Developer Testing in the IDE: Patterns, Beliefs, and Behavior

Moritz Beller, *Member, IEEE*, Georgios Gousios, *Member, IEEE*, Annibale Panichella, *Member, IEEE*, Sebastian Proksch, *Member, IEEE*, Sven Amann, *Member, IEEE*, and Andy Zaidman, *Member, IEEE*

Abstract—Software testing is one of the key activities to achieve software quality in practice. Despite its importance, however, we have a remarkable lack of knowledge on how developers test in real-world projects. In this paper, we report on a large-scale field study with 2,443 software engineers whose development activities we closely monitored over 2.5 years in four integrated development environments (IDEs). Our findings, which largely generalized across the studied IDEs and programming languages Java and C#, question several commonly shared assumptions and beliefs about developer testing: half of the developers in our study do not test; developers rarely run their tests in the IDE; most programming sessions end without any test execution; only once they start testing, do they do it extensively; a quarter of test cases is responsible for three quarters of all test failures; 12 percent of tests show flaky behavior; Test-Driven Development (TDD) is not widely practiced; and software developers only spend a quarter of their time engineering tests, whereas they think they test half of their time. We summarize these practices of loosely guiding one's development efforts with the help of testing in an initial summary on Test-Guided Development (TGD), a behavior we argue to be closer to the development reality of most developers than TDD.

Index Terms—Developer testing, unit tests, testing effort, field study, test-driven development (TDD), JUnit, TestRoots WatchDog, KaVE FeedBag++

1 Introduction

How much should we test? And when should we stop testing? Since the beginning 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 eleven years later, and the questions are still open, appearing as one of the grand research challenges in empirical software engineering [2]. But before we are able to answer how we *should* test, we must first know how we *are* testing.

Post mortem analyses of software repositories by Pinto et al. [3] and Zaidman et al. [4] have provided us with insights into how developers create and evolve tests at the commit level. However, there is a surprising lack of knowledge of how developers *actually* test, as evidenced by Bertolino's and Mäntylä's calls to gain a better understanding of testing practices [5], [6]. 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" [7] that "testing consumes half of the development time."

To replace folklore by real-world observations, we studied the testing practices of 416 software developers [8] and 40 computer science students [9] with our purpose-built IDE plugin WatchDog. While these studies started to shed light on how developers test, they had a number of limitations toward their generalizability: First, they were based on data from only one IDE, Eclipse, and one programming language, Java. It was unclear how the findings would generalize to other programming environments and languages. Second, the data collection period of these studies stretched only a period of five months. This might not capture a complete real-world "development cycle," in which long phases of implementation-heavy work follow phases of test-heavy development [4], [10]. Third, we did not know how strongly the incentives we gave developers to install WatchDog influenced their behavior. Fourth, we had no externally collected data set to validate our observations against.

In this extension of our original WatchDog paper [8], built on top of our initial draft of the WatchDog idea [9] and its technical tool description [11], we address these limitations by analyzing data from four IDEs, namely Eclipse (EC), IntelliJ (IJ), Android Studio (AS), and Visual Studio (VS), and two programming languages, Java and C#. We extended our study from 416 developers to an open-ended field study [12] with 2,433 developers that stretches

- *M. Beller, G. Gousios, and A. Zaidman are with the Software Engineering Research Group, Delft University of Technology, Delft 2628, CD, The Netherlands. E-mail: {m.m.beller, g.gousios, a.e.zaidman}@tudelft.nl.*
- *A. Panichella is with the Interdisciplinary Centre for Security, Reliability and Trust Verification and Validation, University of Luxembourg, Esch-sur-Alzette 4365, Luxembourg. E-mail: annibale.panichella@uni.lu.*
- *S. Amann and S. Proksch are with the Technische Universität Darmstadt, Darmstadt 64289, Germany. E-mail: {proksch, amann}@st.informatik.tu-darmstadt.de.*

Manuscript received 12 June 2017; revised 25 Oct. 2017; accepted 16 Nov. 2017. Date of publication 21 Nov. 2017; date of current version 21 Mar. 2019. (Corresponding author: Moritz Beller.)
Recommended for acceptance by R. Holmes.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2017.2776152

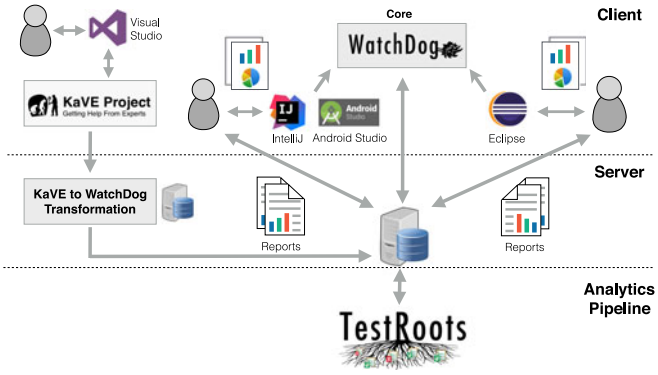

Fig. 1. WatchDog's three-layer architecture.

over a data collection period of 2.5 years. By measuring how developers use the behavior reports WatchDog provides as an incentive, we can now estimate their impact on developers' behavior. Thanks to Visual Studio data from the FeedBaG++ plugin, developed independently in the KaVE project [13], we can compare our findings against an externally collected data set.

In our investigation, we focus on developer tests [14], i.e., codified unit, integration, or system tests that are engineered inside the IDE by the developer. Developer testing in the IDE is often complemented by work outside the IDE, such as testing on the CI server [15], executing tests on the command line, 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 [16], [17], [18], we aim to understand the testing patterns and needs of software engineers, expressed in our five research questions:

RQ1 Which Testing Patterns Are Common In The IDE?
RQ2 What Characterizes The Tests Developers Run In The IDE?
RQ3 How Do Developers Manage Failing Tests In The IDE?
RQ4 Do Developers Follow Test-Driven Development (TDD) In The IDE?
RQ5 How Much Do Developers Test In The IDE?

If we study these research questions in a 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 field study [12] that has run for 2.5 years and involved 2,443 programmers from industry and open-source projects around the world. The field study is enabled by the Eclipse and IntelliJ plugin WatchDog and the Visual Studio plugin FeedBaG++, which instrument the IDE and objectively observe how developers work on and with tests.

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; only a quarter of test cases is responsible for three quarters of all test failures; 12 percent of test cases show flaky behavior; 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 almost twofold. These results counter common beliefs about developer testing and could help explain the observed bug-proneness of real-world software systems.

