When, how, and why developers (do not) test in their IDEs

Beller, Moritz; Gousios, Georgios; Panichella, Annibale; Zaidman, Andy

DOI
10.1145/2786805.2786843

Publication date
2015

Published in
2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings

Citation (APA)
Beller, M., Gousios, G., Panichella, A., & Zaidman, A. (2015). When, how, and why developers (do not) test in their IDEs. In *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings* (pp. 179-190). Association for Computing Machinery (ACM). https://doi.org/10.1145/2786805.2786843

When, How, and Why Developers (Do Not) Test in Their IDEs

Moritz Beller
Delft University of Technology,
The Netherlands
m.m.beller@tudelft.nl

Georgios Gousios
Radboud University Nijmegen,
The Netherlands
g.gousios@cs.ru.nl

Annibale Panichella,
Andy Zaidman
Delft University of Technology,
a.panichella@tudelft.nl,
a.e.zaidman@tudelft.nl

ABSTRACT

The research community in Software Engineering and Software Testing in particular builds many of its contributions on a set of mutually shared expectations. Despite the fact that they form the basis of many publications as well as open-source and commercial testing applications, these common expectations and beliefs are rarely ever questioned. For example, Frederic Brooks' statement that testing takes half of the development time seems to have manifested itself within the community since he first made it in the "Mythical Man Month" in 1975. With this paper, we report on the surprising results of a large-scale field study with 416 software engineers whose development activity we closely monitored over the course of five months, resulting in over 13 years of recorded work time in their integrated development environments (IDEs). Our findings question several commonly shared assumptions and beliefs about testing and might be contributing factors to the observed bug proneness of software in practice: the majority of developers in our study does not test; developers rarely run their tests in the IDE; Test-Driven Development (TDD) is not widely practiced; and, last but not least, software developers only spend a quarter of their work time engineering tests, whereas they think they test half of their time.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Experimentation, Human Factors, Measurement, Theory, Verification

Keywords

Developer Testing, Unit Tests, Testing Effort, Field Study, Test-Driven Development (TDD)

1. INTRODUCTION

How much should we test? And when should we stop testing? Since the dawn of software testing, these questions have tormented developers and their managers alike. In 2006, twelve software companies declared them pressing issues during a survey on unit testing by Runeson [1]. Fast forward to eight years later, and the questions are still unsolved, appearing as one of the grand research challenges in empirical software engineering [2]. But before we are able to answer how much we *should* test, we must first know how much we *are* testing.

Post mortem analyses of software repositories by Pinto et al. [3] and Zaidman et al. [4] have provided us insights into how tests are created and evolved at the commit level. However, there is a surprising lack of knowledge of how developers *actually* test, as evidenced by Bertolino's call to gain a better understanding of testing practices [5]. This lack of empirical knowledge of when, how, and why developers test in their Integrated Development Environments (IDEs) stands in contrast to a large body of folklore in software engineering [2], including Brooks' statement from "The Mythical Man Month" [6] that "testing consumes half of the development time." In this paper, we start to fill our knowledge gap with resilient, empirical observations on developer testing in the real world.

In our investigation, we focus on developer tests [7], i.e. codified unit, integration, or system tests that are engineered inside the IDE by the developer. Developer testing is often complemented by work outside the IDE, such as manual testing, automated test generation, and dedicated testers, which we explicitly leave out of our investigation.

By comparing the *state of the practice* to the *state of the art* of testing in the IDE [8–10], we aim to understand the testing patterns and needs of software engineers, expressed in our five research questions:

RQ1 When and Why Do Developers Test?
RQ2 How and Why Do Developers Run Tests?
RQ3 How Do Developers React to Test Runs?
RQ4 Do Developers Follow Test-Driven Development (TDD)?
RQ5 How Much Do Developers Test?

If we study these research questions in a maximally large and varied population of software engineers, the answers to them can provide important implications for practitioners, designers of next-generation IDEs, and researchers. To this end, we have set up an open-ended, longitudinal field study [11] that has run for five months and involved 416 software engineers from industry as well as open-source projects around the world. The field study is enabled by the Eclipse plugin WatchDog, which instruments the IDE and objectively observes how developers work on and with tests.

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...
http://dx.doi.org/10.1145/2786805.2786843

Our results indicate that over half of the studied users do not practice testing; even if the projects contain tests, developers rarely execute them in the IDE; Test-Driven Development is not a widely followed practice; and, completing the overall low results on testing, developers overestimate the time they devote to testing twofold. These results counter common wisdom about developer testing, and could help explain the observed bug-proneness of real-world software systems.

2. RESEARCH DESIGN

In this section, we describe the setup of our study, the acquisition and demographics of its participants, and the statistical tests that we applied.

2.1 Study Design

To be able to make statements about the *general state of testing*, it is necessary that we study our research questions in a large-scale field study with hundreds of participants. By instrumenting the IDE with a purpose-built plugin, WatchDog, we can reach the desired number of objective observations of developer testing. Using a mixed-methods approach, we compare the results from the automated monitoring of developers to their subjective survey answers.

2.2 Study Participants

In this section, we first explain how we attracted study participants and then report on their demographics.

2.2.1 Acquisition of Participants

We reached out to potential developers to install WatchDog in their IDE by:

1) Providing a high-profile project website.[1]
2) Raffling off prizes.
3) Delivering real-value to the users of WatchDog in that it gives feedback on their development behavior.
4) Writing articles in magazines and blogs relevant to Java and Eclipse developers (Eclipse Magazin, Jaxenter, EclipsePlanet, Heise News).
5) Giving talks and presentations at developer conferences (Dutch Testing Day, EclipseCon).
6) Participating in a YouTube Java Developer series.[2]
7) Penetrating social media (Reddit, Hackernews, Twitter, Facebook).
8) Approaching software development companies.
9) Contacting developers, among them 16,058 Java developers on GitHub.
10) Putting our plugin in a well-established Eclipse marketplace.[3]
11) Launching a second marketplace which increases the visibility of scientific plugins within the Eclipse ecosystem, together with the Eclipse Code Recommenders project.[4]

We put emphasis on the testing focus of WatchDog to attract developers interested in testing.

2.2.2 Demographics of Participants

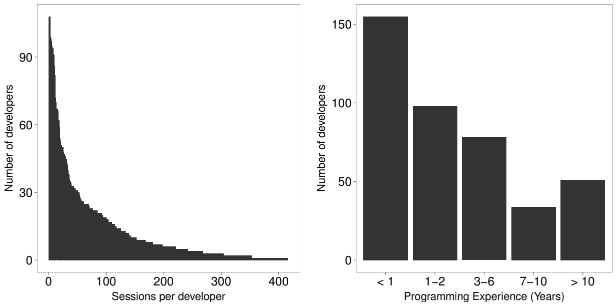


Figure 1: Distributions of the sessions per developer and their programming experience.

In total, 943 users had installed and registered WatchDog by March 1st 2015. In this paper, we report on the data we received from November 1st 2014 to March 1st 2015, excluding student data that we had analyzed separately [12]. As we updated WatchDog to fix bugs and integrate new features (see Section 2.3.2), we also filtered out data from the deprecated versions 1.0 and 1.1. In total, after filtering, we consider 416 different users and their 1,337,872 user actions (so-called intervals, see Section 2.3) from 68 different countries between November 1st 2014 and March 1st 2015.

