Noname manuscript No.
(will be inserted by the editor)

Analyzing The Effects of Test Driven Development In GitHub

Neil C. Borle · Meysam Feghhi · Eleni Stroulia · Russell Greiner · Abram Hindle

Received: date / Accepted: date

Abstract Testing is an integral part of the software development lifecycle, approached with varying degrees of rigor by different process models. Agile process models recommend Test Driven Development (TDD) as a key practice for reducing costs and improving code quality. The objective of this work is to perform a cost-benefit analysis of this practice. To that end, we have conducted a comparative analysis of GitHub repositories that adopts TDD to a lesser or greater extent, in order to determine how TDD affects software development productivity and software quality. We classified GitHub repositories archived in 2015 in terms of how rigorously they practiced TDD, thus creating a TDD spectrum. We then matched and compared various subsets of these repositories on this TDD spectrum with control sets of equal size. The control sets were samples from all github repositories that matched certain characteristics, and that contained at least one test file. We compared how the TDD sets differed from the control sets on the following characteristics: number of test files, average commit velocity, number of bug-referencing commits, number of issues recorded, usage of continuous integration, number of pull requests, and distribution of commits per author. We found that Java TDD projects were relatively rare. In addition, there were very few significant differences in any of the metrics we used to compare TDD-like and non-TDD projects; therefore, our results do not reveal any observable benefits from using TDD.

Neil C. Borle
E-mail: nborle@ualberta.ca

Meysam Feghhi
E-mail: feghhi@ualberta.ca

Eleni Stroulia
E-mail: stroulia@ualberta.ca

Russell Greiner
E-mail: rgreiner@ualberta.ca

Abram Hindle
E-mail: hindle1@ualberta.ca

Department of Computing Science, University of Alberta, Edmonton, Canada

Keywords Test Driven Development · Human Factors in Software Development · GitHub Repositories · Continuous Integration

1 Introduction

Test Driven Development (TDD) is a practice advocated by agile software development methodologies, in which tests are written in advance of source-code development. These tests, intended to initially fail in the absence of any substantial implementation, effectively constitute a specification of the functionality and behavior of the software code, which can be tested as it is being developed (Beck, 2003). The provision of immediate, specific, and local feedback is believed to have a positive effect on code quality and on the rate at which code can be developed.

The TDD development practice requires that every test of functionality be written before the source code that implements that functionality (as an iterative process), but this degree of rigor may not be applied consistently, even by test-conscious developers (Beller et al, 2015a). Since different projects incorporate TDD to different degrees, its effects are also likely to vary. This paper presents a study that (1) explores the way TDD is actually practiced and (2) distinguishes how TDD-like (our variants of TDD; see Section 2.3.1) projects typically differ from non-TDD projects. More specifically, we study the prevalence and influence of TDD and TDD-like methodologies on software development processes in Java repositories hosted on GitHub in 2015. Further, we explore if there are detectable differences, in terms of five research questions (seen below), between repositories that practice TDD and those that do not. We later describe the approach and motivation for each question in Section 3.1.

RQ1: Does the adoption of TDD improve commit velocity?

RQ2: Does the adoption of TDD reduce the number of bug-fixing commits?

RQ3: Does the adoption of TDD affect the number of issues reported for the project?

RQ4: Is continuous integration more prevalent in TDD development?

RQ5: Does the adoption of TDD affect developer collaboration?

To understand the degree to which GitHub projects adopt TDD, we used the capabilities of Boa (Dyer et al, 2013) – a domain-specific language (DSL) and infrastructure that allows researcher to answer mining software repositories (MSR) questions at scale and efficiently. Boa contains an archived subset of GitHub (September 2015), allowing Boa to query massive amounts of data from real world software projects. In our work, we use Boa to investigate every Java repositories that existed on GitHub in 2015 (256,572 repositories in total), an otherwise daunting task. Within the Boa DSL, domain-specific types[1] contain the information of the various structures that are found in repositories, such as the repository

[1] Domain-specific types `http://boa.cs.iastate.edu/docs/dsl-types.php`

itself, its revisions and its files. Researchers can then iterate through these domain-specific types and aggregate or collect results from them. For example, in this work we iterate through each projects files ASTRoot (a container for the files abstract syntax tree) in order to inspect the contents of files and determine if they include import statements for testing frameworks. These domain-specific types also allow us to inspect which Java classes have been defined in different files, as well as which Java classes have been instantiated in those files. Ultimately this means that we can use Boa to look at the abstract syntax trees (ASTs) of source and test code to determine if they follow a testing process.

In principle, TDD requires developers to follow a comprehensive set of practices. In practice, however, software projects do not always follow the prescribed practices to the letter. This is why our work considers two relaxations of TDD. First, we allow test files to be committed within some time Δ (1 hour, 1 day, or 1 week) after committing the source files, relaxing the requirement that test files be written before the code that they test. Second, we allow for some source-code files (10%, 20%, 30% ...) to not have corresponding tests. This resulted in 40 different TDD-like variants, each with a different combination of time and code coverage relaxations.

To generate a baseline for each of the 40 different TDD-like variants against which to compare TDD practices, we assembled repositories sets that were representative of software repositories that incorporate testing. The key requirement for this task was to develop a method for assembling these "control repository sets" so that they were *comparable* to the TDD-like repositories. To create a control set for a given TDD-like repository set, we first used K-means clustering to group the TDD-like repositories by three different size criteria (thus representing each repository as a 3-dimensional vector). Of the K clusters produced, we discarded the $K - 1$ clusters composed of the smallest repositories in terms of commits, authors and LLOC. For the remaining TDD cluster (the TDD-like set to be analyzed), we created the corresponding control set by pairing each TDD-like repository in the cluster with another equally-sized repository. This other repository was sampled from the larger set of repositories that contained at least one test file but did not contain any of the repositories from the TDD-like repository set for which we were creating a control set. Clustering was important in our study as it allowed us to control for the fact that many GitHub repositories have very few commits (Kalliamvakou et al, 2014).

Finally, as we are exploring many different questions, we will be performing a large number of statistical tests. We will therefore use the Holm-Bonferroni method (Holm, 1979) to reduce the chance of false positives, with an initial critical value for tests will be $\alpha = 0.05$.

Section 2 reviews related work. Sections 3, 4 and 5 explain our methodology for this study, report on our findings, and reflect on the threats to the study's validity. Finally, Section 6 concludes with a summary of our work and the lessons we learned from it.

2 Related Work and Background

In 1999, the practice of TDD in was introduced by Beck *et al.* in "Extreme Programming Explained" (Beck and Andres, 2004). Beck later described test driven development in more detail in “Test-driven development: by example” Beck (2003). Since then, many researchers have worked to elucidate how TDD affects software development (Jeffries and Melnik, 2007). This section summarizes the most recent works that are relevant to our research questions. Section 2.3 summarizes works that focus on detecting and quantifying TDD effects, or are related to how we identify TDD; Section 2.4, on works that look at continuous integration and provide context for our clustering approach; and Section 2.5, on background on correcting for multiple comparisons.

2.1 Mining Git and Github

Git (Bird et al, 2009; Kalliamvakou et al, 2014) is a distributed version control system that enables asynchronous collaboration. It superseded other centralized version control systems such as CVS or Subversion in popularity but created distinct challenges and problems in its move to a distributed structure. Bird et al (2009) describe the dangers of mining Git repositories in the context of mining software repositories research. They note that repositories can be rewritten, commits can be intentionally removed or collapsed into super commits via re-basing. Furthermore, repositories typically contain only successful commits, whereas failed branches can be removed or simply not shared. Much development can occur in remote locations with only a small amount of the effort finally pushed and merged to the main branch of a project. A git commit history should be considered potentially lossy and it could be pruned or parts of it re-based together. Merges can also be silently committed.

Kalliamvakou et al. (Kalliamvakou et al, 2014) in their work the “Promises and Perils of mining GitHub” describe many challenges that empirical software engineering researchers will face with mining software projects, their artifacts and history, from GitHub. Specific perils they highlighted when studying Github repositories included duplicate projects with little or no changes, student assignments, partial projects, mirrors of other projects. Other perils include the granularity of projects versus repositories. Project perils were that many projects were personal projects, had few commits, or were inactive. Construct validity threatening perils were that successful pull requests are not consistently identified and that the repositories on Github do not capture all of the software development activities, including the writing of code due to rebasing.

2.2 Testing in practice

In 2007, Hindle *et al.* described a taxonomy for classifying revisions based on the types of files being changed in a revision (Hindle et al, 2007). The classes of this taxonomy include source revisions, test revisions, build revisions, and documentation revisions (STBD). While Hindle *et al.* did not directly address TDD in their work, we consider this taxonomy to be relevant to our study on TDD because it identifies source code revisions and test revisions that, we feel, are the revision classes most relevant to TDD, and are the two classes of revisions that our work primarily studies.