2 Study Infrastructure Design

In this section, we give a high level overview of our field study infrastructure design, explore how a practitioner uses WatchDog to convey an intuitive understanding of the plugin, and describe how our plugins instrument the IDE.

2.1 Field Study Infrastructure

Starting with an initial prototype in 2012, we evolved our IDE instrumentation infrastructure around WatchDog into an open-source, multi-IDE, and production-ready software solution [19]. As of version 1.5 released in June 2016, it features the three-layer architecture depicted in Fig. 1 with a client, server, and data analysis layer, designed to scale up to thousands of simultaneous users. In the remainder of this section, we first describe the client layer containing the four different IDE plugins for Visual Studio, IntelliJ, Android Studio and Eclipse (from left to right). We then describe WatchDog's server and central database and how we converted the KaVE project's FeedBaG++ data to WatchDog's native interval format. We conclude this high-level overview of our technical study design with a short description of our analysis pipeline. In earlier work, we have already given a more technical description of WatchDog's architecture and the lessons we learned while implementing it [11].

2.1.1 IDE Clients

We used two distinct clients to collect data from four IDEs: the WatchDog plugin gathers Java testing data from Eclipse and IntelliJ-based IDEs and the general-purpose interaction tracker FeedBaG++ gathers C# testing data from Visual Studio.

WatchDog *Clients for Eclipse and IntelliJ.* We originally 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 [20]. With WatchDog 1.5, we extended it to support IntelliJ and IntelliJ-based development platforms, such as Android Studio, "the official IDE for Android" [21]. Thanks to their integrated JUnit support, these IDEs facilitate developer testing.

WatchDog instruments the Eclipse JDT and IntelliJ environments and registers listeners for user interface (UI) events related to programming behavior and test executions. Already on the client side, 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 JUnit test or production code class, WatchDog creates a new interval and enriches it with type-specific data.

FeedBaG++ *for Visual Studio.* FeedBaG++ is a general-purpose interaction tracker developed at TU Darmstadt. It is available for Visual Studio, as an extension to the widely used ReSharper plugin [22], which provides static analyses and refactoring tools to C# developers.

FeedBaG++ registers listeners for various IDE events from Visual Studio and the ReSharper extension, effectively capturing a superset of the WatchDog listeners. The captured information relevant for this paper includes how developers navigate and edit source files and how they use the test runner provided by ReSharper. The test recognition

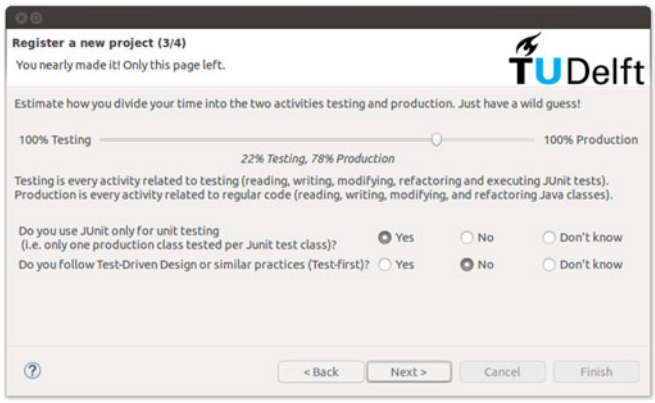

Fig. 2. Exemplary wizard page of WatchDog's project survey.

covers common .NET testing frameworks, such as NUnit or MSUnit. In contrast to WatchDog, which already groups events into intervals on the client side, FeedBaG++ uploads the raw event stream.

2.1.2 WatchDog Server

The WatchDog IDE plugins cache intervals locally, to allow offline work, and automatically send them to our server as a JSON stream. The WatchDog server accepts this JSON data via its REST API. After sanity checking, the intervals are stored in a NoSQL database. This infrastructure scales up to thousands of clients and makes changes in the clients' data format easy to maintain. Moreover, we can remotely trigger an update of all WatchDog clients, which allows us to fix bugs and extend its functionality after deployment. Automated ping-services monitor the health of our web API, so we can immediately react if an outage occurs. Thereby, our WatchDog server achieved an average uptime of 98 percent during the 2.5 years of field study.

2.1.3 WatchDog Analysis Pipeline

The WatchDog pipeline is a software analytics engine written in R comprising over 3,000 source lines of code without whitespaces (SLOC). We use it to answer our research questions and to generate daily reports for the WatchDog users. The pipeline reads in WatchDog's users, projects, and intervals

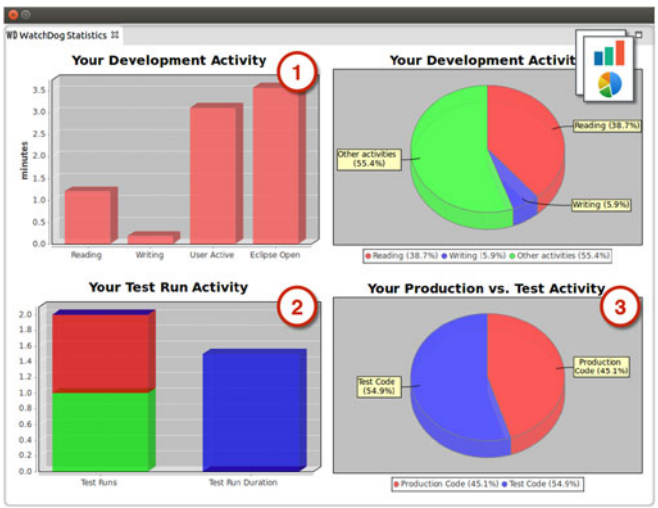

Fig. 3. WatchDog's immediate statistics view in the IDE (Source: [11]).

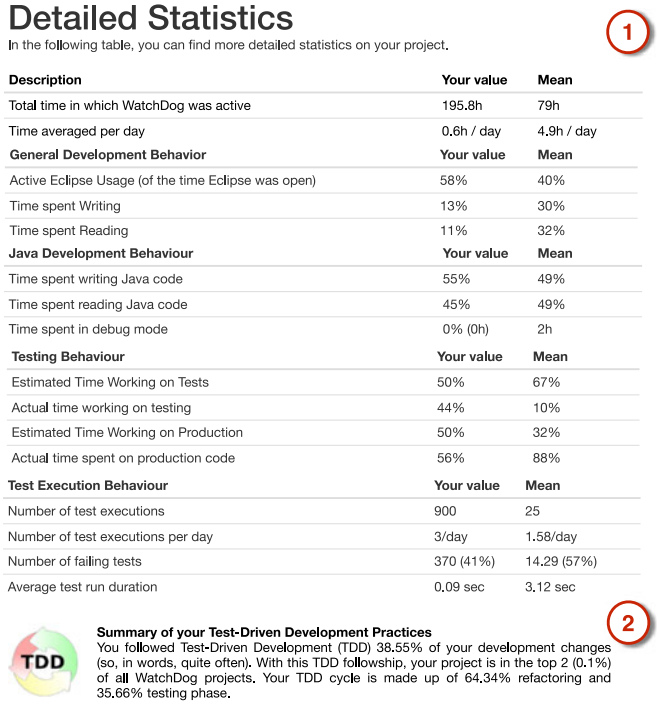

Fig. 4. WatchDog's project report (Source: [11]).

from the NoSQL database and converts them into intermediate formats fit for answering our research questions.