The most frequent country of theses users is the United States (25% of users), followed by China (7%), Germany (7%), Netherlands (5%), and India (5%). The other half of users comes from 62 remaining countries, with less than 5% total share each. Our developers predominately use Windows (75% of users), 14% use MacOS and 11% Linux. Their programming experience, shown in Figure 1, is normally distributed (a Shapiro-Wilks test fails to reject the null hypothesis that it is not normally distributed at $p = 0.74$). Generally, we have slightly more in-experienced (< 3 years, 60% of users) than experienced users. On the other hand, very experienced developers ($\geqslant$ 7 years) represent more than 20% of our population.

The 416 participants registered 460 unique projects (a project cannot be shared among users). The registered projects stem from industry as well as famous open-source initiatives like the Apache Foundation, but also include private projects. In total, we observed 24,255 hours of working time in which Eclipse was open for these registered projects. Using the average work time for OECD countries of 1770 hours per year,[5] this amounts to 13.7 observed developer years in the IDE. In total, these recorded 13.7 years encompass 5,665 distinct Eclipse sessions. This large-scale approach broadens our technical study on developer testing in the IDE to a very large set of developers (compared to 4.2 years of student data in our ICSE NIER paper [12]). Furthermore, our technical development behavior data is complemented by 416 survey responses from the registration of WatchDog users and projects.

2.3 WatchDog Infrastructure

Starting with an initial prototype in 2012, we evolved WatchDog into an open-source and production-ready software solution[6] with a client-server architecture, which is designed to scale up to thousands of simultaneous users.

2.3.1 WatchDog Client

We implemented WatchDog as an Eclipse plugin, because the Eclipse Java Development Tools edition (JDT) is one of the most widely used IDEs for Java programming [13]. Thanks to its integrated JUnit support, the Eclipse JDT facilitates developer testing.

WatchDog instruments the Eclipse JDT environment and registers listeners for UI events related to programming behavior and

[1] `http://www.testroots.org`

[2] `http://youtu.be/-06ymo7dSHk`

[3] `https://marketplace.eclipse.org/content/testroots-watchdog`

[4] `http://www.codetrails.com/blog/test-analytics-testroots-watchdog`

[5] `http://stats.oecd.org/index.aspx?DataSetCode=ANHRS`

[6] `https://github.com/TestRoots/watchdog`

test executions. We group coherent events as *intervals*, which comprise a specific type, a start and an end time. This abstraction allows us to closely follow the workflow of a developer without being overwhelmed by hundreds of fine-grained UI events per minute. Every time a developer reads, modifies or executes a test or production code class, WatchDog creates a new interval and enriches it with type-specific data.

2.3.2 *WatchDog Server*

WatchDog intervals are locally cached, allowing offline work, and automatically sent to our server as a JSON stream. The WatchDog server accepts the JSON data via a REST API. After sanity checking, the intervals are stored in a NoSQL database. This infrastructure provides high extensibility up to thousands of clients and easy maintenance in case of changes in the data format in the client. Moreover, we can remotely trigger an update of all WatchDog clients, which allows us to fix bugs and extend its functionality any time. Automated ping-services monitor the health of our web API, so we can immediately react if a problem occurs. Thereby, our WatchDog service achieved an uptime of 99.6%.

2.4 Developer Survey

Having installed WatchDog, a developer first signs-up as a user with WatchDog (or reuses an already registered ID), and after that registers the current Eclipse workspace as a WatchDog project. Both registrations includes a short survey that the developer can fill out in the IDE. Key questions in the survey regard developers' programming expertise, whether and how they test their software, and if so, which testing frameworks they employ and how much time they think they spend on testing.

2.5 Statistical Evaluation

When applying statistical tests in the remainder of this paper, we regard results as significant at a 95% confidence interval ($\alpha = 0.05$), i.e. iff $p \leqslant \alpha$. All results of tests t_i in the remainder of this paper are statistically significant at this level, i.e. $\forall i : p(t_i) \leqslant \alpha$.

For each test t_i, we first perform a *Shapiro-Wilk Normality test* s_i [14]. Since all our distributions significantly deviate from a normal distribution according to Shapiro-Wilk ($\forall i : p(s_i) < 0.01 \leqslant \alpha$), we use non-parametric tests: 1) For testing whether there is a significant statistical difference between two distributions, we use the non-parametric *Wilcoxon Rank Sum test.* 2) For performing correlation analyses, we use the non-parametric *Spearman rank-order* *(ρ) correlation coefficient* [15]. Hopkins's guidelines facilitate the interpretation of ρ [16]: they assume no correlation for $0 \leqslant |\rho| < 0.3$, a weak correlation for $0.3 \leqslant |\rho| < 0.5$, a moderate correlation for $0.5 \leqslant |\rho| < 0.7$, and a strong correlation for $0.7 \leqslant |\rho| \leqslant 1$.

3. RESEARCH METHODS

In this section, we describe how the WatchDog plugin instruments the Eclipse IDE and collects data, and how we prepared the data for our research questions on testing pattern correlations and the TDD process.

3.1 IDE Instrumentation

WatchDog focuses around the concept of intervals. Table 1 gives a technical description of the different interval types. They appear in the same order as in Figure 2, which exemplifies a typical development workflow to demonstrate how WatchDog monitors IDE activity with intervals:

A developer, Mike, starts Eclipse, and WatchDog creates three intervals: `EclipseOpen`, `Perspective`, and `UserActive` (1). Thereafter, Mike executes the unit tests of the production class he needs to change, triggering the creation of a `JUnitExecution` interval, enriched with the test result "Passed" (2). Having browsed the source code of the file (3) to understand which parts need to change (a `Reading` interval is triggered), Mike then performs the necessary changes. A re-execution of the unit test shows Mike it fails after his edit (4). Mike steps through the test with the debugger (5) and fixes the error. The final re-execution of the test (6) succeeds.

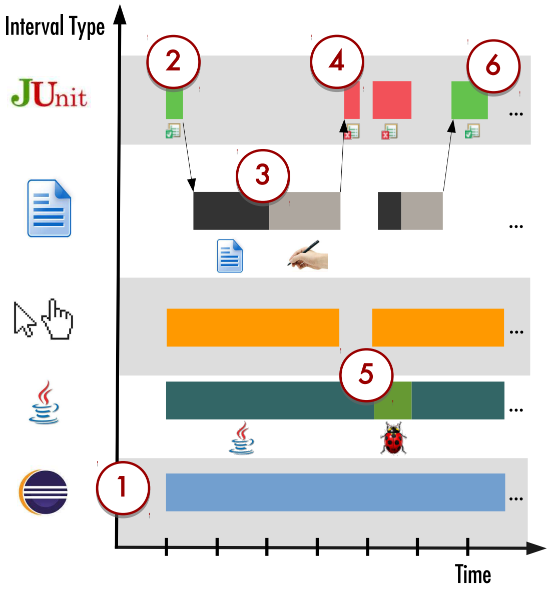


Figure 2: Exemplary workflow visualization with intervals. The interval types are described in the same order in Table 1.

Intervals concerning the user's activity (`Reading`, `Typing`, and other general activity) are backed by an inactivity timeout, so that we only record them when the user is actively working in the IDE. However, if we detect that the IDE lost the focus (end of `EclipseActive` interval), or the user switched from writing file *X* (`Typing`) to reading file *Y* (`Reading`), we immediately end the currently opened interval. Intervals may overlap. For example, `Typing` or `Reading` intervals are wrapped inside a user activity (which is again wrapped within an `EclipseActive`, `Perspective` and `EclipseOpen` interval). However `Reading` and `Typing` intervals are by nature mutually exclusive. We refer to an Eclipse session as the timespan in which Eclipse was open and not closed or interrupted, for example because the developer suspended the computer. All intervals that belong to one Eclipse session are hence wrapped within one `EclipseOpen` interval, as in Figure 2 (1).