Another work that does not directly address TDD, but has implications on our study, was done in 2014 by Athanasiou *et al.*. These authors found that there is a positive correlation between factors such as test code quality, throughput, and productivity (Athanasiou et al, 2014). This is relevant to our work because it shows that an emphasis placed on testing can result in software development benefits. This work motivates the belief that a TDD approach, where testing is emphasized, should lead to benefits in software development.

2.3 TDD practice

Zaidman et al (2008) studied two different open-source repositories to determine if approaches such as TDD are detectable. They developed a method of associating source code with test code by relying on a naming convention where the word "Test" is added as a postfix to a test file corresponding to a similarly named source file. They also referenced the use of JUnit imports as a method of test file identification. Our study similarly considered the use of import statements, as well as this naming convention for associating test files to source files when detecting TDD. We differ by performing case-insensitive matching of the word "test" to a file name, instead of just considering "Test" as a postfix.

Beller et al (2015a) studied the prevalence of TDD practices among several developers by having them install a tool that monitored their practices in their integrated development environment (IDE). They found that TDD is rarely practiced by software developers. In their subsequent work, Beller et al (2015b) found that, in a group of 40 students, 50% spent very little time testing their code. This suggests that an analysis of GitHub repositories may yield only small numbers of test files in repositories.

Fucci et al (2016b) sought to identify whether TDD affected the number of tests written, the external quality of code (meeting functional requirements) or the productivity of developers when producing software. They explicitly measured developers who were asked to follow TDD. Building upon and replicating their previous work in multiple geographic locations, they observed that there was no statistical difference between TDD and a test-last development approach (TLD). Fucci et al (2016a) decomposed TDD into four different dimensions: granularity, uniformity, refactoring, and sequencing (where sequencing corresponds to the familiar test-first aspect of TDD). Their work showed that granularity and uniformity had the most influence on external quality and developer productivity, and that the order in which tests and source code were developed had a negligible influence. Similar to Fucci et al (2016a), we also consider sequencing as an indicator of TDD. However, our work diverges from theirs by studying sequencing when combined with class coverage and by considering a spectrum of adherence to these properties. Class coverage in this work refers to how often classes are reachable from test code.

The results of Fucci et al (2016b) suggest that if you look at TDD primarily as a test first methodology (as we do), differences will not be seen between TDD and other sound testing approaches that are not test first (such as TLD). However, because we compare TDD to repositories in general, we might see differences simply because the testing rigor of other repositories in general might be lower than that of TDD repositories. Our work complements Fucci *et al.* (2016a; 2016b) by studying the evidence of TDD in Github repositories.

2.3.1 Definition of Test Driven Development

Test driven development (TDD) in this paper will be the intent of following a process of combining test-first-development, refactoring, design, and specification. "Test first behavior" means writing tests before writing code. The intent is often to specify and to verify code once it is written. TDD also can involve refactoring, which might be done to enable writing of tests or to enable passing tests. This means that some code might be written first if it is already transitively called in a test case. Test driven development attempts to employ test-first and refactoring into a development loop whereby code is specified as tests and then verified by tests. That is, classes written should generally be covered by tests first rather than later.

One construct validity issue that this study faces is that our record of change, the git version control histories of Java projects, is not perfect and files can be added, modified, and committed at different times, or in different orders than recorded. Ordering of commits is not necessarily the ordering of development (Bird et al, 2009; Kalliamvakou et al, 2014). In a git history, test first could look like testing at the same time, or even testing later depending on how the git commits were formed.

TDD-like refers to behaviours depicted in version control histories that are emblematic of TDD but perhaps not perfectly, so it is unclear if they are truly and purely TDD. So while someone may be intentionally trying to follow TDD practices, they might adhere to TDD only partially, and their commit history might show this in different ways depending on their commit behaviours. In particular, TDD-like behaviours must include testing, but coverage can be lower and lag time between source and test commits can be permitted, where lag time (time Δ) is the amount of time that passes, *after* the source file is committed, *before* the test file is committed[2]. TDD-like behaviour could be the result of TDD behaviour but inconsistent commit behaviour, or the partial application of TDD to the project, perhaps by a subset of developers. We consider TDD-like because we do not have a perfect record of the true change history of a project. Furthermore to ensure that potentially corrupted sequences are not leading the research astray, we use class coverage to ensure that tests actually call the classes that are committed to the repository. A project following TDD should cover classes with test code before they are actually written.

2.4 Continuous Integration (CI)

In 2016, Santos and Hindle studied the relationship between build failure in GitHub repositories and the "unusualness" of the commit messages that they found in

[2] Here, we consider four time Δ categories (0 minutes, 1 hour, 1 day, 1 week). Repositories are assigned to the category with the shortest time Δ that is greater than all the time Δs found in that repository – hence if the largest Δ for any code in a repository is 5 minute, this entire repository would be in the "1 hour" category, and if the longest was 3 days, it would be in "1 week".

those repositories. To explore this, they looked specifically at repositories using Travis-CI, a tool for continuous integration that is widely used in the open-source community[3] (Santos and Hindle, 2016). Our work will determine how popular Travis-CI is among practitioners of TDD and TDD-like software development.

Vasilescu et al (2015) studied the usage of continuous integration, focusing on Travis-CI, in 223 "large and active python, ruby, and Java repositories on GitHub. They found that 92.3% of these repositories were configured to use Travis-CI, but of those configured repositories, only about half actually had recorded Travis-CI builds. We also study the usage of Travis-CI for automated continuous integration in GitHub repositories, but ask a distinct question: how prevalent is Travis-CI in repositories utilizing TDD, as compared to other repositories in general. This methodological choice of using Travis-CI to represent continuous integration is supported by Vasilescu et al (2015)'s statement that Travis-CI is "arguably the most popular CI service on GitHub".

2.5 Determining the Clustering Parameter K

To assess the quality of clusters generated from any clustering method such as K-means, we use a visualization technique known as the silhouette plot (Rousseeuw, 1987), which uses

$$s(i) \quad = \quad \frac{b(i) - a(i)}{\max\{\, a(i),\, b(i)\,\}} \tag{1}$$

where $s(i)$ is the silhouette of the i^{th} data point, $a(i)$ is the average dissimilarity (based on a given distance metric) between the i^{th} data point and the other members of its cluster, and $b(i)$ is the minimum average dissimilarity between the i^{th} data point and the other clusters that exist in the partitioned space. We can use this equation to obtain the average silhouette width from all the silhouette values for each cluster to determine the quality of the clusters individually. Alternatively, we can find the average silhouette width across all clusters to determine the quality of a particular k partition of the data space using K-means clustering, for example. We use silhouette for clustering tuning parameters when clustering repositories to determine the k number of clusters.

2.6 Correcting For Multiple Comparisons

Throughout our study we perform multiple statistical tests, which can increase the overall chance of Type 1 Errors (false discoveries) (Aickin and Gensler, 1996). One approach, the Bonferroni Procedure (Aickin and Gensler, 1996) is designed to address this family wise error rate (FWER) issue (Hochberg, 1988) by dividing the significance threshold by the total number of tests performed.

As an alternative, the Holm-Bonferroni Procedure offers FWER correction that is uniformly more powerful, in a statsitical sense, while still insuring the family wise error at significance level α. This is done by comparing *p-values* to an adjusted significance level $\frac{\alpha}{n-i+1}$, where n is the number of tests performed and i is the i^{th} index in the sorted list of p-values when the inequality $P_i \leq \frac{\alpha}{n-i+1}$ holds, where P_i is the *p-value* for the first rejected hypothesis (Aickin and Gensler, 1996).

[3] `https://travis-ci.org/`

3 The Study Method

In this section, we describe our research questions as well as our approach for analyzing GitHub repositories to assess the potential impacts of using TDD. Section 3.1 describes our research questions and their motivations; Section 3.2 describes the data used; Section 3.3 describes our criterion for determining if a repository practices TDD; Section 3.4 describes our filter for non-trivial TDD repositories; Section 3.5 describes our procedure for creating control sets for statistical comparison; Section 3.7 describes how we ensure the quality of our analysis; and Section 3.8 addresses how we correct for statistical error.

3.1 Our Research Questions

RQ1: Does the adoption of TDD improve commit velocity?
Why: Some practitioners of TDD argue that TDD, once a three step cycle (Writing a failing test, Making the test pass, Refactor), is now practiced as a four step cycle (including a committing step). It is argued that the commit step has arisen from the growing popularity of version control systems (DevIQ, 2017; Lakeview Labs, 2017). In the work of Fucci et al (2016a) it is noted that uniform, short grained cycles are likely the aspect of TDD that improve external quality and productivity. Together, we use these two ideas as motivation for measuring commit velocity. We wish to see if the TDD repositories available through GitHub have faster commits (faster cycles), which would then imply greater productivity.
Approach: We extract the timestamps associated with all the commits collected and then look at the average of the timestamp differences (Δs) between commits in each repository.