2.2 WatchDog Developer Survey & Testing Analytics

To give an understanding of the study context and incentives that WatchDog offers, we explore it from a practitioner's perspective in this section. Wendy is an open-source developer who wants to monitor how much she is testing during her daily development activities inside her IDE. Since Wendy uses IntelliJ, she installs the WatchDog plug-in from the IntelliJ plug-in repository.

Registration. Once installed, a wizard guides Wendy through the WatchDog registration process: First, she registers herself as a user, then the project for which WatchDog should collect development and testing statistics, and finally, she fills in an interactive voluntary in-IDE survey about testing. Fig. 2 shows one of the up to five pages of the survey. Key questions regard developers' programming expertise, whether and how they test their software, which testing frameworks they employ and how much time they think they spend on testing. Since FeedBaG++ does not collect comparable survey data, we exclude it from research questions relying on it. Wendy, however, continues to work on her project using IntelliJ, as usual, while WatchDog silently records her testing behavior in the background.

Developer Statistics. After a short development task, Wendy wants to know how much of her effort she devoted to testing and whether she followed TDD. She can retrieve two types of analytics: the *immediate statistics* inside the IDE shown in Fig. 3 and her personal *project report* on our website shown in Fig. 4. Wendy opens the *immediate statistics* view. WatchDog automatically analyzes the recorded data and generates the view in Fig. 3, which provides information about production and test code activities within a selected time window. Sub-graph ① in Fig. 3 shows Wendy that she spent more time (over one minute) reading than writing

TABLE 1
Overview of WatchDog Intervals and How We Transformed FeedBaG++ Events to Them. Related Intervals Appear without Horizontal Separation

Interval Type	WatchDog Description	FeedBaG++ Transformation
`JUnitExecution` †	Interval creation invoked through the Eclipse JDT-integrated JUnit runner, which also works for Maven projects (example in Fig. 6). 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.	FeedBaG++ tracks the ReSharper runner for the execution of NUnit tests. The results of running tests are easy to match to JUnit's result states. However, NUnit does not differentiate between *errored* and *failed* tests, so we map all failing runs to the latter and only report errors for *inconclusive* test runs.
`Reading`	Interval in which the user was reading in the IDE-integrated file editor. Enriched with an abstract representation of the read file, containing the SHA-1 hash of its filename, its SLOC, and whether it is production 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, or (4) which contains "test" in the project file path (case-insensitive). Backed by inactivity timeout.	FeedBaG++ tracks document and window events, allowing us to identify when a developer opens a specific file or brings it back to focus. If no other activity interrupts this, we count it as reading, until the inactivity threshold is reached.
`Typing`	Interval in which the user was typing in the IDE. Enriched with the Levenshtein edit distance, backed by inactivity timeout.	We use FeedBaG++'s edit events to distinguish `Reading` from `Typing` intervals and approximate the Levenshtein distance via the number of `Typing` intervals.
`UserActive`	Interval in which the user was actively working in the IDE (evidenced for example by keyboard or mouse events). Backed by inactivity timeout.	Each user-triggered event extends the current interval (or creates a new one, if there is none). Once the inactivity threshold is reached or the event stream ends, we close the current interval.
`EclipseActive` † *	Interval in which the IDE had the focus on the computer.	FeedBaG++ monitors the active window in the same way as WatchDog does. We group events into intervals.
`Perspective`	Interval describing which perspective the IDE was in (Debugging, regular Java development, ...).	We approximate the manually changed Eclipse `Perspectives`, with Visual Studio's automatically changing perspectives.
`WatchDogView` *	Interval that is created when the user consults the immediate WatchDog statistics. Only available in the Eclipse IDE.	*Not provided in* FeedBaG++.
`EclipseOpen` †	Interval in which the IDE 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.	FeedBaG++ generates specific events that describe the IDE state. From the start-up and shutdown events of the IDE, we generate `EclipseOpen`intervals.

† *As of WatchDog 1.5, we support multiple IDEs, so better interval names would have been* `TestExecution`, `IDEActive`, *and* `IDEOpen`.
**Not shown in Fig. 5.*

(only a few seconds). Moreover, of the two tests she executed ② , one was successful and one failed. Their average execution runtime was 1.5 seconds. Finally, Wendy observes that the majority (55 percent) of her development time has been devoted to engineering tests ③ , not unusual for TDD [8].

While the *immediate statistics* view provides Wendy with an overview of recent activities inside the IDE, the *project report* gives her a more holistic view of her development behavior, including more computationally expensive analyses over the whole project history. She accesses her report through a link from the IDE or directly via the TestRoots website,[1] providing the project's ID. Wendy's online project report summarizes her development behavior in the IDE over the whole recorded project lifetime. Reading the report in Fig. 4, Wendy observes that she spent over 195 hours in total on the project under analysis, an average of 36 minutes per day ①. She worked actively with IntelliJ in 58 percent of the time that the IDE was actually open. The time spent on writing Java code corresponds to 55 percent of the total time, while she spent the remaining 45 percent reading Java code. When registering the project, Wendy estimated the working time she would spend on testing to equal 50 percent. With the help of report, she finds out that her initial estimation was relatively precise, since she actually spent 44 percent of her time working on test code.

The *project report* also provides Wendy with *TDD statistics* for the project under analysis, ② in Fig. 4. Moreover, anonymized and averaged statistics from the large WatchDog user base allow Wendy to put her own development practices into perspective. This way, project reports foster comparison and learning among developers. Wendy finds that, for her small change, she was well above average regarding TDD use: She learned how to develop TDD-style from the "Let's Developer" YouTube channel.[2] The WatchDog project for "Let's Developer" is the second highest TDD follower of all WatchDog users on 5th June, 2017 (following TDD for 37 percent of all modifications).[3]

2.3 IDE Instrumentation

Here, we explain how WatchDog clients instrument the IDE. We then continue with a description of how we transform FeedBaG++ events into WatchDog intervals.