Depending on the type of the interval, we enrich it with different numerical and categorical information: in a `Reading` or `Typing` interval, we store whether the underlying file is a Java class, a hash of its filename, its length in source lines of code without whitespaces (SLOC), and whether it is production or test code. Additionally, for `Typing` intervals, we calculate the Levensthein edit distance [17] between the content of the file before and after the modification in the interval. This gives us an indication of the size of the changes made in the `Typing` interval. If it is a Java class, we rate the file that the developer accesses in a `Reading` or `Typing` interval as either production or test code. We classify any other file type, for example an XML configuration file, as unknown.

We have four different recognition categories for test classes (see Table 1): To designate the file as a test that can be executed

Table 1: Overview of WatchDog intervals.

Interval Type	Description
JUnitExecution	Interval creation invoked through the Eclipse JDT-integrated JUnit runner (also working for Maven projects). Each test execution is enriched with the SHA-1 hash of its test name (making a link to a Reading or Typing interval possible), test result, test duration and child tests executed.
Reading	Interval in which the user was reading in the IDE. Backed by inactivity timeout. Enriched with an abstract representation of the read file, containing the SHA-1 hash of its filename, its SLOC, and an assessment whether it is production code, or test code. A test can further be categorized into a test (1) which uses JUnit and is therefore executable in the IDE; (2) which employs a testing framework; (3) which contains "test" in its filename; (4) or contains "test" in the project file path (case-insensitive). Backed by inactivity timeout.
Typing	Interval in which the user was typing in the IDE. Backed by inactivity timeout.
UserActive	Interval in which the user was actively working in the IDE (evidenced for example by keyboard or mouse events). Backed by inactivity timeout.
EclipseActive*	Interval in which Eclipse had the focus on the computer. *) Not shown in Figure 2.
Perspective	Interval describing which perspective the IDE was in (Debugging, regular Java development, ...)
EclipseOpen	Interval in which Eclipse was open. If the computer is suspended, the EclipseOpen is closed and the current sessions ends. Upon resuming, a new EclipseOpen interval is started, discarding the time in which the computer was sleeping. Each session has a random, unique identifier.

in Eclipse, we require the presence of at least one JUnit import together with at least one method that has the @Test annotation or that follows the testMethod name convention. This way, we support both JUnit3 and JUnit4. Furthermore, we recognize imports of common Java test frameworks and their annotations (Mockito, PowerMock). As a last resort, we recognize when a file contains "Test" in its file name or the project file path. It seems a common convention to pre- or postfix the names of test files with Test [4], or to place all test code in one sub-folder. For example, the standard Maven directory layout mandates that tests be placed under src/test/java.[7] Thereby, we can identify and differentiate between all tests that employ standard Java testing frameworks as test runners for their unit, integration, or system tests, test-related utility classes, and even tests that are not executable. We consider any Java class that is not a test according to this broad test recognition strategy to be production code.

3.2 Sequentialization of Intervals

For RQ3 and RQ4, we need a linearized stream of intervals following each other. We generate such a sequence by ordering the intervals according to their start time. For example, in Figure 2, this sequenced stream after the first test failure in (4) is:
Failing Test → Switch Perspective → Start JUnit Test → Read Production Code → ...

3.3 Correlation Analysis

We address our research questions RQ1 and RQ2 with the help of correlation analyses. For example, one of the steps to answer RQ1 is to correlate the amount of changed code, i.e. the code churn [18], introduced in all Typing intervals on test code, with the number of test executions. Even though strongly coupled ($\rho = 0.91$), the code change is a more precise measure of how much a developer changed a file than purely counting the number of Typing intervals on the file. For our analysis, the size of the modifications to a file is more important than in how many Typing intervals the modification happened.

Intuitively, if developers change a lot of test code, they should run their tests more often. Like all correlation analyses, we first compute the churn and the number of test executions for each Eclipse session and then calculate the correlation over these summed-up values of each session. Eclipse sessions form a natural divider between work tasks and work days, as we expect that developers typically do not close their IDE or laptop at random, but exactly when they do not need it anymore, see Table 1. Therefore, we refrained from artificially dividing our data into smaller time units. Introducing such a division would also pose a problem to a necessary sequentialization of our intervals, because it is not clear how to linearize overlapping intervals.

3.4 Recognition of Test-Driven Development

Test-Driven development (TDD) is a software development process originally proposed by Beck [19]. While a plethora of studies have been performed to quantify the supposed benefits from TDD [20, 21], it is unclear how many developers use it in practice. In RQ4, we investigate (1) how many developers follow TDD (2) to which extent. In the following, we apply Beck's definition of TDD to the WatchDog interval concept, providing the first *verifiable definition of TDD* in practice.

TDD is a cyclic process comprising a functionality-evolution phase depicted in Figure 3, optionally followed by a functionality-preserving refactoring phase depicted in Figure 4. We can best illustrate the first phase with the strict non-finite automaton (NFA, [22]) at the top of Figure 3 and our developer Mike, who is now following TDD: before Mike introduces a new feature or performs a bug fix, he assures himself that the test for the production class he needs to change passes (JO in Figure 3 stands for a JUnitExecution that contains a successful execution of the test under investigation). Thereafter, he first changes the test class (hence the name "test-first" software development) to assert the precise expected behavior of the new feature or to document the bug he is about to fix. We record such changes in a Typing interval on test code. Naturally, as Mike has not yet touched the production code, the test must fail (JE). Once work on the test is finished, Mike switches to production code (Type Prod.), in which he makes

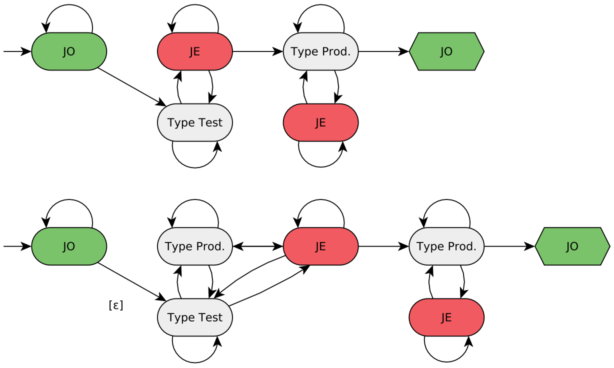


Figure 3: Strict (top) and lenient NFA of TDD.

[7] http://maven.apache.org/guides/getting-started

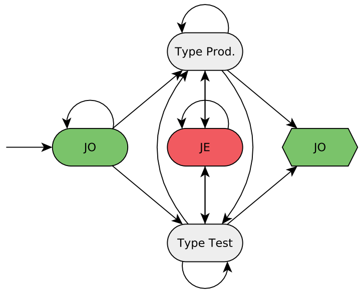
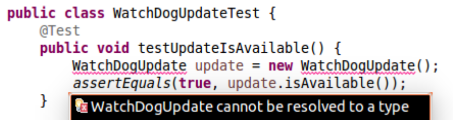


Figure 4: NFA for the refactoring phase of TDD.



```
public class WatchDogUpdateTest {
    @Test
    public void testUpdateIsAvailable() {
        WatchDogUpdate update = new WatchDogUpdate();
        assertEquals(true, update.isAvailable());
    }
```

WatchDogUpdate cannot be resolved to a type

Figure 5: Compile errors while creating a TDD test.

precisely the minimal required set of changes for his failing test to pass again (JO). The TDD cycle can begin anew.