RQ2: Does the adoption of TDD reduce the number of bug-fixing commits?
Why: Developers have asserted that one of the benefits of TDD is that it reduces the number of bugs introduced into software (Winter, 2016). We use this to motivate an investigation into the number of bug fixing commits present in software repositories. If fewer bugs have been introduced into the software, then perhaps there will be fewer bugs that need to be fixed.
Approach: The following regular expression [4] is used to identify bug-fixing commits in the repositories:

```
/.*((solv(ed|es|e|ing))|(fix(s|es|ing|ed)?)
|((error|bug|issue)(s)?)).*/i
```

We consider this count to be an approximation of the number of bugs that had existed and been addressed in that repository.

RQ3: Does the adoption of TDD affect the number of issues reported for the project?
Why: In a similar way to RQ2, we motivate this research question from the assertion that TDD reduces the number of bugs during software development (Elliott,

[4] modified from Boa examples found at http://boa.cs.iastate.edu/

2016), even up to 40% - 80% of bugs (Jeffries and Melnik, 2007). If TDD leads to fewer bugs then perhaps there will be fewer bugs reported.
Approach: We count the number of issues (obtained through the GitHub Application Program Interface) associated with each repository from the two sets and quantify any observed differences between them.

RQ4: Is continuous integration more prevalent in TDD development?
Why: TDD and continuous integration (CI) have been a topic of discussion for other vendors, including Microsoft (Budhabhatti, 2008; Weiss, 2017; Travis-CI, 2017), who seek to sell test and integration tools as well. The Travis-CI website (Travis-CI, 2017) claims that Travis-CI helps to "foster test-driven development". Therefore, we would expect to see a correlation between TDD users and the use of Travis CI.
Approach: We use a part of a publicly available Boa script[5] to determine if a Travis-CI `travis.yml` file was present in a repository. While we focus specifically on Travis-CI, we use this as an approximation for counting the number of repositories that were using continuous integration.

RQ5: Does the adoption of TDD affect developer collaboration?
Why: TDD is a process that asks a lot from the developers. Such a process could aide collaboration in terms of verification and communication of requirements (Brack, 2016), but it could hinder collaboration in terms of a high bar to meet in order to share code with TDD project. Furthermore in later literature TDD is suggested to help developers and customers discuss hard requirements via acceptance tests (Pugh, 2010).
Approach: Here, we use the GitHub API to extract the pull requests associated with each of the repositories. Following this, we examine the number of pull requests present for each repository and considered this as an indirect measure of the level of collaboration that occurred in that repository. TDD could ease collaboration by making it clear that a contribution needs to include tests before it is considered. Alternatively, TDD could harm collaboration as its requirement to include tests makes it harder to contribute a TDD driven project. We also explored the proportion of developer contributions in each repository, to see whether these contributions are shared evenly or are skewed (Vasa et al, 2009).

3.2 The Data Set

We used the Boa infrastructure (Dyer et al, 2013) to obtain $256,572$ Java repositories from a copy of GitHub, archived September 2015. From each of these repositories, we obtained the repository URL, its commit logs, its commit timestamps, and its revisions. For each revision, we collected the names of the Java files created in it. Further, we used GitHub's application programming interface (API) to obtain the repository issues and pull requests.

Of these $256,572$ Java repositories, $41,301$ (16.1%) contain test files. Table 1 provides the complete distribution of TDD repositories, whose columns represent different levels of class coverage (to be defined later), and whose rows represent

[5] `http://boa.cs.iastate.edu/boa/?q=boa/job/public/30188`

the maximum elapsed time permitted between the creation of source code files and test files. Here we use the term "TDD" loosely because we report the number of repositories found at different levels of stringency; this usage allows us to distinguish these sets of repositories from their corresponding control sets.

Table 1 Distribution of Repos for Different Time Δs and Class Coverage Relaxations. Each cluster is labeled so that it can be referenced later. For example, '9d' (bottom right) is the cluster corresponding to the 10^{th} coverage interval (here "$> 90\%$") and the 4^{th} time Δ (here, "Week").

	$\leq 10\%$	$11-20\%$	$21-30\%$	$31-40\%$	$41-50\%$
None	2623 (0a)	2729 (1a)	1937 (2a)	1955 (3a)	2129 (4a)
Hour	2710 (0b)	2828 (1b)	2021 (2b)	2041 (3b)	2223 (4b)
Day	2962 (0c)	3145 (1c)	2262 (2c)	2298 (3c)	2454 (4c)
Week	3424 (0d)	3658 (1d)	2675 (2d)	2664 (3d)	2802 (4d)
	$51-60\%$	$61-70\%$	$71-80\%$	$81-80\%$	$> 90\%$
None	698 (5a)	859 (6a)	649 (7a)	233 (8a)	1991 (9a)
Hour	744 (5b)	904 (6b)	676 (7b)	245 (8b)	2054 (9b)
Day	830 (5c)	1003 (6c)	768 (7c)	274 (8c)	2136 (9c)
Week	1006 (5d)	1149 (6d)	870 (7d)	320 (8d)	2230 (9d)

While Table 1 shows the entire distribution of repositories identified as TDD, we only actually analyze a subset of the repositories in each category (reported in Table 2) to ensure that analyzed repositories have sufficient numbers of commits, authors, and logical lines of code. For example, this analysis only includes repositories that have at least one commit. Finally, note that the TDD sets are disjoint from their control sets.

3.3 Recognizing Repositories That Practice TDD

For this study, we consider the following three characteristics of software projects as evidence of TDD adoption: (a) the inclusion of test files, (b) the development of tests before the development of the source code that these tests exercise, and (c) the high coverage of the overall source code by tests. While these features characterize repositories that truly practice TDD, we realize that those attempting to practice TDD may not do it perfectly, thus we also study repositories that exhibit TDD with varying degrees of success.

The rationale behind these criteria is that Kent Becks book "Test driven development by example" (Beck, 2003) he describes TDD as two bullet points:

1. "Dont write a line of new code unless you first have a failing automated test"
2. "Eliminate duplication"

Focusing on the first point, we reason that it implies two things. The first is that you need to write a failing test before you write the code it tests, and secondly, that all code to be written must have had test code written first. Hence our test first and coverage characteristics. Our final characteristic (including tests) is the result of the focus of our study, comparing the TDD testing framework to other testing frameworks in general. Comparing TDD to repositories that do no test defeats this purpose.

a) Finding repositories with test files

To determine how many repositories included test files, we used the abstract syntax trees (available through Boa) to obtain all import statements in each Java file for each repository. This included import statements that matched the following regular expression where JUnit[6], TestNG[7] and Android test[8] are frameworks and tools for implementing test cases.

```
‘^(org\.junit\.\*)|
  (org\.junit\.Test)|
  (junit\.framework\.\*)|
  (junit\.framework\.Test)|                    (2)
  (junit\.framework\.TestCase)|
  (org\.testng\.*)|
  (android\.test\.*)$‘
```

Once a file was found to contain one of these imports, we considered the repository as containing test files, and thus meeting the first criterion for being considered as following TDD.

The regular expression that we apply is simply used to extract the relevant imports obtained from the ASTs. Had we not used these imports and the regular expression, we would have had to assess whether the Java class created in the source file extended one of the testing frameworks. If it did, it would still have needed to import the testing framework, and so this approach would be considered a subset of our approach.

b) Evaluating if test files are written first

This process involved the following four steps and excludes repositories that contained no files with test imports.

Step 1. For each of the repositories, we partitioned the Java files into two sets: The first set included files identified as test files, because their filename matched regular expression (3) and their imports matched regular expression (2). The second set contained the remaining Java files.

```
/.*test.*\.java/i                              (3)
```

Step 2. For each file in each set, we identified its creation time as the timestamp of the revision in which it was created.

Step 3. For each of the test files we found matching, or similarly named files, under the assumption that Java programmers typically name their test files according to their source-code files. For example, “someFile.java” might have a corresponding test file called “TestSomeFile.java” or “someFileTest.java”. If we found no matching file, we searched for files with similar file names, using on the Python Standard Library object `difflib.SequenceMatcher()` for string comparison. We

[6] http://junit.org/junit4/
[7] http://testng.org/doc/index.html
[8] http://developer.android.com/tools/testing/index.html

set a similarity threshold of 0.8, where 0 is completely dissimilar and 1 is exactly the same. This threshold was chosen because it could match, for example, words like "search" with "searching". Here the file "searchTest.java" would be similar to "searching.java". Finally, if no matching or similar files were found for a test file, that test file was ignored in this step.

Step 4. Having identified pairs of test and corresponding source-code file, we then ensured that the timestamp associated with the test file was either older or the same as the source file(s). This ensured that the test file was either committed before or at the same time as the source file(s).

