.47