When we tried to apply this strict TDD process, we found that it is very difficult to follow in reality, specifically the clear separation between changes to the test, and later changes to the production code. Especially when developing a new feature like WatchDog-Update in Figure 5, developers face compilation errors during the test creation phase of TDD, because the class or method they want to assert on (WatchDogUpdate) does not exist yet. To be able to have an executing, but failing test, they have to mix in the modification or creation of production code. Moreover, developers often know the result of a test without executing it (for example, if it contains compile errors), and that a test case succeeds before they start to work on it (for example, because they fixed the test on their previous day at work). To adjust for these deviations between a strict interpretation of TDD and its application, we have created the lenient non-finite automaton (ε-NFA, [22]) at the bottom of Figure 3, which is more suitable for the recognition of TDD in practice. Due to the ε-edge, a TDD cycle can directly start with modifications of test code.

TDD does not only comprise a functionality changing phase, but also the code refactor phase depicted in Figure 4. In this phase, developers have the chance to perform functionality-preserving refactorings. Once they are finished with refactoring, the tests must still pass [19]. It is impossible to separate changes between production and test classes in the refactoring phase in practice, as the latter rely on the API of the first.

To assess how strictly developers follow TDD, we convert all three NFAs to their equivalent regular expressions and match them against the linearized sequence of intervals (see Section 3.2). For a more efficient analysis, we can remove all intervals from the sequentialized stream except for JUnitExecution and Typing intervals, which we need to recognize TDD. To be able to draw a fine-grained picture of developers' TDD habits, we performed the analysis for each session individually. We count refactoring activity towards the total usage of TDD. The portion of matches in the whole string sequence gives us a precise indication of a developer's adherence to TDD.

4. RESULTS

In the following, we detail the results to each of our research questions individually per subsection.

4.1 RQ1: When and Why Do Developers Test?

To be able to answer how and why developers test, we must first assess:

RQ1.1 How Common Is Testing?

When we apply the broadest recognition of test classes as described in Section 3.1 and Table 1, 200 of the 460 analyzed projects contain tests that a user either read or modified. Hence, for 57% of projects, we could not detect work on tests in the IDE, neither to execute, nor to read or modify them.

If we narrow down the recognition of tests to JUnit tests, which can be run through Eclipse, we find that 86 projects have such tests. When we compare these projects to the projects who claimed to have JUnit tests in the survey, an intriguing discovery emerges: only for 47% of projects which claimed to have JUnit tests in the survey, could we *technically* detect them in our interval data (as either Reading, Typing, or JUnitExecution). Our second sub-research question is:

RQ1.2 How Frequently Are Tests Executed?

From the 86 projects, we observed test executions in the IDE in 73 projects (85%). The 68 developers of these 73 projects contributed 3,424 sessions and ran 10,840 test executions.

We can divide the 3,424 sessions into two groups: we find 2,897 sessions (85%) in which no test was run (but could have been run, as we know the projects contains executable tests), and only 527 sessions in which at least one test was run. Consequently, the average number of executed tests per session is relatively small (3.2) for these 73 projects. When we consider only sessions in which at least one test was run, the average number of test runs per session is 20.7.

When developers work on tests, we expect that the more they change their tests, the more they run their tests to inform themselves about the current execution status of the test they are working on. RQ1.3 and following can therefore give an indication as to why and when developers test:

RQ1.3 Do Developers Test Their Test Code Changes?

The correlation between test code churn and the number of test runs yields a moderately strong $\rho = 0.66$ in our dataset. While there is an obvious relationship between the two variables, the correlation does not imply a causation or a direction. Therefore, we cannot say that developers executed more tests *because* they changed more test code, although this is one of the likely possibilities.

A logical next step is to assess whether the same holds for modifications to production code: Do developers assert that their production code still passes the tests?

RQ1.4 Do Developers Test Their Production Code Changes?

This correlation is significant, but weaker, with $\rho = 0.38$. Finally, in how many cases do developers modify their tests, when they touch their production code (or vice versa), expressed in:

RQ1.5 Do Developers Co-Evolve Test and Production Code?

A weak $\rho = 0.35$ suggests that tests and production code have some tendency to change together, but it is certainly not the case that developers modify their tests for every production code change, and vice versa.

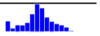
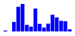
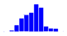
4.2 RQ2: How and Why Do Developers Run Tests?

When developers run tests in the IDE, they want to see their execution result as fast as possible. To be able to explain how and why developers execute tests, we must therefore first know how long developers have to wait before they see a test run finish:

RQ2.1 How Long Does a Test Run Take?

50% of all test executions finish within half a second, and over 75% within five seconds (see Table 2). Test durations longer than

Table 2: Descriptive statistics for important variables. Histograms are in log scale.

Variable	Unit	Min	25%	Median	Mean	75%	Max	Histogram
`JUnitExecution` duration	Sec	0	0.03	0.54	47.14	3.45	73,810	
Tests per `JUnitExecution`	Items	1	1	1	8.28	1	1,917	
Percentage of executed tests	%	0	1	1	12	12.5	100	
Time to fix failing test	Min	0	1.7	9.42	65.12	25.04	4,881	

one minute represent 7.4% of the `JUnitExecutions`. Only 4.8% of runs take more than two minutes.

Having observed that most test runs are short, our next step is to examine whether short tests facilitate testing:

RQ2.2 Do Quick Tests Lead to More Test Executions?

To answer this research question, we collect the test execution length and the number of times developers executed tests in each session, as in Section 4.1. Then, we compute the correlation between the two distributions. If our hypothesis was true, we would receive a negative correlation between the test duration and the number of test executions. This would mean that short tests are related to more frequent executions. However, the Spearman rank correlation test shows that this is not the case, as there is no correlation ($\rho = 0.26$). Combined with the fact that only a small number of tests are executed, it may suggest that developers explicitly select test cases [23]. While test selection is a complex problem on build servers, it is interesting to investigate how developers perform it locally in their IDE:

RQ2.3 Do Developers Practice Test Selection?

In JUnit, a test execution which we capture in a `JUnitExecution` interval may comprise multiple child test cases. 87% of test executions contain only one test case, while only 6.2% of test executions comprise more than 5 tests, and only 2.9% more than 50 tests (Table 2).

Test selection likely happened if the number of executed tests in one `JUnitExecution` is smaller than the total number of tests for the given project (modulo test renames, test deletion and test moves). The ratio between these two measures allows us to estimate the percentage of selected test cases. If it is significantly smaller than 100%, developers practiced test selection. Our data in Table 2 shows that developers selected only 1% of all their available tests for execution in 50% of the cases, while they ran all tests without test selection in only 3.7% of cases.

To explain how and why test selection happens, we investigate two possible scenarios: in the first, we assume that the developer picks out only one of the tests run in the previous test execution, for example to examine why the selected test failed. In the second scenario, we assume that the developer excludes a few disturbing tests from the previous test execution. In the 240 cases in which developers performed test selection, we can attribute 90% of selections to scenario 1, and 9% to scenario 2. Hence, our two scenarios are able to explain 99% of test selections in the IDE.

4.3 RQ3: How Do Developers React to Test Runs?

Having established how often programmers execute tests in their IDE in the previous research questions, it remains to assess:

RQ3.1 How Frequently Do Tests Pass and Fail?

There are three scenarios under which a JUnit execution can return an unsuccessful result: The Java compiler might detect compilation errors, an unhandled runtime exception is thrown during the test case execution, or a test assertion is not met. In either case, the test acceptance criterion is never reached, and we therefore consider them as a test failure, following JUnit's definition.