2.3.1 WatchDog Clients

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 rows in Fig. 5, which exemplifies a typical development workflow to demonstrate how WatchDog monitors IDE activity with intervals.

1. http://testroots.org/report.html

2. http://www.letsdeveloper.com
3. Project report: http://goo.gl/k9KzYj

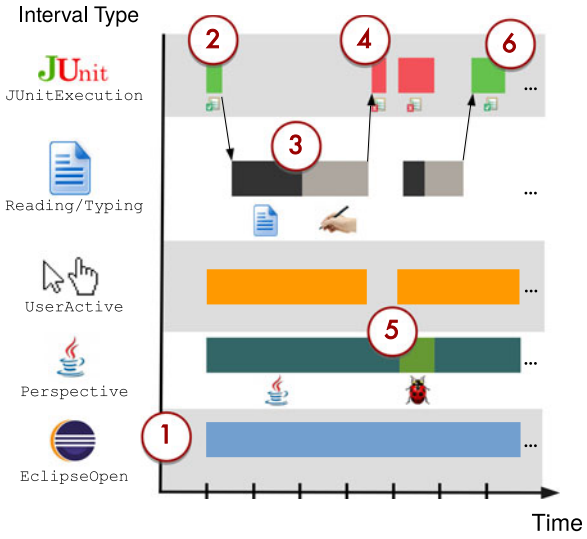

Fig. 5. Exemplary workflow visualization with intervals. Table 1 describes the interval types in the same order as they appear in the different rows.

Exemplary Development Workflow. Our developer Wendy starts her IDE. The integrated WatchDog plugin creates three intervals: `EclipseOpen`, `Perspective`, and `UserActive` ①. Thereafter, Wendy executes the unit tests of the production class she 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), Wendy performs the necessary changes. A re-execution of the unit test shows Wendy that there is a failing test after her edit ④. Wendy steps through the test with the debugger ⑤ and fixes the error. The final re-execution of the test ⑥ succeeds.

Interval Concept. WatchDog starts or prolongs intervals concerning the user's activity (`Reading`, `Typing`, and other general activity) once it detects an interval-type preserving action. For example, if there is a `Reading` interval on class X started for 5 seconds and the plugin receives a scroll event, the interval is prolonged. However, if we detect that the IDE lost focus (end of `EclipseActive` interval), or the user switched from reading file X (`Reading`) to typing in file Y (`Writing`), we immediately end the currently opened interval. WatchDog closes all such activity-based intervals after an inactivity timeout of 16 seconds, so that we adjust for breaks and interruptions. A timeout length of roughly 15 seconds is standard in IDE-based observational plugins [9], [23], [24]. Most interval types may overlap. For example, WatchDog always wraps `Typing` or `Reading` intervals inside a `UserActive` interval (which it, in turn, wraps in an `EclipseActive`, `Perspective`, and `EclipseOpen` interval). However, `Reading` and `Typing` intervals are by nature mutually exclusive. We refer to an IDE session as the time span in which the IDE was continuously running (even in the background) and not closed or interrupted, for example, because the developer suspended the computer. All intervals that belong to one IDE session are hence wrapped within one `EclipseOpen` interval, ① in Fig. 5.

We enrich `Reading` and `Typing` intervals with different information about the underlying file. To all intervals we add a hash of the filename and its file type, such as XML or Java class. For Java classes, we add their SLOC and classify them as production or test code. As our churn measure for the size of a change, we also add the Levenshtein edit distance [25] between the content of the file before and after the modification during the interval to `Typing` intervals.

Test Recognition. WatchDog has four different recognition categories for test classes (see Table 1): To designate the file as a test that can be executed in the IDE, 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` naming 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 is a common convention to pre- or postfix the names of test files with `Test` [4], or to place all test code in one subfolder. For example, the standard Maven directory layout mandates that tests be placed under `src/test/java` [26]. 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 in the IDE. We consider any Java class that is not a test according to this broad test recognition strategy to be production code.

2.3.2 FeedBaG++-to-WatchDog Interval Transformation

In contrast to the native WatchDog clients, FeedBaG++ provides us with a raw event stream (see Section 2.1.1). To feed FeedBaG++ data into the WatchDog pipeline, we derive intervals via a post factum analysis of FeedBaG++ data. In addition to this technical difference, several minor semantic differences exist in the instrumented IDEs. We had to find congruent concepts for them and transform FeedBaG++ events to intervals.

Concept Mapping. The Eclipse, IntelliJ, and the Visual Studio IDEs are similar conceptually, yet differ in some implementation details important to our study. In addition to IDE concepts, we had to map C# concepts to their Java counterparts.

One such central difference is the different testing frameworks available in the C# ecosystem. FeedBaG++ recognizes the same four categories of test classes described in Section 2.3.1: To designate a file as a test that can be executed in Visual Studio, we require an import of one of the .NET testing frameworks NUnit, XUnit, MSUnit, csUnit, MbUnit, or PetaTest. Furthermore, we recognize imports of the C# mocking frameworks moq, Rhino.Mocks, NSubstitute, and Simple.Mocking.

A difference between Visual Studio and Eclipse is that the former does not have perspectives that developers can manually open, but instead it automatically switches between its *design view* for writing code, and its *debug view* for debugging a program run. We map the concept of these Visual Studio views to the `Perspective` intervals in WatchDog.

Arguably the largest difference between IDEs is how they manage different projects and repositories. Eclipse organizes source code in a *workspace* that may contain many potentially unrelated *projects*. IntelliJ groups several *modules* in a *project*. Visual Studio organizes code in a *solution*, which contains

a number of usually cohesive *projects*. In Java, a single project or module often contains both the production code and test code. This is not the case in Visual Studio, where the two kinds of source code are typically split into two separate projects. If not accounted for, this leads to a higher number of observed projects in Visual Studio and distorts the answers to some of our project-level research questions. To counter this problem, we need functions to map test code from one project to its corresponding production code in another. The notion of a Visual Studio *solution* and even more so, IntelliJ's *project* matches the definition of a *Watchdog project*, understood as a cohesive *software development effort*. To avoid confusion about the overloaded "project" term, we asked the user explicitly whether "all Eclipse projects in this workspace belong to one 'larger' project?" in the WatchDog registration dialogues (see Section 2.2).