We also considered repositories where we permitted a grace period (hour, day or week) for the creation of the test file after the source file. Here, only Step 4 differs in the above procedure. For example, in the case of a 'one day grace period', we required that tests are written within at most one day of the source code, for all files in a repository – *i.e.*, $((\text{timestamp}_{test} - \text{timestamp}_{source}) \leq (60 \times 60 \times 24)$ seconds).

c) Evaluating test class coverage

For this facet of TDD, we again used JUnit, Android test and TestNG imports as identifying test files and consider "class coverage", where all the classes defined in source files are referenced in at least one test file. We specifically consider class level coverage for several reasons. First, classes are immediately compilable and we do not have the uncertainty as to whether or not methods are reachable or mutable. Further, method level coverage has the added complexity of dealing with automatically generated getters and setters. Finally, we also view package level coverage as being too coarse grained because only a single test is required to cover an entire package. Note that we do not explicitly look at inner classes because we view them as handlers within the class. To determine class coverage, we used the ASTs provided in the Boa framework. Each Java file in a repository is associated with an ASTRoot AST Type, a container class that holds the AST (Dyer et al, 2013). After obtaining this object, Boa allows navigation through the namespace of the source file (the Java package) so that the declarations made in the namespace can be observed. In our case, when looking through source files we only recorded class declarations in the namespaces provided by the ASTRoot. To deal with test files, we examined and recorded all the types that were referenced in the test file. This includes types that were explicitly referenced in the initial declaration of a variable as well as any type changes that occur when a variable is reassigned. For example, if a variable of type "MyInterface" is declared and later assigned an instance of a class "ImplementsMyInterfaceClass", our procedure would detect the use of this class.

Finally combining the knowledge of types used in test files with classes declared in source files, we defined the class coverage percentage of a repository to be the percentage of classes defined in source files that were referenced in at least one test file in the most recent version of the repository.

3.4 Identifying Substantial Repositories

In “The Promises and Perils of Github” (Kalliamvakou et al, 2014),the authors note that there are issues with researchers working on GitHub data. In particular they highlight the fact that many GitHub repositories are simply personal projects, and that many repositories are inactive. In order to avoid using these repositories we came up with a set of three distinguishing criteria: logical lines of code (LLOC), number of authors, and number of commits. These criteria allow us to cluster repositories, thereby keeping the *significant* software projects and avoiding forks, student projects, personal projects, and other sources of noise on GitHub. Note, we use significant to denote repositories with the greatest quantities of LLOC, commits, and authors.

We motivate the first two criteria from “The Promises and Perils of GitHub”, which advocates to look at the number of commits in a repository when determining if it is active, and to determine if a project is a personal project by looking at the number of committers (authors). Finally LLOC is included in an effort to further increase the likelihood that the repositories that we are analyzing are not inactive. Together, we use these criteria to filter out repositories that would otherwise skew our results.

To identify the subset of the TDD repositories for each group in Table 1 that were *significant* software projects, we applied K-Means clustering, where each repository is described in terms of the criteria mentioned above, to partition the repositories into different groups. One issue with using K-Means clustering is its requirement of specifying the number of clusters K initially. Here, we considered a range of K values from 2 to 10, and used the silhouette width for each clustering to determine which K value provided the best amount of intra-cluster compactness and inter-cluster separation. During clustering we restricted the maximum value to K to be 10. This was done because when larger values of K were permitted (e.g. $K = 11$), clusters were generated that had fewer than 30 samples. This was undesirable for conducting statistical tests due to a lack of power.

Finally, because the distributions of commits, authors and lines of code are highly right-skewed, we used their natural logarithm to prevent larger repositories from being fragmented into small isolated clusters.

Figure 1 shows an example Silhouette plot that illustrates how K is selected when clustering. Here, as an example, we select $K = 2$ and $K = 9$ for two arbitrary TDD clusters.

3.5 Control Set Construction

Once we obtained TDD repositories of non-trivial size, we next need to produce control sets of repositories of comparable size, against which to compare the TDD repositories. Again, recall the “peril of GitHub” (Kalliamvakou et al, 2014), that many repositories are underdeveloped and inactive. Therefore, if we do not control for size then this size may be a confounding factor in our study, because we may be comparing larger active repositories against smaller inactive ones. To make a fair comparison between the TDD repository clusters and a control set representing repositories in general, we attempt to sample the control set to match the distribution of the size features of the TDD set.

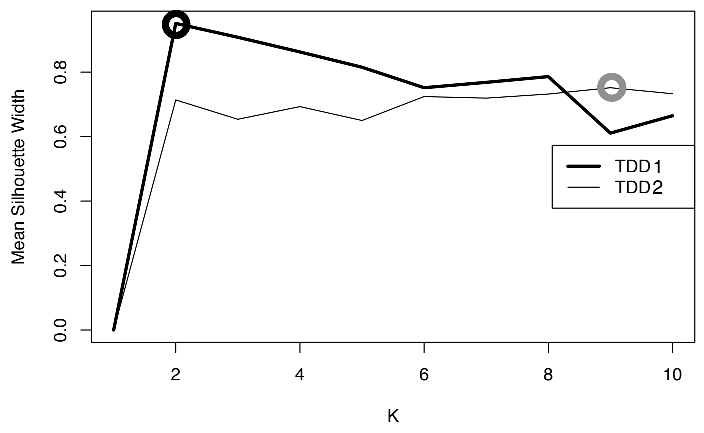

Fig. 1 Example of Mean Silhouette Width for selecting K. For a given curve (e.g. TDD1) representing a set of repositories, the value of K which results in the largest mean silhouette value is the K which best approximates the true number of clusters in the data. These are indicated by circles in the figure.

To create a control set for any particular TDD repository set, we first created a base pool of repositories by taking the set difference between the 41,301 repositories containing test files and that particular set of TDD repositories. Note, this was done because we intend to compare TDD-like development practices against other testing practices in general. Therefore, comparing with repositories that show no evidence of testing is beyond the scope of this study. From this base pool, we then sampled a set of equal cardinality by iterating over our TDD set and sampling a repository minimizing Euclidean distance (L^2 norm)

$$d(\mathbf{x}, \mathbf{y}) \quad = \quad \|\mathbf{x} \ - \ \mathbf{y}\|_2 \tag{4}$$

in each iteration (breaking ties randomly). Note that $\mathbf{x}$ and $\mathbf{y}$ in Eqn 4 refer to vectors that describe repositories in terms of their size features.

In Table 2 we provide details describing the distributions of our TDD sets (produced from clustering) and their corresponding control sets. Here, the percentiles show how closely the control set distributions match the TDD distributions for each cluster pair.

3.6 Visualizing our Results as Heat Maps

In our work, we present our results as heat maps (such as Fig 7), where each square is a statistical test comparing one of the aforementioned clusters to its corresponding control set. Here, the "Time Window" axis corresponds to a particular grace period, and "Coverage" axis corresponds to percentage of class coverage (both concepts introduced in Section 3.3). This means that squares represent a comparison between repositories with some Coverage/Time Window combination (e.g. 10-20% coverage and a grace period of 1 day), and the control repositories paired with them. In each square, we use black to means the centre of the control set was significantly larger than the TDD set, grey if there is no statistical

Table 2 TDD Repository Clusters and Controls. The "Cluster" label refers to the parameters described in Table 1, where the corresponding labels can be found. To explain the other column headers (*i.e.*, top row): The A, C and L represent the numbers of authors, commits and logical lines of code respectively; the subscript 25, 50 and 75 refer to the 25^{th}, 50^{th} and 75^{th} percentiles of each A, C, L distribution. Size refers to the number of repositories in a cluster. The pair of entries (TDD, Control) of each $[r, c]$ cell shows column c's characteristic of the largest post-clustering TDD set associated with Cluster r, with its corresponding control set. Collectively, the similarity between these pairs of values illustrates the lack of disparity between TDD and control sets.