In the aggregated results of all observed 10,840 test executions, 65% (7,047) of JUnit executions fail, and only 35% pass successfully. As test failures are apparently a situation developers are often facing, we ask:

RQ3.2 How Do Developers React to a Failing Test?

For each failing test case, we generate a linearized stream of subsequently following intervals, as explained in Section 3.2. By counting and summing up developers' actions after each failing test for up to five minutes (300 seconds), we can draw a precise picture of how developers manage a failing test in Figure 6. The most immediate reaction in over 60% of the cases within the first seconds is to read production code. The second most common reaction is to read test code with 17% in the first second. However, already after five seconds, switching to another window is more common than reading test code. While switching focus away from Eclipse becomes a popular reaction five seconds after a test failure, reading production code mirrors this curve in the opposite direction, decreasing by 15% points within the first five seconds. The other curves show a more steady distribution from the beginning. Interestingly, switching to the Debug perspective or altogether quitting Eclipse almost never happens and is therefore not shown. After two minutes, the different reactions trend asymptotically towards their overall distribution, with little variability.

The logical follow-up to RQ3.2 is to ask whether developers' reactions to a failing test are in the end successful, and:

RQ3.3 How Long Does It Take to Fix a Failing Test?

To answer this question, we determine the set of unique test cases per project and their execution result. The 7,047 failing test executions were caused by 2,936 unique tests cases (according to their file name hash). For 2,051 (70%), we observed at least one successful execution. Hence, we never saw a successful execution for 30% of all tests.

For the 2,051 failing tests that we know have been fixed later, we examine know how long developers take to fix a failing test. 50% of test repairs happen within 10 minutes, and 75% within 25 minutes (Table 2).

4.4 RQ4: Do Developers Follow TDD?

In RQ4, we aim to give an answer to the adoption of TDD in practice.

Our results reveal that the sessions of only ten developers match against a strict TDD definition, the top NFA in Figure 3 (2% of all developers, or 15% of developers who executed tests, see Section 4.1). In total, only 4% of sessions with test executions contain strict TDD patterns. Only one developer uses strict TDD in more than 30% of the development process on average. The majority (68%) of the developer's intervals are devoted to the refactoring phase of TDD (depicted in Figure 4). The remaining nine developers use strict TDD in less than 8% of their intervals. Refactoring is the dominant phase in TDD, consuming on average 72% of the TDD process. All developers who practiced strict TDD have a lot of programming experience: four declared an experience between 7 and 10 years, the remaining six greater than 10 years.

Sessions from 33 developers match against the lenient TDD NFA

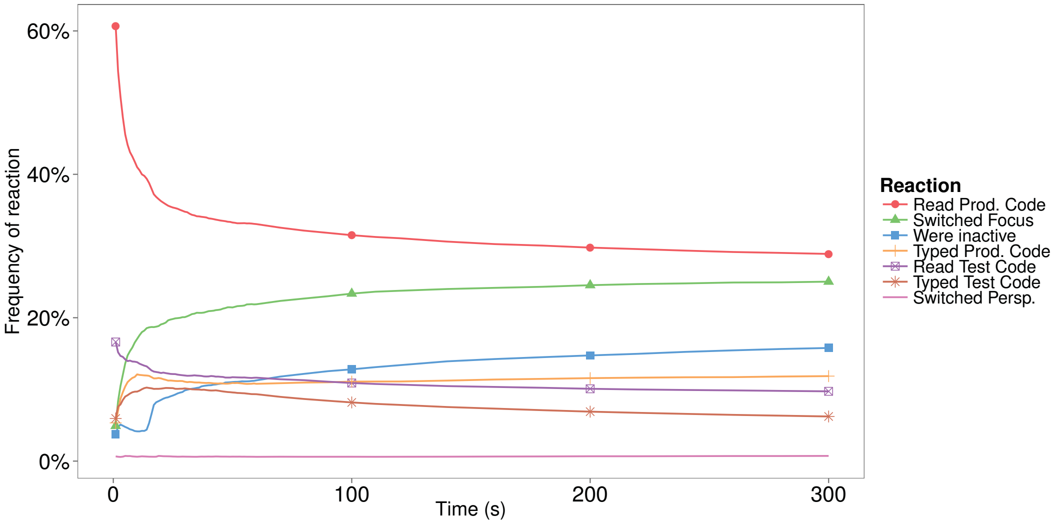


Figure 6: The immediate reactions to a failing test.

in Figure 3 (8% of all developers, or 49% of developers who executed tests, see Section 4.1). Just two developers use lenient TDD in more than 30% of their intervals, including the developer who has over 30% strict TDD matches. Six developers use lenient TDD in more than 10%, but less than 30% of their intervals. 25 of the 33 developers who use lenient TDD also refactor their code according to the TDD refactoring process in Figure 4. For them, 52% of intervals that match against the lenient TDD are due to refactoring. Of the 33 developers, seven have little programming experience (1-2 years), three have some experience (3-6 years), and the majority with 22 are very experienced (> 7 years).

Even the top TDD users do not follow TDD in most sessions. For example, the user with the highest TDD usage has one session with 69% compliance to TDD. On the other hand, in the majority of the remaining sessions, the developer did not use TDD at all (0%). We verified this to be common also for the other developers who partially used TDD. These low results on TDD are complemented by 93 projects where users *claimed* to use TDD, but in reality only 12 of the 93 *did*.

4.5 RQ5: How Much Do Developers Test?

We motivated our investigation by asking how much time developers spend on engineering tests. To answer this question, we consider `Reading` and `Typing` intervals, and further split the two intervals according to the type of the document the developer works on: either a production or test class. The duration of test executions does not contribute to it, as developers can typically work while tests execute. The short duration is negligible compared to the time spent on reading and typing, because test executions normally finish within 5 seconds (see Section 4.2). When registering new projects, developers estimated the time they spend on testing in the project. Hence, we have the possibility to verify how accurate their estimation was by comparing it to their actual testing behavior.

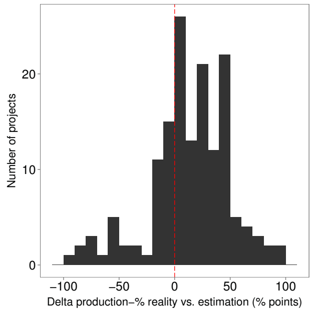


Figure 7: The delta between estimation and reality.

There are two ways to aggregate this data at different levels of granularity. The first is to explore the phenomenon on a per-project-basis: we separately sum up the time developers are engineering (i.e. reading and writing) production classes and test classes, and divide it by the sum of the two. Then, we compare this value to the developers' estimation for the project. This way, we measure how accurate each individual prediction was. The second way is to explore the phenomenon in our whole dataset, by averaging across project and *not* normalizing for the contributed development time (only multiplying each estimation with it).

Figure 7 shows a histogram of the difference between the measured production percentage and its estimation *per project*. A value of 0 means the estimation was accurate. A value of 100 denotes that the programmer expected to only work on tests, but in reality only worked on production code (-100 precisely the opposite). The median of the distribution is shifted to the right of 0. Thus, developers tend to overestimate the time they devote to testing, on average by 14% percentage points per project. For our whole dataset, we find that all developers spend in total 75% of their time writing or reading production classes, and 25% of their time on testing. However, they estimated a distribution of 52% on production code, and 48% on tests, so they overestimated the time spent on testing twice. The average time spent on production code versus test code is very similar to this overall ratio, with 77% and 23% respectively.