FeedBaG++ does not measure the Levenshtein distance in `Typing` intervals. However, WatchDog data shows that the edit distance generally correlates strongly with the number of edits: The number of production code edits correlates at $\rho = 0.88$ with production code churn, i.e., the amount of changed code [27], and the number of test edits is correlated at $\rho = 0.86$ with test code churn. Hence, we use the number of edits as a proxy for the missing churn in FeedBaG++ data.

Event Transformation. As a second step, we transformed the event stream to intervals. We re-implemented transformation rules that work on the raw FeedBaG++ event stream based on the interval detection logic that the WatchDog plugin family performs within the IDE. We then store it in WatchDog's central NoSQL database store (see Fig. 1). In the right column of Table 1, we sketch how we derive the various WatchDog interval types from the events that FeedBaG++ captures. From there, we simply re-use the existing WatchDog analysis pipeline.

3 Research Methods

In this section, we describe the methods with which we analyze the data for our research questions.

3.1 Correlation Analyses (RQ1, RQ2)

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 test code churn introduced in all `Typing` intervals with the number of test executions.

Intuitively, we have the assumption that if developers change a lot of code, they would run their tests more often. Like all correlation analyses, we first compute the churn and the number of test executions for each IDE session and then calculate the correlation over these summed-up values of each session. IDE sessions form a natural divider between work tasks, as we expect that developers typically do not close their IDE or laptop at random, but exactly when they have finished a certain task or work step (see Table 1).

3.2 Analysis of Induced Test Failures (RQ3)

We abstract and aggregate the tests of multiple projects to derive general statements like "only 25 percent of tests are responsible for 75 percent of test failures in the IDE." Algorithm 1 outlines the steps we use to count the number of executed test cases and the number of corresponding test failures they have caused per project. We iterate over all failed test cases (line 9), determine which percentage of failed test executions they are responsible for (line 10) and put the resulting list of test cases in descending order, starting with the test case with the highest responsibility of test failures (line 12). We then normalize the absolute count numbers to the relative amount of test cases in the project (line 14) by calling calcFailingTestPercentage on every project, average the results so that each project has the same weight in the graph, and plot them.

The algorithm makes assumptions that lead to a likely underestimation of the percentage of test failures caused by a specific test: First, it assumes that test names are stable. If test names change during our field study, they count as two different tests, even though their implementation might stay the same. Second, it excludes projects that only have a small number of test cases (< 10). If, for instance, a project only has two test cases, the result that 50 percent (i.e., one) of them is responsible for all test failures would be too coarse-grained for our purposes.

Algorithm 1. Sketch of Test Failure Percentage Calculation

```
1: procedure calcFailingTestPercentage(project)
2:   tcs.ok ← successful(testcases(project))   ▷List of every single successful execution of a test case
3:   tcs.failed ← failed(testcases(project))   ▷List of every single failed or errored execution of a test case
4:   tcs ← tcs.ok ∪ tcs.failed
5:   if n(unique(tcs) < 10) then              ▷Not enough test cases
6:     return
7:   end if
8:   fail.tc   ▷Map between a test case name (key) and the relative amount of test executions in which it failed (value)
9:   for tc ∈ unique(tcs.failed) do
10:    fail.tc(tc) ← n(tc ∈ tcs) / n(failed(tests(project)))
11:  end for
12:  values(fail.tc) ← order(values(fail.tc), descending)
13:  fail.perc   ▷Per percentage of all test cases, returns which percentage of failures they are responsible for
     ▷Invariants: fail.perc(0) = 0 and fail.perc(1) = 1
14:    for i ∈ {0%, 0.1%, 0.2%, ..., 100%} do
15:      first.i.tcs ← head(fail.rate, round(i · n(unique(tcs))))
16:      failure.rate(i) ← sum(values(first.i.tcs))
17:  end for
18:  return fail.perc
19: end procedure
```

3.3 Sequentialization of Intervals (RQ3, RQ4)

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 Fig. 5, the sequenced stream after the first test failure in (4) is:

`Failing Test` → `Switch Perspective` → `Start JUnit Test` → `Read Production Code` → ...

3.4 Test Flakiness Detection (RQ3)

Flaky tests are defined as tests that show non-deterministic runtime behavior: they pass one time and fail another time without modifications of the underlying source code or test [28]. Applied to the WatchDog interval concept, we

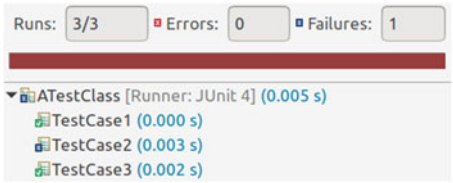

Fig. 6. Eclipse's visualization of the `JUnitExecution` constituents.

look for subsequent executions of test cases embedded in `JUnitExecution` intervals that have no `Typing` interval to either production or source code in-between them in the above linearized interval stream from Section 3.3. If the result of those subsequent executions differs, for example `Failing Test` $\rightarrow \cdots \rightarrow$ `Passing Test`, we regard such a test as flaky. To control for external influences, we only do this within the confines of a session, not across sessions. Otherwise, the risk for external influences becomes too large, for example through updating the project via the command line without our IDE plugin noticing.

3.5 Recognition of Test-Driven Development (RQ4)

Test-Driven development is a software development process originally proposed by Beck [29]. While a plethora of studies have been performed to quantify the supposed benefits of TDD [30], [31], it is unclear how many developers use it in practice. In RQ4, we investigate how many developers follow TDD to which extent. In the following, we apply Beck's definition of TDD to the WatchDog interval concept, providing a *formally verifiable definition of TDD* in practice. Since TDD is a process sequence of connected activities, it lends itself toward modeling as a state machine [32].

TDD is a cyclic process comprising a functionality-evolution phase depicted in Fig. 7, optionally followed by a functionality-preserving refactoring phase depicted in Fig. 8. We can best illustrate the first phase with the strict non-finite automaton (NFA, [33]) in Fig. 7a and our developer Wendy, who is now following TDD: before Wendy introduces a new feature or performs a bug fix, she assures herself that the test for the production class she needs to change passes (`JOk` in Fig. 7 stands for a `JUnitExecution` that contains a successful execution of the test under

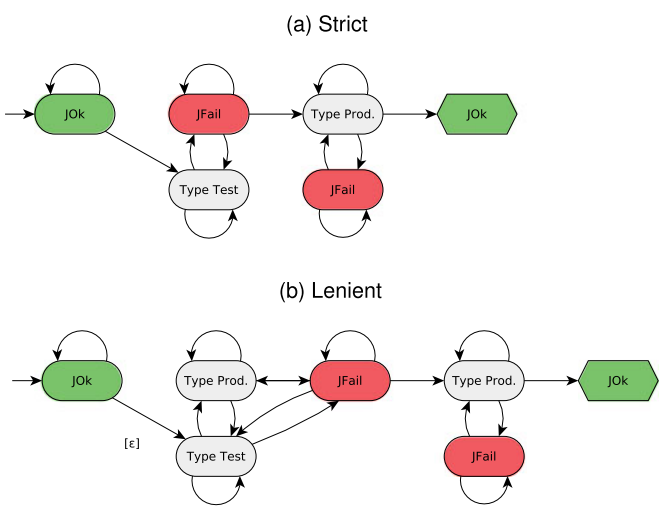

Fig. 7. Strict and lenient NFAs of TDD. `JOk` stands for a passing and `JFail` for a failing test execution (`JUnitExecution`).