Cluster	Size	A_{25}	A_{50}	A_{75}	C_{25}	C_{50}	C_{25}
0a	511	(3, 2)	(5, 3)	(8, 7)	(55, 48)	(108, 101)	(256, 237)
0b	537	(3, 2)	(5, 3)	(8, 6)	(56, 50)	(109, 104)	(247, 236)
0c	626	(3, 2)	(5, 4)	(8, 6)	(58, 52)	(108, 104)	(237, 229)
0d	860	(3, 2)	(4, 3)	(7, 6)	(53, 48)	(99, 92)	(209, 208)
1a	938	(2, 1)	(2, 2)	(3, 3)	(16, 15)	(27, 27)	(49, 50)
1b	994	(2, 1)	(2, 2)	(3, 3)	(16, 15)	(28, 28)	(52, 52)
1c	1153	(2, 2)	(2, 2)	(4, 3)	(18, 17)	(31, 30)	(57, 57)
1d	1387	(2, 2)	(3, 2)	(4, 3)	(20, 19)	(37, 37)	(68, 67)
2a	676	(2, 1)	(2, 2)	(3, 3)	(13, 12)	(23, 23)	(41, 42)
2b	724	(2, 1)	(2, 2)	(3, 3)	(13, 13)	(23, 23)	(42, 44)
2c	870	(2, 1)	(2, 2)	(3, 3)	(15, 15)	(25, 25)	(46, 46)
2d	1059	(2, 1)	(2, 2)	(3, 3)	(18, 18)	(31, 31)	(55, 54)
3a	289	(2, 1)	(3, 2)	(4, 3)	(17, 17)	(26, 25)	(44, 44)
3b	729	(2, 1)	(2, 2)	(3, 3)	(8, 8)	(15, 15)	(30, 29)
3c	772	(2, 1)	(2, 2)	(3, 3)	(10, 9)	(18, 18)	(37, 37)
3d	945	(2, 1)	(2, 2)	(3, 3)	(13, 13)	(25, 25)	(46, 46)
4a	718	(2, 2)	(2, 2)	(3, 2)	(6, 6)	(11, 11)	(26, 25)
4b	890	(1, 1)	(2, 2)	(3, 2)	(9, 9)	(14, 14)	(24, 24)
4c	1016	(1, 1)	(2, 2)	(3, 2)	(10, 10)	(15, 15)	(27, 27)
4d	1192	(1, 1)	(2, 2)	(3, 3)	(12, 12)	(18, 19)	(34, 35)
5a	258	(2, 1)	(2, 2)	(3, 2)	(9, 9)	(16, 16)	(29, 28)
5b	281	(2, 1)	(2, 2)	(3, 3)	(10, 10)	(18, 17)	(30, 30)
5c	332	(2, 1)	(2, 2)	(3, 3)	(11, 11)	(20, 20)	(33, 33)
5d	353	(2, 2)	(2, 2)	(3, 3)	(12, 12)	(21, 21)	(49, 45)
6a	333	(2, 1)	(2, 2)	(3, 2)	(8, 8)	(14, 15)	(24, 25)
6b	366	(2, 1)	(2, 2)	(3, 2)	(8, 8)	(14, 15)	(25, 26)
6c	385	(2, 1)	(2, 2)	(3, 3)	(10, 10)	(18, 18)	(29, 30)
6d	517	(1, 1)	(2, 2)	(3, 3)	(12, 12)	(19, 19)	(31, 32)
7a	190	(2, 1)	(2, 2)	(3, 2)	(7, 8)	(16, 16)	(28, 29)
7b	189	(2, 2)	(2, 2)	(3, 3)	(7, 8)	(15, 16)	(32, 32)
7c	127	(2, 1)	(3, 2)	(4, 3)	(16, 17)	(24, 24)	(41, 41)
7d	322	(2, 2)	(2, 2)	(3, 2)	(6, 6)	(12, 13)	(24, 25)
8a	100	(1, 1)	(2, 2)	(3, 2)	(11, 11)	(15, 15)	(24, 24)
8b	117	(1, 1)	(2, 2)	(3, 2)	(9, 9)	(15, 15)	(22, 23)
8c	135	(1, 1)	(2, 2)	(2, 2)	(10, 10)	(15, 15)	(24, 24)
8d	43	(3, 2)	(4, 3)	(5, 3)	(14, 15)	(22, 22)	(46, 46)
9a	751	(1, 1)	(2, 2)	(2, 2)	(6, 6)	(9, 9)	(16, 16)
9b	792	(1, 1)	(2, 2)	(2, 2)	(6, 6)	(9, 9)	(16, 16)
9c	851	(1, 1)	(2, 2)	(2, 2)	(6, 6)	(9, 9)	(17, 16)
9d	873	(1, 1)	(1, 1)	(2, 2)	(7, 7)	(10, 10)	(18, 18)

Cluster	Size	L_{25}	L_{50}	L_{75}
0a	511	(1341, 1342)	(3360, 3363)	(11179, 11200)
0b	537	(1332, 1334)	(3285, 3285)	(10778, 10815)
0c	626	(1280, 1289)	(2907, 2918)	(9122, 9138)
0d	860	(1100, 1105)	(2318, 2309)	(7365, 7358)
1a	938	(322, 322)	(711, 710)	(1675, 1674)
1b	994	(322, 322)	(703, 703)	(1666, 1667)
1c	1153	(308, 310)	(681, 682)	(1593, 1592)
1d	1387	(353, 353)	(763, 764)	(1675, 1674)
2a	676	(247, 248)	(580, 581)	(1364, 1365)
2b	724	(234, 234)	(558, 556)	(1338, 1340)
2c	870	(249, 249)	(579, 580)	(1325, 1324)
2d	1059	(301, 301)	(680, 681)	(1418, 1418)
3a	289	(262, 263)	(637, 637)	(1507, 1505)
3b	729	(140, 140)	(347, 346)	(924, 923)
3c	772	(158, 158)	(384, 384)	(963, 964)
3d	945	(208, 209)	(459, 460)	(1038, 1039)
4a	718	(106, 105)	(276, 276)	(712, 711)
4b	890	(117, 117)	(272, 272)	(702, 702)
4c	1016	(126, 126)	(288, 288)	(708, 709)
4d	1192	(160, 160)	(369, 369)	(818, 817)
5a	258	(216, 217)	(432, 432)	(1037, 1037)
5b	281	(221, 222)	(432, 432)	(1095, 1095)
5c	332	(227, 227)	(427, 428)	(1032, 1032)
5d	353	(252, 252)	(535, 536)	(1342, 1336)
6a	333	(116, 116)	(293, 294)	(721, 719)
6b	366	(117, 118)	(309, 309)	(722, 719)
6c	385	(135, 135)	(361, 361)	(799, 800)
6d	517	(171, 170)	(389, 388)	(842, 842)
7a	190	(120, 120)	(292, 291)	(591, 590)
7b	189	(120, 120)	(291, 290)	(555, 554)
7c	127	(192, 192)	(355, 355)	(636, 635)
7d	322	(122, 122)	(273, 273)	(544, 544)
8a	100	(187, 187)	(329, 329)	(690, 691)
8b	117	(173, 173)	(315, 315)	(684, 683)
8c	135	(174, 174)	(283, 283)	(536, 537)
8d	43	(184, 183)	(421, 419)	(1032, 1035)
9a	751	(33, 33)	(82, 82)	(177, 177)
9b	792	(33, 33)	(81, 82)	(177, 177)
9c	851	(36, 36)	(84, 84)	(180, 180)
9d	873	(48, 48)	(104, 104)	(205, 205)

difference between the control and TDD repositories, and light grey if that the TDD set's center was significantly larger than the control set. (See the colour bar legend shown on the right hand side of the figure.) Recall we use Holm-Bonferroni correction for all the comparisons made in this work; see discussion above. Because of the large number of non-significant results in this work (leaing to our resulting conclusions), we used this measure to make sure that an overly conservative multiple correction technique was not a large factor in our findings.

3.7 Sanity checks

To quantify that test files were named as expected, we looked at the co-occurrence of this naming convention and used of a testing framework. We found that of 41302 repositories that contained test files (files with JUnit, TestNG or Android Test import statements), 40368 repositories also contained test files that matched our test file name regular expression and included one of these imports. Furthermore, 31747 of these repositories contained at least one occurrence of a test file name exactly matching a source file name once the case insensitive suffix or prefix test was removed from the test name.

The authors also wanted to quantify how well the bug fixing commit finding regular expression worked. To do this, the regular expression was applied to all the commit logs in the GitHub data to obtain a set of 530410 commit logs that matched the regular expression. From these, 100 commit logs were randomly sampled and then manually inspected to see if they referred to a bug fix or if they were erroneously collected by the regular expression. It was found that 79 of the commits were correctly identified as bug fixes and 21 were not. Example commit logs are shown in Table 3.7.

Also important to note is that the bug fixing commit finding regular expression is a modified version of the one provided as a built in function (`isfixingrevision`) Boa. This Boa function was originally presented by Dyer as a method for finding revisions that were likely to be bug fixing revisions (Dyer, 2013).

Table 3 Examples of Manually Labeled Commits. Each log message is either labeled with YES or NO as to whether it represents a bug fixing commit.

BFC?	Log Message

YES	`RESOLVED - bug 235243: [patch]`
	`Generalize the UI legend dialog for reuse by clients.`
	`https://bugs.eclipse.org/bugs/show_bug.cgi?id=235243`

YES	`Fix issue for passed arguments by references to method/function`

NO	`Revert "minor error change"`
	`This reverts commit 06e1ac494c64ae039cd254f8b1ea45ee5a4435bb.`

NO	`Adding LMDebug, a simple sketch for debugging two beams`
	`LMDebug relies very little on classes, and is easy to follow precedurally.`
	`At this time, LMDebug works mostly fine.`

To ensure that control repositories included test files, we used the Wilcoxon rank sum test with Holm-Bonferroni correction to visualize any differences between TDD and control groups, over different time windows and coverage for TDD. Figure 2 shows that there are no differences for any of the TDD variants – *i.e.*, that all statistical tests show that there is no significant difference between the number of control repositories that contain test files and the number of TDD repositories containing test files. This is exactly as the authors expected because all repositories in the control set were sampled from the distribution of repositories containing at least one test file.