Moreover, reading and writing are not uniformly spread across test and production code: while developers read production classes for 67% of the total time they spend in them, they read tests much longer, namely 77% of the total time they spend in them. To verify whether this preliminary finding is statistically relevant, we use a one-tailed Wilcoxon Rank Sum test, comparing the pairwise percentage of time spent in reading test and in reading production code for each project. It confirms that, relatively, developers spend more time reading test than production code (significant at $p = 0.039$).

5. DISCUSSION

In this section, we first interpret our results and then present possible threats to validity.

5.1 Interpretation of Results

In RQ1, we established that in over half of the projects, we did not see a single opened test, even when considering a very lenient definition that likely overestimates the number of tests. While this does not mean that the projects contain no tests (a repository analysis might find that there exist a handful of test), it does indicate that testing is not an important activity for the registered WatchDog developers. Moreover, only 47% of the projects which claimed to have JUnit tests in the survey actually had intervals showing tests. For the other 53%, their developer did not execute, read, or modify a single test within five months. Since we likely overestimate tests, these two discoveries raise questions: Which value do such tests have in practice? And, further, are developers' answers true?

> The majority of projects and users do not practice testing actively.

Only 19% of all projects comprise tests that developers can run in the IDE. For 15% of projects that have such tests, developers never use the possibility to execute them. This gives a first hint that testing might not be as popular as we thought [24]. Reasons might include that there are often no preexisting tests for the developers to modify, that they are not aware of existing tests, or that testing is too time-consuming or difficult to do. The apparent lack of tests might be one factor for the bug-proneness of many current software systems.

Even for projects which have tests, developers did not execute them in most of the sessions. In contrast, the mean number of test runs for sessions with at least one test execution was high (20).

> Developers largely do not run tests in the IDE. However, when they do, they do it heftily.

One reason why some developers do not execute tests in the IDE is that the tests would render their machine unusable, for example during the execution of UI tests in the Eclipse Platform UI project. The Eclipse developers push their untested changes to the Gerrit review tool [25] and rely on it to trigger the execution of the tests on the continuous integration server. In such a scenario, the changes only become part of the "holy repository" if the tests execute successfully. Otherwise, the developer is notified via email. Despite the tool overhead and a possibly slower reaction time, our low results on test executions in the IDE suggest that developers increasingly prefer such more complex setups to manually executing their tests in the IDE. IDE creators could improve the continuous integration server support in future releases to facilitate this new workflow of developers.

Every developer is familiar with the phrase "Oops, I broke the build" [26]. The weak correlations between test churn and test executions (RQ1.3), and production churn and test executions (RQ1.4) suggest an explanation: developers simply do not assert for every change that their tests still run, because "this change cannot possibly break the tests." Even when the modifications to production or test code get larger, developers do not necessarily execute tests in the IDE more often [43]. These observations could stem from a development culture that embraces build failures and sees them as part of the normal development life-cycle, especially when the changes are not yet integrated into the main development line.

The weak correlation between production and test code churn in RQ1.5 is on the one hand expected: tests often serve as documentation and specification of how production code should work, and are therefore less prone to change. This conclusion is in line with previous findings from repository analyses [4, 27]. If, on the other hand, a practice like TDD was widely adopted (RQ4), we would expect more co-evolution of tests and production code, expressed in a higher correlation.

> Tests and production code do not co-evolve gracefully.

Another factor that could influence how often developer run tests, is how long they take to run. In RQ2, we found that testing in the IDE happens fast-paced. Most tests finish within five seconds, or less.

> Tests run in the IDE take a very short amount of time.

We could not observe a relation between the test duration and their execution frequency. The reason for this could be that there is little difference between a test which takes 0.1 seconds and one which takes 5 seconds in practice. Both give almost immediate feedback to the programmer. Hence, it seems unlikely that software engineers choose not to run tests because of their duration.

One reason for the short test duration is that developers typically do not execute all their tests in one test run. Instead, they practice test selection, and run only a small subset of their tests, mostly less than 1% of all available tests. This observed manual behavior differs strongly from an automated test execution as part of the build, which typically executes all tests.

> Developers frequently select a specific set of tests to run in the IDE. In most cases, developers execute one test.

We can explain 99% of these test selections with two scenarios: developers either want to investigate a possibly failing test case in isolation (90% of test selections), or exclude such an irritating test case from a larger set of tests (9%). This finding complements and strengthens a study by Gligoric et al., who compared manual test selection in the IDE to automated test selection in a population of 14 developers [28].

One other possible explanation for the short time it takes tests to run in the IDE is that 65% of them fail (RQ3): once a test fails, the developer might abort the execution of the remaining tests and focus on the failing test, as discovered for RQ2.3.

> Most test executions in the IDE fail.

For 30% of the failing tests, we never saw a successful execution. We built the set of tests in a project on a unique hash of their file names, which means we cannot make a connection between a failed and a successful test execution when it was renamed in-between. However, this very specific scenario is very rare, as observed at the commit-level by Pinto et al. [3]. Consequently, a substantial part of tests of up to 30% are broken and not repaired immediately. As a result, developers exclude such "broken" tests from tests executions in the IDE, as observed for RQ2.3.

Since test failures in the IDE are such a frequently recurring event, software engineers must have good strategies to manage and react to them.

The typical immediate reaction to a failing test is to dive into the offending production code.

Closing the IDE, perhaps out of frustration that the test fails, or opening the debug perspective to examine the test are very rare reactions. Five seconds after a test failure, ∼20% of programmers have already switched focus to another application on their computer. An explanation could be that they search for a solution elsewhere, for example in a documentation PDF or on the Internet. This is useful if the test failure originates from (miss-)using a language construct, the standard library, or other well-known APIs and frameworks. Researchers try to integrate answers from Internet fora like Stack Overflow into the IDE [29] to make this possibly interrupting context switch unnecessary.

TDD is one of the most widely studied software development processes [20, 21]. Even so, it is unknown how widespread its use is in practice. We have developed a formal technique that can precisely measure how strictly developers follow TDD. In all our 460 projects, we found only three users that employed TDD for more than 30% of their changes, and only one session where the majority of changes happened according to TDD. Similar to RQ1, we notice a stark contrast between survey answers and the observed behavior of developers. Only in 12% of the projects claiming to do TDD, the developers actually followed it (to a small degree).

TDD is not widely practiced. Programmers who claim to do it, neither follow it strictly nor for all their modifications.

Possible reasons for the small adoption of TDD in practice are manifold: developers might not find TDD practical or useful for most of their changes (for example, when implementing a UI), they might take shortcuts and skip the mandatory test executions in Figure 3 because they know the test cannot or must succeed, or the underlying code does simply not allow development in a test-first manner, for example because of framework restrictions. Furthermore, TDD might simply be misunderstood by developers.

The question of how much time software engineers put into testing their application was first asked (and anecdotally answered) by Brooks in 1975 [6]. Nowadays, everybody seems to know that "testing takes 50% of your time." While their estimation was remarkably on-par with Brooks' general estimation (48:52), developers tested considerably less than they thought they would (only 25% of their time), overestimating the real testing time two-fold.

Developers spend a quarter of their time engineering tests in the IDE. They overestimated this number twofold.

In comparison, students tested 9% of their time [12], and overestimated their testing effort threefold. Hence, real-world developers test more and have a better understanding of how much they test than students. Surprisingly, their perception is still far from reality. While the reasons for this "Test Effect" might be psychological (testing is usually not attributed with "fun" and a more destructive activity by nature), its consequences can have severe implications on the quality of the resulting product. Software developers should be aware of how little they test, and how much their perception deviates from the actual effort they invest in testing in the IDE. Together with RQ1 and RQ3, this observation also casts doubt on whether we can trust untriaged answers from developers in surveys, especially if the respondents are unknown to the survey authors.