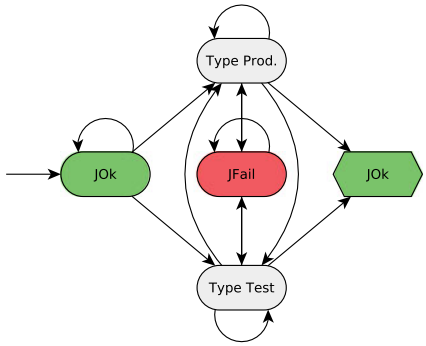

Fig. 8. NFA for the refactoring phase of TDD.

investigation). Thereafter, she 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 she is about to fix. We record such changes in a `Typing` interval on test code. Naturally, as Wendy has not yet touched the production code, the test must fail (`JFail`). Once work on the test is finished, Wendy switches to production code (`Type Prod.`), in which she makes precisely the minimal required set of changes for his failing test to pass again (`JOk`). The TDD cycle can begin anew.

When we applied this strict TDD process, we found that it is difficult to follow in reality, specifically the clear separation between changes to test code and later changes to production code. Especially when developing a new feature like the `Board` of a board game in Fig. 9, developers face compilation errors during the test creation phase of TDD, because the class or method they want to assert on (`Board`) does not exist yet, since the test has to be created before the production code. 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, because it contains obvious compile errors like in Fig. 9), 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 created the lenient non-finite automaton (ϵ-NFA, [33]) in Fig. 7b, 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 Fig. 8. In this phase, developers have the chance to perform functionality-preserving refactorings. Once they are finished with refactoring, the tests must still pass [29]. 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

Fig. 9. Compile errors while creating a TDD test.

TABLE 2
Descriptive Statistics of Study Data and Participants

IDE	Language	Plugin & Version	#Users	#Countries	#Projects	Work Time	#Sessions	#Intervals	Collection Period	Runtime
EC	Java	WD 1.0 – 2.0.2	2,200	115	2,695	146.2 years	66,623	12,728,351	15 Sept. 2014 – 1 March 2017	488 min
IJ	Java	WD 1.5 – 2.0.2	117	30	212	3.9 years	5,511	950,998	27 June 2015 – 1 March 2017	25 min
AS	Java	WD 1.7 – 2.0.2	71	27	178	1.0 year	2,717	347,468	26 Jan. 2016 – 1 March 2017	13 min
VS	C#	FB 0.1010 – 0.1015	55	unknown	423	9.7 years	2,259	239,866	12 June 2016 – 1 March 2017	13 min
Σ	Java, C#	WD, FB	2,443	118	3,508	161 years	77,110	14,266,683	15 Sep. 2014 – 1 March 2017	541 min
Σ_{CN}	Java, C#	WD, FB	181	38	434	33.9 years	15,928	3,137,761	15 Sep. 2014 – 1 March 2017	83 min

and match them against the linearized sequence of intervals (see Section 3.3). 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.

3.6 Statistical Evaluation (RQ1–RQ5)

When applying statistical tests in the remainder of this paper, we regard results as significant at a 95 percent confidence interval ($\alpha = 0.05$), i.e., iff $p \leqslant \alpha$. All results of tests t_i 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 [34]. 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* [35]. Hopkins's guidelines facilitate the interpretation of ρ [36]: they describe $0 \leqslant \mid \rho \mid < 0.3$ as no, $0.3 \leqslant \mid \rho \mid < 0.5$ as a weak, $0.5 \leqslant \mid \rho \mid < 0.7$ as a moderate, and $0.7 \leqslant \mid \rho \mid \leqslant 1$ as a strong correlation.

4 Study Participants

In this section, we first explain how we attracted study participants, report on their demographics, and then show how we produced a normalized sample.

4.1 Acquisition of Participants

We reached out to potential developers to install WatchDog (WD) and FeedBaG++ (FB) in their IDE by:

1) Providing project websites (WD, FB).[4]
2) Raffling off prizes (WD).
3) Delivering value to WatchDog users in that it gives feedback on their development behavior (WD).
4) Writing articles in magazines and blogs relevant to Java and Eclipse developers: Eclipse Magazin, Jaxenter, EclipsePlanet, Heise News (WD).
5) Giving talks and presentations at developer conferences: Dutch Testing Day, EclipseCon (WD).
6) Presenting at research conferences [8], [9], [13], [23], [37] (WD, FB).
7) Participating in a YouTube Java Developer series [38] (WD).
8) Penetrating social media: Reddit, Hackernews, Twitter, Facebook (WD, FB).
9) Approaching software development companies (WD, FB).
10) Contacting developers, among them 16,058 Java developers on GitHub (WD).
11) Promoting our plugins in well-established Eclipse [39], IntelliJ [40], and Visual Studio [41] marketplaces (WD, FB).
12) Launching a second marketplace that increases the visibility of scientific plugins within the Eclipse ecosystem, together with the Eclipse Code Recommenders project [42] (WD).
13) Promoting the plugin in software engineering labs at TU Darmstadt (FB).
14) Approaching an electrical engineering research group working with Visual Studio (FB).

We put emphasis on the testing reports of WatchDog to attract developers interested in testing. Instead, for FeedBaG ++, we mainly advertised its integrated code completion support.