To test whether our clustering approach created control sets whose sizes match our TDD sets, we performed statistical (wilcoxon) tests to determine if any significant difference could be found between their respective lines of code, number of

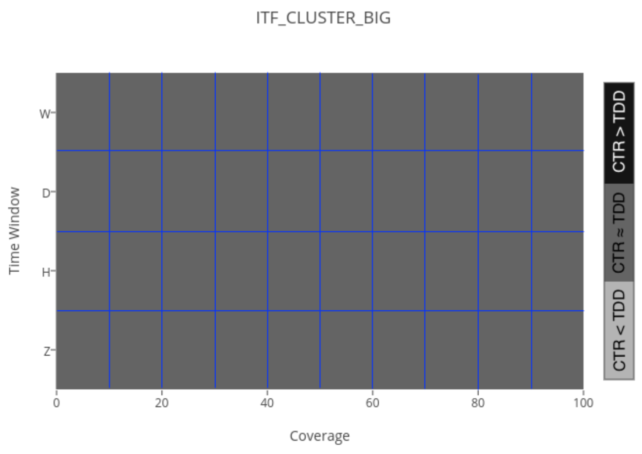

Fig. 2 Wilcoxon Tests for Including Test Files

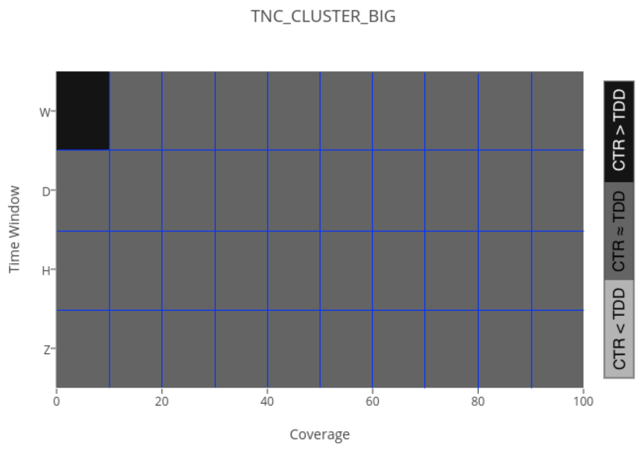

Fig. 3 Wilcoxon Tests for Total Numbers of Commits

authors or number of commits for each of the TDD variants. The results of these tests show that there are no differences for logical lines of code (LLOC) for any of the groups (Figure 5) but that two clusters showed differences in the number of authors and commits when compared to their controls (Figures 3 and 4). This is likely the result of not having repositories in our sampling set that could match all three dimensions for each TDD repository. The implication of these sanity checks is that the results obtained for these two clusters, in particular, may be biased by the size discrepancies between the clusters and their corresponding controls.

Finally, note that Figure 6 shows differences between the total number of test files in TDD repositories and their controls. While we require that all control repositories had to have at least one test file, the number of test files was allowed

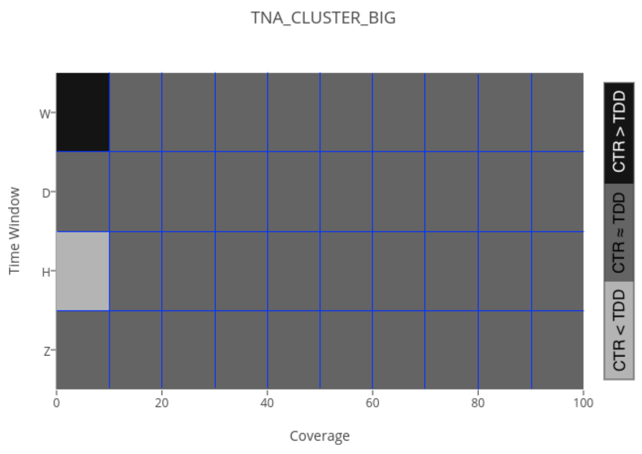

Fig. 4 Wilcoxon Tests for Total Numbers of Authors

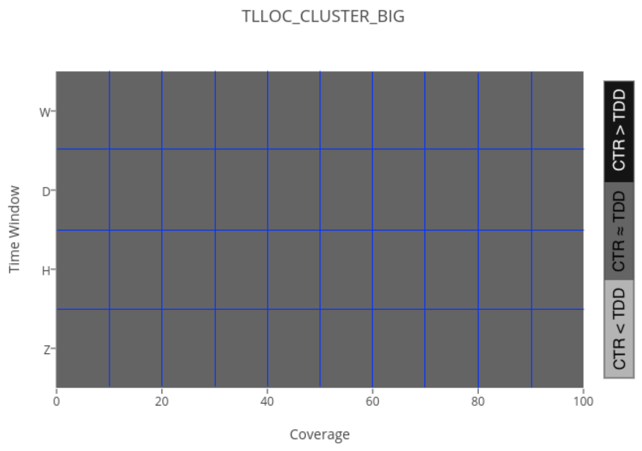

Fig. 5 Wilcoxon Tests for LLOC

to vary, giving us a representative sample of general GitHub repositories that have done some testing of their code (See Table 4 for mean and median group values). In the general case, the TDD repositories had more test files comparatively, with the exception of TDD repositories with extremely low class coverage.

3.8 Family-Wise Error Rate Correction

When answering each question, we used the Holm-Bonferroni Procedure to deal with the family-wise error rate (FWER), as this is uniformly more powerful than the obvious Bonferroni procedure. This is important as previous work suggests

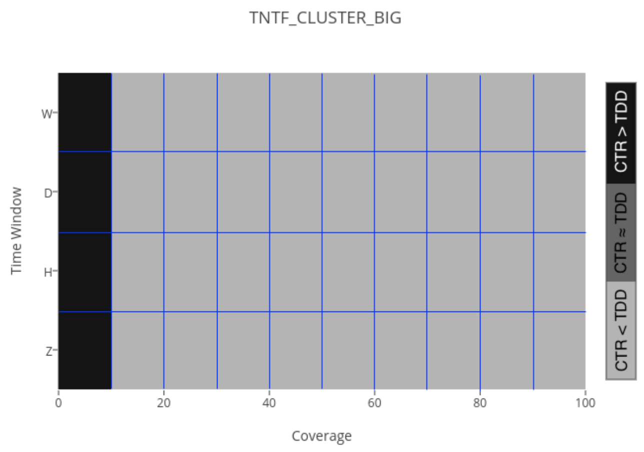

Fig. 6 Wilcoxon Tests for Total Numbers of Test Files

Table 4 Mean and Median Test File Numbers for Different Time Δs and Class Coverage Relaxations. Similarly to Table 1, each column shows a different class coverage percentage and each row corresponds to a different Time Δ where z, h, d and w map to Time Δs of 0 minutes, 1 hour, 1 day and 1 week, respectively. Further subscript "med" indicates the median number of test files while subscript "μ" is the mean. In each pair, the first value is for the control repository set and the second is for the TDD repository set.

	$\leq 10\%$	$11-20\%$	$21-30\%$	$31-40\%$	$41-50\%$
z_{med}	8.0, 3.0	3.0, 4.0	3.0, 6.0	3.0, 5.0	3.0, 4.0
z_{μ}	30.5, 8.3	6.5, 7.6	12.0, 19.6	7.4, 13.2	6.1, 11.3
h_{med}	8.0, 3.0	3.0, 4.0	3.0, 7.0	3.0, 5.0	3.0, 4.0
h_{μ}	30.6, 8.5	6.5, 7.6	12.1, 20.7	7.3, 12.9	6.1, 12.1
d_{med}	8.0, 3.0	3.0, 4.0	3.0, 6.0	3.0, 5.0	3.0, 4.0
d_{μ}	29.6, 8.2	6.3, 7.9	11.8, 20.1	8.0, 12.8	5.9, 12.1
w_{med}	6.0, 3.0	3.0, 4.0	4.0, 5.0	4.0, 5.0	3.0, 5.0
w_{μ}	25.7, 8.2	6.1, 7.6	9.2, 12.9	7.9, 13.0	5.9, 11.9

	$51-60\%$	$61-70\%$	$71-80\%$	$81-80\%$	$> 90\%$
z_{med}	7.5, 155.5	3.0, 7.0	3.0, 6.0	3.0, 6.0	2.0, 2.0
z_{μ}	70.5, 187.1	8.5, 25.9	6.7, 9.9	4.1, 9.3	2.6, 3.0
h_{med}	13.0, 44.0	3.0, 7.0	3.0, 6.0	3.0, 6.0	2.0, 2.0
h_{μ}	54.1, 140.6	8.4, 25.2	6.4, 10.1	4.0, 9.6	2.6, 3.0
d_{med}	18.0, 286.0	3.0, 7.0	3.0, 7.0	2.0, 6.0	2.0, 2.0
d_{μ}	77.4, 206.1	9.6, 24.2	6.1, 9.5	3.8, 9.0	2.6, 3.1
w_{med}	4.0, 9.0	3.0, 9.0	2.0, 5.5	3.0, 7.0	2.0, 2.0
w_{μ}	13.3, 34.6	12.5, 41.9	5.8, 8.1	4.2, 10.4	2.7, 3.3

that using a test-first approach may have no impact on measures of code quality and developer productivity (Fucci et al, 2016b,a); motivating us to use a measure that increases our confidence that a lack of statistical difference is not the result of an overly conservative FWER correction procedure. For the Holm-Bonferroni Procedure, we had $n = 400$ statistical tests at a significance of $\alpha = 0.05$, which resulted in the rejection of null hypothesis when the p-value was less than 0.03.