Observed behavior often contradicted survey answers.

5.2 Threats to Validity

In this section, we discuss the limitations and threats that can affect the validity of our study and show how we mitigated them.

Limitation. The main limitation of our study is that we can only capture what happens inside Eclipse. Conversely, if developers perform work outside the IDE, we cannot record it. Examples for such behavior include pulling-in changes through an external revision control tool like `git` or `svn` or modifying a file loaded in the IDE with an external editor. We cannot detect work on a white board or thought processes of developers, which are generally very difficult to quantify. However, in our research questions, we are not interested in the absolute time of work processes, but in their ratio. As such, it seems reasonable to assume that work outside the IDE happens in the same ratio as in the IDE. For example, we have no indication to assume that test design requires more planning or white board time than production code.

Construct validity concerns errors caused by the way we collect data. For capturing developers' activities we use WatchDog (described in Section 2.3), which we thoroughly tested with end-to-end, integration and developer tests. Moreover, 40 students already had used it before the start of our data collection phase [12]. To verify the integrity of our infrastructure and the correctness of the analysis results, we performed end-to-end tests on Linux, Windows, and MacOS with short staged Eclipse sessions, starting from the original data collection (behavior in Eclipse) and ending at the analyzed results.

Internal validity regards threats inherent to our study.

Our population (see Section 2.2.2) shows no peculiarity, like an unusually high number of users from one IP address or from a country where the software industry is weak. Combined with the fact that we use a mild form of security (HTTP access authentication), we have no reason to believe that our data has been tampered with (for example, in order to increase the chances of winning a prize).

A relatively small set of power-users contribute the majority of development sessions. However, the distribution in Figure 1 does not follow a Power-Law distribution (the goodness-of-fit test after Clauset et al. [30] fails to reject that it is at $p = 0.09$). This does not mandate an a-priori need to further filter or sample sessions or users. Moreover, as we are exploring a phenomenon, we would run the risk of distorting it through sampling. Since WatchDog is freely available, we cannot control who installs it. Due to the way we advertise it (see Section 2.2.1), our sample might be biased towards developers who are actively interested in testing.

The Hawthorne effect poses a similar threat [31]: participants of our study would be more prone to use, run and edit tests than they would do in general, because they know (1) that they are being measured and (2) they can preview a limited part of their behavior. As discussed in Section 2.2.1, it was necessary to give users an incentive to install WatchDog. Without the preview functionality, we would likely not have had any users.

All internal threats point in the direction that our low results on testing are an overestimation of the real testing practices.

External validity threats concern the generalizability of our results. While we observed over 13 years of development worktime (collected in 1,337,872 intervals originating from 416 developers over a period of five months), the testing practices of particular individuals, organizations, or companies are naturally going to deviate from our population phenomenon observation. Our contribution can be understood as an observation of the general state of developer testing among a large corpus of developers and projects. However, we also examined if certain sub-groups deviated significantly form our general observations. As an example of this, we identified that only very experienced programmers follow TDD to some extent in Section 4.4.

Since other programming language communities have different testing cultures and use other IDEs that might not facilitate testing in the same way that the Eclipse IDE does, their results might deviate from the relatively mature and test-aware Java community.

Last but not least, the time we measure for an activity like testing in the IDE does not equal the effort an organization has to invest in it. Arguments against this are that developer testing per hour is as expensive as development (since both are done by the same persons), and that time is typically the critical resource in software development. An in-depth investigation with management data such as real project costs is necessary to validate this in practice.

Our conclusions are drawn from the precisely-defined and scoped setting of developer testing. To draw a complete picture of the state of testing, more multi-faceted research in different environments and settings is needed.

6. RELATED WORK

A number of tools have been developed to assess development activity at the sub-commit level. These tools include Syde [32], Spyware [33], CodingTracker [34], the "Change-Oriented Programming Environment,"[8] the "Eclipse Usage Data Collector,"[9] QuantifiedDev,[10] Codealike,[11] and the work by Minelli et al. [35]. However, none of these focuses on time-related developer testing.

When investigating the presence or absence of tests, Kochar et al. mined 20,000 open-source projects and found that 62% contain unit tests [36]. LaToza et al. [37] surveyed 344 software engineers, testers and architects at Microsoft, with 79% of the respondents indicating to use unit tests. Our findings indicate that only 35% of projects are concerned with testing. One factor why our figure might be smaller is that we do not simply observe the presence of some tests, but that we take into account whether they are actually being worked with.

Pham et al. [38] interviewed 97 computer science students and observed that novice developer perceive testing as a secondary task. The authors conjectured that students are not motivated to test as they have not experienced its long-term benefits. Similarly, Meyer et al. found that 47 out of 379 surveyed software engineering professionals perceive tasks such as testing as unproductive [39].

Zaidman et al. [4] and Marsavina et al. [27] studied when tests are introduced and changed. They found that test and production code typically do not gracefully co-evolve. Our findings confirm this observation on a more fine-grained level. Moreover, they found that writing test code is phased: after a longer period of production code development, developers switch to test code. Marinescu et al. [40] observed that test coverage usually remains constant because already existing tests execute part of the newly added code. Feldt [41] on the other hand notes that test cases "grow old": if test cases are not updated, they are less likely to identify failures. In contrast, Pinto et al. [3] found that test cases evolve over time. They highlight that tests are repaired when the production code evolves, but they also found that non-repair test modifications occurred nearly four times as frequently as test repairs. Deletions of tests are quite rare and if they happen, this is mainly due to refactoring the production code. A considerable portion of test modifications are for the purpose of augmenting a test suite.

The work presented in this paper differs from the aforementioned works in that the data that we use is not obtained (1) from a software repository [3, 4, 27, 36, 41] or (2) purely by means of a survey or interview [37–39, 42]. Instead, our data is automatically gathered inside the IDE, which makes it (1) more fine-grained than commit-level activities and (2) more objective than surveys.

7. CONCLUSION

Our work studies how developers test in their IDE. Our goal was to uncover the underlying habits of how developers drive software development with tests. To this end, we performed a large-scale field study using low-interference observation instruments installed within the developer's working environment to extract developer activity. We complemented and contrasted these objective observations with surveys of said developers. We found that testing (at least in the IDE) is not a popular activity, that developers do not test as much as they believe they do, and that TDD is not a popular development paradigm.

This work makes the following key contributions:

1) A low interference method and its implementation to record fine-grained activity data from within the developers' IDEs.
2) A formalized approach to detect the use of TDD.
3) A thorough statistical analysis of the activity data resulting in both qualitative and quantitative answers in developers' testing activity habits, test run frequency and time spent on testing.

In general, we find a distorting gap between expectations and beliefs about how testing is done in the IDE, and the real practice. This gap manifests itself in the following implications:

Software Engineers should be aware that they tend to overestimate their testing effort and do not follow test-driven development by the book. This might lead to a lower-than-expected quality in their software.

IDE creators could design next-generation IDEs that support developers with testing by integrating: 1) solutions from Internet fora, 2) reminders for developers to execute tests during large code changes, 3) automatic test-selection, and 4) remote testing on the build server.

Researchers can acknowledge the difference between common beliefs about software testing, and our observations from studying developer testing in the real world. Specifically, there is a discrepancy between the general attention to testing and TDD in research, and their observed popularity in practice. More abstractly, developers' survey answers did not match their behavior in practice, and student data deviated significantly from real-world observations. This may have implications on the credibility of certain research methods in software engineering and showcases the importance of triangulation with mixed-method approaches.