4.2 Demographics of Study Subjects

Table 2 and Fig. 10 provide an overview of the observational data we collected for this paper. In total, we observed 14,266,683 user interactions (so-called intervals, see Section 2.1) in 77,110 distinct IDE sessions. Fig. 10a shows how 10 percent of our 2,443 users contributed the wealth of our data (80 percent). The majority of users and, thus, data stems from the Eclipse IDE, shown in Fig. 10b. Reasons include that the collection period for Eclipse is longer than that of the other IDEs and that we advertised it more heavily. In this paper, we report on an observatory field study stretching over a period of 2.5 years, on data we collected from the 15th of September 2014 to March 1st 2017, excluding student data that we had analyzed separately [9], but including our original developer data [8]. Data periods for other plugins are shorter due to their later release date. As we updated WatchDog to fix bugs and integrate new features (see Section 2.1.2), we also filtered out data from deprecated versions 1.0 and 1.1.

Our users stem from 118 different countries. The most frequent country of is the United States (19 percent of users), followed by China (10 percent), India (9 percent), Germany (6 percent), The Netherlands (4 percent), and Brazil (4 percent). The other half comes from the 112 remaining

4. http://www.testroots.org, http://kave.cc

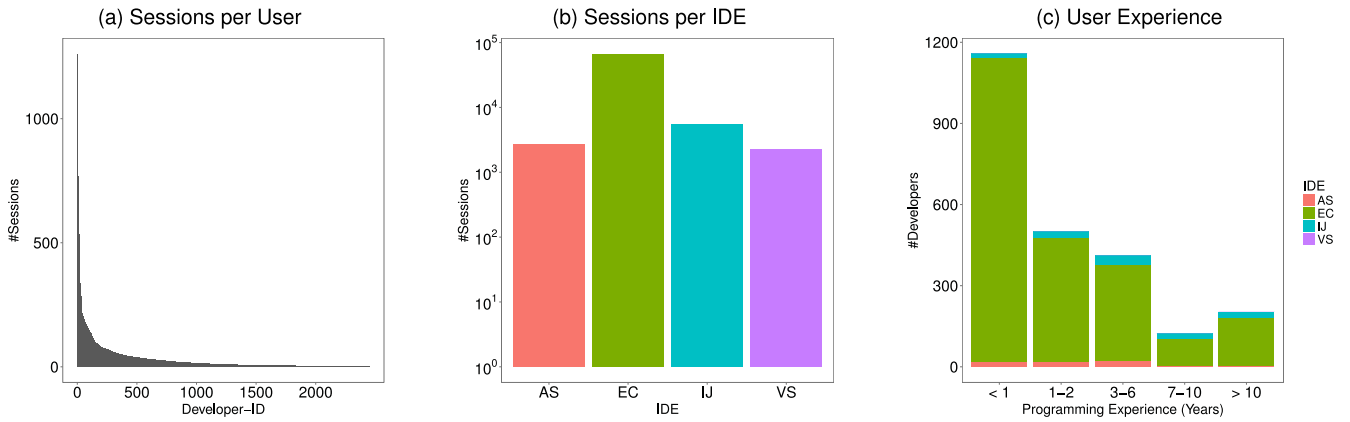

Fig. 10. Distributions of the number of sessions per developer (all IDEs), per IDE (log scale), and their programming experience (WatchDog).

countries, with less than 4 percent total share each. Our developers predominately use some variant of Windows (81 percent of users), MacOS (11 percent), or Linux (8 percent). Their programming experience in Fig. 10c is normally distributed (a Shapiro-Wilks test fails to reject the null hypothesis that it is not normally distributed at $p = 0.15$). Generally, we have more inexperienced (< 3 years, 69 percent of users) than experienced users. On the other hand, very experienced developers ($\geqslant 7$ years) represent more than 13 percent of our population.

Overall, the 2,443 participants registered 3,508 unique projects. The registered projects stem from industry as well as famous open-source initiatives, such as the Apache Foundation, but also include private projects.

Using the average work time for OECD countries of 1,770 hours per year,[5] we observed a total work time of 161 developer years on these registered projects in the IDE. The last column in Table 2 denotes the runtime of our analysis pipeline running on a dedicated server with 128GB RAM using eight Intel Xeon E5-2643 cores at 3.50 GHz.

This paper broadens our single-IDE study on developer testing in the IDE to a very large set of developers (a ten-fold increase over our original WatchDog data [9]). Survey responses from 2,291 registrations of WatchDog users and projects complement our technical IDE observations that now stem from four IDEs in two mainstream programming languages. FeedBaG++ data stems from the March 1st, 2017 event data set [43].

4.3 Data Normalization

As discussed in Section 4.2, the majority of our intervals (80 percent) stems from only 378 users. The long tail of users that contributed only little data might impact some of our analyses (see Fig. 10a). Conversely, the large amount of data we received from few developers might affect our results with a bias toward the individual development preferences of those few developers. To reduce both biases, we cap and normalize our data using stratified random sampling on the number of sessions per user. We chose sessions, because they are at a finer granularity than projects, but still allow analyses such as the TDD recognition, which would not work when sampling random intervals that have no connection to each other.

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

We first order our users by the number of sessions each user submitted and cap at below the user at which we reached 80 percent of all sessions. This leaves in users with at least 88 sessions each, effectively removing the bulk of users who barely contributed data and might, thus, skew user- or project-based analyses. The problem that few users have a disproportionately large impact on the analyzed data remains. Hence, we normalize the data by randomly sampling 88 of the available sessions for each user. After this, every user has the same influence on the results in our new capped, normalized data set, depicted as Σ_{CN} in Table 2. In comparison to our overall population Σ, the distribution of originating countries and IDEs is similar. The only apparent change in population demographics is an almost three-fold increase of very experienced developers to 32 percent in Σ_{CN}.

Since our study is a large-scale observatory field study, we primarily use our non-normalized data set Σ when answering research questions. Filtering criteria remain to some extent arbitrary and might induce a bias themselves. Whenever there is a significant difference in the capped normalized data set Σ_{CN}, we report and discuss this in the answer to the appropriate research question.

5 Results

In the following, we report the results to each of our research questions individually per section.

5.1 RQ1: Which Testing Patterns Are Common in the IDE?

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

RQ1.1 How Common Is Codified Testing in the IDE?

When we apply our broad recognition of test classes as described in Section 2.3.1 and Table 1, we detect test activities in only 43 percent of projects in our data set (EC: 46 percent, IJ: 26 percent, AS: 28 percent, VS: 26 percent), meaning that, in total, only 1,498 projects out of 3,508 contain tests that a user either read, changed, or executed in the IDE. This is one of the analyses that is potentially impacted by data skews due to a short amount of observed development behavior for many users. However, even in Σ_{CN}, only 255 projects out of 434 (58 percent) showed testing activity.