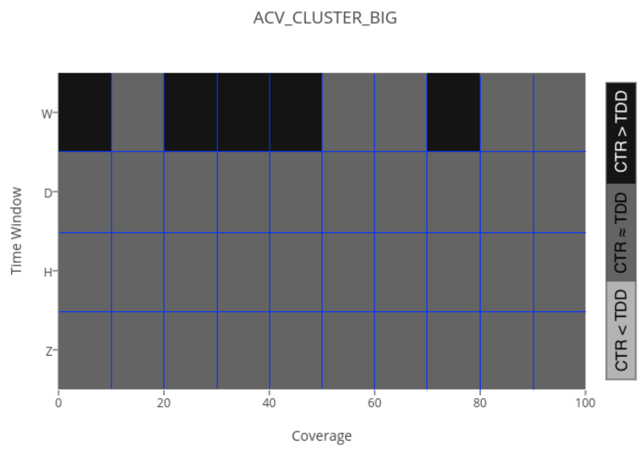

Fig. 7 Wilcoxon Tests for Average Commit Velocity

4 Findings and Analysis/Discussion

4.1 Repository Counts

Of the 256,572 Java repositories available in our GitHub data set, we found 41,302 (16.1%) repositories with test files. As reported above, Table 1 provides a comprehensive listing of the number of repositories found for each of our different TDD variants. We found only 1,991 repositories whose test files are created strictly before the associated source files (no grace period) and where class coverage is over 90%. This means that only a very small proportion (0.8%) of Java repositories in GitHub truly practice TDD.

4.2 Results for RQ1

Figure 7 shows that there are no statistical differences between the average commit velocities in the TDD repositories and the control repositories, in most cases. The only exceptions occur when we allow a time Δ of one week. Here, the controls have a higher commit velocity, which suggests that they are committing code at a slower rate than their TDD counterparts. While this does not occur for all class coverage clusters, it suggests that, in some cases, delayed testing increases the rate at which software can be developed.

4.3 Results for RQ2

Figure 8 shows that there are generally no statistically significant differences for the total number of bug-fixing commits for the majority of the TDD variants. This indicates that TDD generally seems to have no effect on the number of commits that reference bugs, and, by proxy, it also seems to indicate that TDD does not

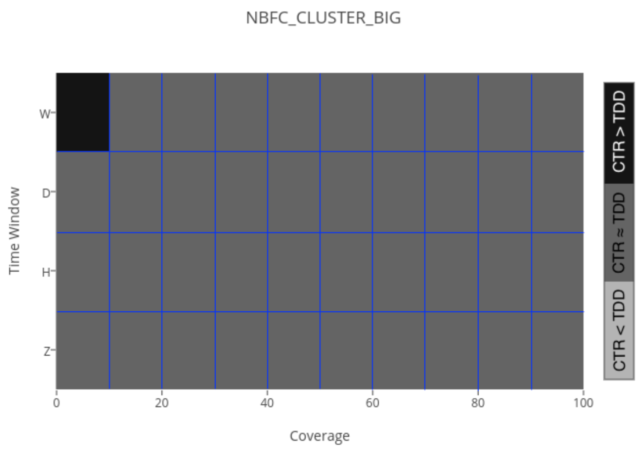

Fig. 8 Wilcoxon Tests for the Total Number of Bug Fixing Commits

impact the number of software bugs introduced during the development process. The one statistically significant difference is for the cluster with the lowest class coverage and largest time Δ. For this case it was very surprising to see that the controls had more commits referencing bug fixes, however, this is likely the result of being unable to generate an appropriate control set for this particular cluster. In particular, Figure 3 shows that the controls generated for this cluster have significantly more commits, so it is reasonable to expect that with more commits you would see more commits containing bug fixes.

4.4 Results for RQ3

Figure 9 explores the number of issues associated with the Java repositories; here again, we see no statistical differences for any of the TDD variants. This criterion is also being used as a measure of software quality and shows that TDD repositories seem to have no more issues filed against them than the general repositories represented in the control set.

4.5 Results for RQ4

As shown in Figure 10, most clusters do not show significantly increased adoption of TravisCI. While this demonstrates that practicing TDD does not generally lead to the adoption of continuous integration, we can see that clusters with extremely high class coverage do tend to use TravisCI more than general repositories. This result confirms our intuition that projects that adopt rigorous testing, also tend to use continuous integration to further ensure code quality.

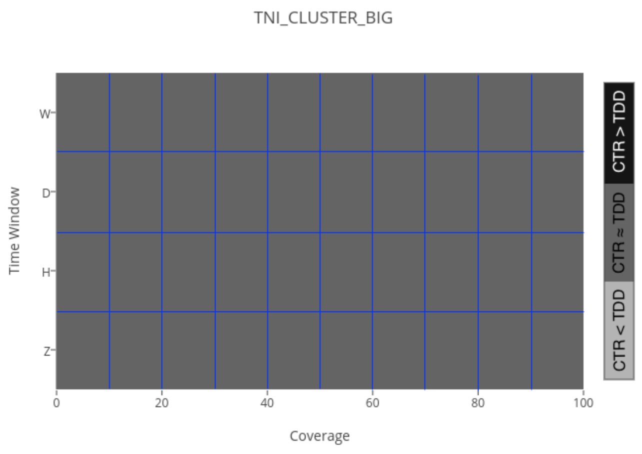

Fig. 9 Wilcoxon Tests for the Number of Issues

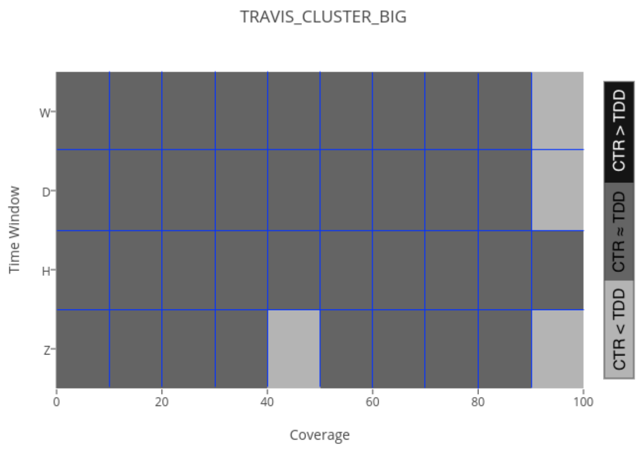

Fig. 10 Wilcoxon Tests for the Inclusion of Travis CI

4.6 Results for RQ5

Finally, Figure 11 shows that TDD does not generally lead to any significant difference in the number of pull requests made to Java repositories. While this is the apparent general trend in the data, there are three clusters that show a significant increase in the number of pull requests for TDD repositories. It is unclear why these three clusters should specifically show this result, but this constitute evidence that TDD practices may increase developer collaboration in certain situations. Perhaps Pull requests are not providing the entire story. Thus we compared the distributions of developer contribution per repository. Figure 12 depicts the results of comparing Gini coefficients between control repositories and TDD repositories.

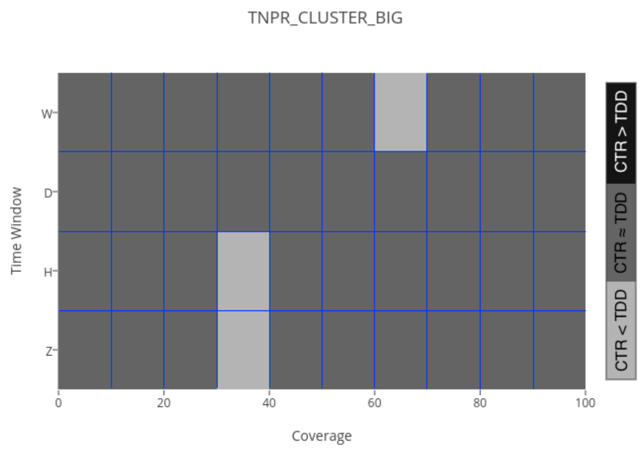

Fig. 11 Wilcoxon Tests for Numbers of Pull Requests

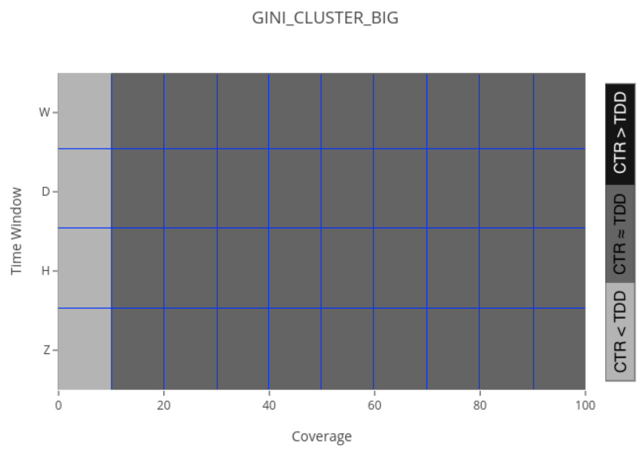

Fig. 12 Wilcoxon Tests for Gini Coefficients

The Gini coefficient is a measure of inequity in terms of proportion of contribution (Vasa et al, 2009), so we wanted to know are TDD repositories more equitable per author in terms of number of commits per author than the control repositories. The vast majority of TDD repositories are not statistically significantly different from control repositories in terms of Gini coefficients, but there are significant differences in the low (0.1) test coverage distributions where TDD repositories show more skewed or inequitable contributions than control repositories. We conclude that there is a lack of clear evidence if TDD practices are a deterrent or detriment to pull-request based collaboration and to the skew of contribution of commits per author.