8. ACKNOWLEDGMENTS

We owe our biggest gratitude to the hundreds of WatchDog users. Moreover, we thank Nepomuk Seiler, Shane McIntosh, Michaela Greiler, Diana Kupfer, Marcel Bruch, Ian Bull, Sven Amann, Katrin Kehrbusch, Maaike Beliën, and the anonymous reviewers.

[8] `http://cope.eecs.oregonstate.edu`

[9] `https://eclipse.org/epp/usagedata`

[10] `http://www.quantifieddev.org`

[11] `https://codealike.com`

9. REFERENCES

[1] P. Runeson, “A survey of unit testing practices,” *IEEE Software*, vol. 23, no. 4, pp. 22–29, 2006.

[2] A. Begel and T. Zimmermann, “Analyze this! 145 questions for data scientists in software engineering,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 12–13, ACM, 2014.

[3] L. S. Pinto, S. Sinha, and A. Orso, “Understanding myths and realities of test-suite evolution,” in *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, pp. 33:1–33:11, ACM, 2012.

[4] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, “Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining,” *Empirical Software Engineering*, vol. 16, no. 3, pp. 325–364, 2011.

[5] A. Bertolino, “Software testing research: Achievements, challenges, dreams,” in *Proceedings of the International Conference on Software Engineering (ISCE), Workshop on the Future of Software Engineering (FOSE)*, pp. 85–103, 2007.

[6] F. Brooks, *The mythical man-month*. Addison-Wesley, 1975.

[7] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[8] R. L. Glass, R. Collard, A. Bertolino, J. Bach, and C. Kaner, “Software testing and industry needs,” *IEEE Software*, vol. 23, no. 4, pp. 55–57, 2006.

[9] A. Bertolino, “The (im)maturity level of software testing,” *SIGSOFT Softw. Eng. Notes*, vol. 29, pp. 1–4, Sept. 2004.

[10] J. Rooksby, M. Rouncefield, and I. Sommerville, “Testing in the wild: The social and organisational dimensions of real world practice,” *Comput. Supported Coop. Work*, vol. 18, pp. 559–580, Dec. 2009.

[11] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering: Guidelines and Examples*. Wiley, 2012.

[12] M. Beller, G. Gousios, and A. Zaidman, “How (much) do developers test?,” in *Proceedings of the 37th International Conference on Software Engineering (ICSE), NIER Track*, pp. 559–562, IEEE, 2015.

[13] P. Muntean, C. Eckert, and A. Ibing, “Context-sensitive detection of information exposure bugs with symbolic execution,” in *Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices (InnoSWDev)*, pp. 84–93, ACM, 2014.

[14] S. S. Shapiro and M. B. Wilk, “An analysis of variance test for normality (complete samples),” *Biometrika*, vol. 52, no. 3-4, pp. 591–611, 1965.

[15] J. L. Devore and N. Farnum, *Applied Statistics for Engineers and Scientists*. Duxbury, 1999.

[16] W. G. Hopkins, *A new view of statistics*. 1997. `http://newstatsi.org`, Accessed 16 March 2015.

[17] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” in *Soviet physics doklady*, vol. 10, pp. 707–710, 1966.

[18] J. C. Munson and S. G. Elbaum, “Code churn: A measure for estimating the impact of code change,” in *Proceedings of the International Conference on Software Maintenance (ICSM)*, p. 24, IEEE, 1998.

[19] K. Beck, *Test Driven Development – by Example*. Addison Wesley, 2003.

[20] H. Munir, K. Wnuk, K. Petersen, and M. Moayyed, “An experimental evaluation of test driven development vs. test-last development with industry professionals,” in *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pp. 50:1–50:10, ACM, 2014.

[21] Y. Rafique and V. B. Misic, “The effects of test-driven development on external quality and productivity: A meta-analysis,” *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 835–856, 2013.

[22] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata theory, languages, and computation*. Prentice Hall, 2007.

[23] G. Rothermel and S. Elbaum, “Putting your best tests forward,” *IEEE Software*, vol. 20, pp. 74–77, Sept 2003.

[24] A. Patterson, M. Kölling, and J. Rosenberg, “Introducing unit testing with BlueJ,” *ACM SIGCSE Bulletin*, vol. 35, pp. 11–15, June 2003.

[25] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, “Modern code reviews in open-source projects: which problems do they fix?,” in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pp. 202–211, ACM, 2014.

[26] E. Derby, D. Larsen, and K. Schwaber, *Agile retrospectives: Making good teams great*. Pragmatic Bookshelf, 2006.

[27] C. Marsavina, D. Romano, and A. Zaidman, “Studying fine-grained co-evolution patterns of production and test code,” in *Proceedings International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 195–204, IEEE, 2014.

[28] M. Gligoric, S. Negara, O. Legunsen, and D. Marinov, “An empirical evaluation and comparison of manual and automated test selection,” in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 361–372, ACM, 2014.

[29] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, “Mining stackoverflow to turn the ide into a self-confident programming prompter,” in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pp. 102–111, ACM, 2014.

[30] A. Clauset, C. R. Shalizi, and M. E. Newman, “Power-law distributions in empirical data,” *SIAM review*, vol. 51, no. 4, pp. 661–703, 2009.

[31] J. G. Adair, “The Hawthorne effect: A reconsideration of the methodological artifact.,” *Journal of applied psychology*, vol. 69, no. 2, pp. 334–345, 1984.

[32] L. Hattori and M. Lanza, “Syde: a tool for collaborative software development,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 235–238, ACM, 2010.

[33] R. Robbes and M. Lanza, “Spyware: a change-aware development toolset,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 847–850, ACM, 2008.

[34] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, “A comparative study of manual and automated refactorings,” in *Proceedings of the 27th European Conference on Object-Oriented Programming*, 2013.

[35] R. Minelli, A. Mocci, M. Lanza, and L. Baracchi, “Visualizing developer interactions,” in *Proceedings of the Working Conference on Software Visualization (VISSOFT)*, pp. 147–156, IEEE, 2014.

[36] P. Kochhar, T. Bissyande, D. Lo, and L. Jiang, “An empirical study of adoption of software testing in open source projects,” in *Proceedings of the International Conference on Quality Software (QSIC)*, pp. 103–112, IEEE, 2013.

[37] T. D. LaToza, G. Venolia, and R. DeLine, “Maintaining mental models: a study of developer work habits,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 492–501, ACM, 2006.

[38] R. Pham, S. Kiesling, O. Liskin, L. Singer, and K. Schneider, “Enablers, inhibitors, and perceptions of testing in novice software teams,” in *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pp. 30–40, ACM, 2014.

[39] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann, “Software developers’ perceptions of productivity,” in *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pp. 19–29, ACM, 2014.

[40] P. D. Marinescu, P. Hosek, and C. Cadar, “Covrig: a framework for the analysis of code, test, and coverage evolution in real software,” in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pp. 93–104, ACM, 2014.

[41] R. Feldt, “Do system test cases grow old?,” in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pp. 343–352, IEEE, 2014.

[42] M. Greiler, A. van Deursen, and M. Storey, “Test confessions: a study of testing practices for plug-in systems,” in *Software Engineering (ICSE), 2012 34th International Conference on*, pp. 244–254, IEEE, 2012.

[43] V. Hurdugaci and A. Zaidman, “Aiding software developers to maintain developer tests,” in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 11–20, IEEE, 2012.