5 Threats to Validity

In this work, internal validity is threatened by our choice to use file names as our basis of TDD identification. In particular, this approach does not consider file contents, which might be problematic as not all developers use the source/test file naming convention we assumed. For example, they may instead name files by their use case. Also, it may be the case that repositories truly employing TDD were omitted from our count due to low test-to-source-file ratio. Another threat to internal validity was our use of hand-crafted regular expressions for the identification of bugs fixing commits. This may have resulted in an over- or an under-estimation of the true number of bugs. A final threat to internal validity is that the data used for this study was a byproduct of software development and was not collected specifically to study TDD.

Construct validity is threatened in this work by not considering dynamic code such as Java reflections, where the behaviour of a class changes at run time[9]. Therefore we may be erroneously excluding repositories that are practicing TDD. Another threat to construct validity is that we assume repositories follow a particular TDD variant consistently. For example, this means that repositories that start with high class coverage but then drop in coverage are simply binned into the lower class coverage grouping. Our study relies on opertional data, that is artifacts and information left behind by developers while developing software. Developers did not make these records for us to study, these records and artifacts were created to make software. Because of this we do not have a clean and controlled study, we are are trying to attribute behaviour in the wilds of Github to known processes, and thus we cannot make controlled experiments, we can only rely on control subsets. Construct validity is further threatened by our choice to only consider file creation times, as this does not account for the order in which the contents of files are completed. This means we might not know if TDD is truly being practiced – *i.e.*, if all the test code in a test file is written before the source code that it tests. Finally, construct validity is threatened by our choice to measure test files by their imports of JUnit, TestNG and Android test frameworks. This may not capture all testing activities as developers may test with other frameworks or without the use of a framework. Our choice to look at import statements may also threaten construct validity because software developers could conceivably use java annotations and decorator patterns to produce test classes that do not explicitly use the Java import syntax.

External validity is threatened by our choice to only work with Java files. This was done out of convenience but means that this work may not generalize to other programming languages. Another threat to external validity was our decision to measure the occurrence of continuous integration by only looking Travis-CI and the presence of a `travis.yml` file. It is possible that repositories practicing continuous integration were ignored in this study for not using Travis-CI. A final threat to external validity was our choice to not exclude all small or personal projects from the study. While we have studied a set of repositories that are representative of GitHub, this work may not necessarily generalize to enterprise level software.

[9] `http://www.javatpoint.com/java-reflection`

6 Conclusions and Future Work

In this work we studied Java repositories on GitHub and compared those practicing Test Driven Development to those that did not. While our results are interesting, this study cannot claim that any of these results are the direct effect of implementing *TDD-like* or TDD practices as there may be confounding factors in the data.

In our study we found that 16.1% (41,302) of Java repositories on GitHub have test files and that only 0.8% (1,991) repositories strictly practiced TDD in September 2015. This corroborates evidence by Beller *et al.* (2015*a*) that TDD is not commonly practiced.

We found that that there was no statistically significant support for any of the research questions posed in this work: practicing TDD does not seem to affect commit velocity, number of bug fixing commits, numbers of issues, usage of TravisCI nor numbers of pull requests. This lack of evidence for the benefits claimed by TDD advocates suggests that TDD may not be worth the overhead when choosing a testing process.

Having studied the differences between repositories practicing TDD versus those that do not, future work needs to be done to more rigorously determine if these are the direct results of implementing a TDD methodology and to determine if any confounding factors have influenced these results. Extensions of this work will involve studying the order in which methods within classes are developed and tested, as well as investigating how source and test files correlate over time between TDD repositories and repositories using other development methodologies. Furthermore future work needs to determine the economic cost versus the empirical benefit of TDD, as currently we cannot empirically state any significant difference in the qualities that we measured.

Acknowledgements The authors wish to acknowledge the support of NSERC, and of AMII.

References

Aickin M, Gensler H (1996) Adjusting for multiple testing when reporting research results: the bonferroni vs holm methods. American journal of public health 86(5):726–728

Athanasiou D, Nugroho A, Visser J, Zaidman A (2014) Test code quality and its relation to issue handling performance. Software Engineering, IEEE Transactions on 40(11):1100–1125

Beck K (2003) Test-driven development: by example. Addison-Wesley Professional

Beck K, Andres C (2004) Extreme Programming Explained: Embrace Change (2Nd Edition). Addison-Wesley Professional

Beller M, Gousios G, Panichella A, Zaidman A (2015a) When, how, and why developers (do not) test in their ides. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, pp 179–190

Beller M, Gousios G, Zaidman A (2015b) How (much) do developers test? In: Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on, IEEE, vol 2, pp 559–562

Bird C, Rigby PC, Barr ET, Hamilton DJ, German DM, Devanbu P (2009) The promises and perils of mining git. In: Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on, IEEE, pp 1–10

Brack F (2016) Code review and test first development: How to leverage existing tools and processes to make sure an efficient test first approach is being applied. https://medium.com/@fagnerbrack/code-review-and-test-driven-development-4c19b69b5761, accessed July 2017

Budhabhatti M (2008) Test-driven development and continuous integration for mobile applications. https://msdn.microsoft.com/en-us/library/bb985498.aspx, accessed July 2017

DevIQ (2017) Test driven development explained. http://deviq.com/test-driven-development/, accessed July 2017

Dyer R (2013) Bringing ultra-large-scale software repository mining to the masses with boa. PhD thesis, Iowa State University

Dyer R, Nguyen HA, Rajan H, Nguyen TN (2013) Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In: 35th International Conference on Software Engineering, ICSE 2013, pp 422–431

Elliott E (2016) 5 common misconceptions about tdd and unit tests. https://medium.com/javascript-scene/5-common-misconceptions-about-tdd-unit-tests-863d5beb3ce9, accessed July 2017

Fucci D, Erdogmus H, Turhan B, Oivo M, Juristo N (2016a) A dissection of test-driven development: Does it really matter to test-first or to test-last? IEEE Transactions on Software Engineering

Fucci D, Scanniello G, Romano S, Shepperd M, Sigweni B, Uyaguari F, Turhan B, Juristo N, Oivo M (2016b) An external replication on the effects of test-driven development using a multi-site blind analysis approach. In: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ACM, p 3

Hindle A, Godfrey MW, Holt RC (2007) Release pattern discovery via partitioning: Methodology and case study. In: Mining Software Repositories, 2007. ICSE Workshops MSR'07. Fourth International Workshop on, IEEE, pp 19–19

Hochberg Y (1988) A sharper bonferroni procedure for multiple tests of significance. Biometrika 75(4):800–802

Holm S (1979) A simple sequentially rejective multiple test procedure. Scandinavian journal of statistics pp 65–70

Jeffries R, Melnik G (2007) Guest editors' introduction: Tdd–the art of fearless programming. IEEE Software 24(3):24–30

Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining github. In: Proceedings of the 11th working conference on mining software repositories, ACM, pp 92–101

Lakeview Labs (2017) Chicago app development process. https://lakeviewlabs.io/process/development-process.html, accessed July 2017

Pugh K (2010) Lean-Agile acceptance test-driven-development. Pearson Education

Rousseeuw PJ (1987) Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. Journal of computational and applied mathematics 20:53–65

Santos EA, Hindle A (2016) Judging a commit by its cover: Correlating commit message entropy with build status on travis-ci. In: Proceedings of the 13th International Conference on Mining Software Repositories, ACM, pp 504–507

Travis-CI (2017) Learn to code with confidence. `https://education.travis-ci.com/`, accessed July 2017

Vasa R, Lumpe M, Branch P, Nierstrasz O (2009) Comparative analysis of evolving software systems using the gini coefficient. In: Software Maintenance, 2009. ICSM 2009. IEEE International Conference on, IEEE, pp 179–188

Vasilescu B, Van Schuylenburg S, Wulms J, Serebrenik A, van den Brand MG (2015) Continuous integration in a social-coding world: Empirical evidence from github.** updated version with corrections**. arXiv preprint arXiv:151201862

Weiss K (2017) How to do test-driven development with merge control and continuous integration. `http://www.izymes.com/how-to-do-test-driven-development-with-merge-control-and-continuous-integration/`, accessed July 2017

Winter D (2016) 9 benefits of test driven development. `https://www.madetech.com/blog/9-benefits-of-test-driven-development`, accessed July 2017

Zaidman A, Van Rompaey B, Demeyer S, Van Deursen A (2008) Mining software repositories to study co-evolution of production & test code. In: Software Testing, Verification, and Validation, 2008 1st International Conference on, IEEE, pp 220–229