TABLE 3
Descriptive Statistics for RQ2 and RQ3 in the $\sum$ Data (Similar Across IDEs, Hence Abbreviated)

Variable	Unit	Min	25%	Median	Mean	75%	Max	Log-Histogram
`JUnitExecution` duration	Sec	0	0	0.5	107.2	3.1	652,600	
Tests per `JUnitExecution`	Items	1	1	1	5.0	1	2,260	
Time to fix failing test	Min	0	0.9	3.7	44.6	14.9	7,048	
Test flakiness per project	Percent	0	0	0	12.2	15.8	100	

If we restrict the recognition to tests that can be run through the IDEs, we find that 594 projects have such tests (EC: 436, IJ: 88, AS: 27, VS: 40), about 17 percent of the registered projects (EC: 16 percent, IJ: 22 percent, AS: 15 percent, VS: 9 percent). In Σ_{CN}, this percentage is somewhat higher at 29 percent, with 124 projects with executable tests. By comparing the WatchDog projects IDE data to what developers claimed in the survey, we could *technically* detect JUnit tests in our interval data (as either `Reading`, `Typing`, or `JUnitExecution`) for only 43 percent of projects that should have such tests according to the survey (EC: 42 percent, IJ: 61 percent, AS: 32 percent). Here, we find the only obvious difference in Σ_{CN}, where the percentage of users who claimed to have JUnit tests and who actually had them, is 73 percent.

Our second sub-research question is:

RQ1.2 How Frequently Do Developers Execute Tests?

Of the 594 projects with tests, we observed in-IDE test executions in 431 projects (73 percent, EC: 75 percent, IJ: 68 percent, AS: 37 percent, VS: 80 percent). In these 431 projects, developers performed 70,951 test runs (EC: 63,912, IJ: 3,614, AS: 472, VS: 2,942). From 59,198 sessions in which tests could have been run because we observed the corresponding project to contain an executable test at some point in our field study, we observed that in only 8 percent or 4,726 sessions (EC: 8.1 percent, IJ: 7.4 percent, AS: 3.4 percent, VS: 8.9 percent) developers made use of them and executed at least one test. The average number of executed tests per session is, thus, relatively small, at 1.20 for these 431 projects. When we consider only sessions in which at least one test was run, the average number of test runs per session is 15 (EC: 15.3, IJ: 11.1, AS: 7.6, VS: 17.9).

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 Excute Their Test Code Changes?

The correlation between test code changes and the number of test runs yields a moderately strong $\rho = 0.65$ (EC: 0.64, IJ: 0.60, AS: 0.41, VS: 0.66) in our data sample (p-value < 0.01). In other words, the more changes developers make to a test, the more likely are they to execute this test (and vice versa).

A logical next step is to assess whether developers run tests when they change the production code: Do developers assert that their production code still passes the tests?

RQ1.4 Do Developers Test Their Production Code Changes?

The correlation between the number of test runs and number of production code changes is generally weaker, with $\rho = 0.39$ (EC: 0.38, IJ: 0.47, AS: 0.20, VS: 0.60) and p-value < 0.01.

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?

In this case, the Spearman rank correlation test indicates no correlation ($\rho = 0.31$, EC: 0.26, IJ: 0.58, AS: 0.43, VS: 0.73) between the number of changes applied to test and production code. This means that developers do not modify their tests for every production code change, and vice versa.

5.2 RQ2: What Characterizes the Tests Developers Run in the IDE?

When developers run tests in the IDE, they naturally 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?

In all IDEs except for Visual Studio, 50 percent of all test executions finish within half a second (EC: 0.42, AS: 1.8s, IJ: 0.47s, VS: 10.9s), and over 75 percent within five seconds (EC: 2.37s, IJ: 2.17s, AS: 3.95s, VS: 163s), see Table 3 for the average values. Test durations longer than one minute represent only 8.4 percent (EC: 4.2 percent, IJ: 6.9 percent, AS: 6.1 percent, VS: 32.0 percent) of the `JUnitExecutions`.

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 and average the test runtime and the number of times developers executed tests in each session, as in Section 5.1. Then, we compute the correlation between the two distributions. If our hypothesis was true, we would receive a negative correlation between the test runtime 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 at $\rho = 0.27$ (EC: 0.40, IJ: 0.24, AS: 0.83, VS: 0.41). In Android Studio's case, the opposite is true, indicating a strong relationship between the runtime of a test and its execution frequency. Combined with the fact that only a small number of tests are executed, our results suggest that developers explicitly select test cases [44]. 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?

A test execution that we capture in a `JUnitExecution` interval may comprise multiple child test cases. However, 86 percent of test executions contain only one test case (EC: 86 percent, IJ: 88 percent, AS: 80 percent, VS: 85 percent), while only 7.7 percent of test executions comprise more than 5 tests (EC: 7.8 percent, IJ: 4.8 percent, AS: 7.6 percent, VS: 10.3 percent), and only 2.2 percent more than 50 tests (Table 3, EC: 2.2 percent, IJ: 0.1 percent, AS: 0.0 percent, VS: 4.4 percent).

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, moves, and deletions). The ratio between these two measures allows us to estimate the percentage of selected test cases. If it is significantly smaller than 100 percent, developers practiced test selection. Our data in Table 3 shows that 86.4 percent of test executions include only one test case.

To explain how and why this test selection happens with regard to a previous test run, we investigate two possible scenarios: 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 1,719 cases in which developers performed test selection, we can attribute 94.6 percent (EC: 94.6 percent, IJ: 91.8 percent, AS: 82.4 percent, VS: 95.5 percent) of selections to scenario 1, and 4.9 percent (EC: 5.2 percent, IJ: 0.0 percent, AS: 5.8 percent, VS: 3.6 percent) to scenario 2. Hence, our two scenarios together are able to explain 99.5 percent (EC: 99.8 percent, IJ: 91.8 percent, AS: 88.2 percent, VS: 99.1 percent) of test selections in the IDE.

5.3 RQ3: How Do Developers Manage Failing Tests?

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 test execution can return an unsuccessful result: The 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 70,951 test executions, 57.4 percent of executions fail, i.e., 40,700 `JUnitExecutions` (EC: 57.4 percent, IJ: 60.7 percent, AS: 56.8 percent, VS: 43.2 percent), and only 42.6 percent pass successfully. Moreover, when we regard the child test cases that are responsible for causing a failed test execution, we find that in 86 percent (EC: 95 percent, IJ: 84 percent, AS: 88 percent, VS: 94 percent) of test executions only one single test case fails, and is, thus, responsible for making the whole test execution fail, even though other test cases from the same test class might pass, as exemplified in Fig. 6.

To zoom into the phenomenon of broken tests, we ask:

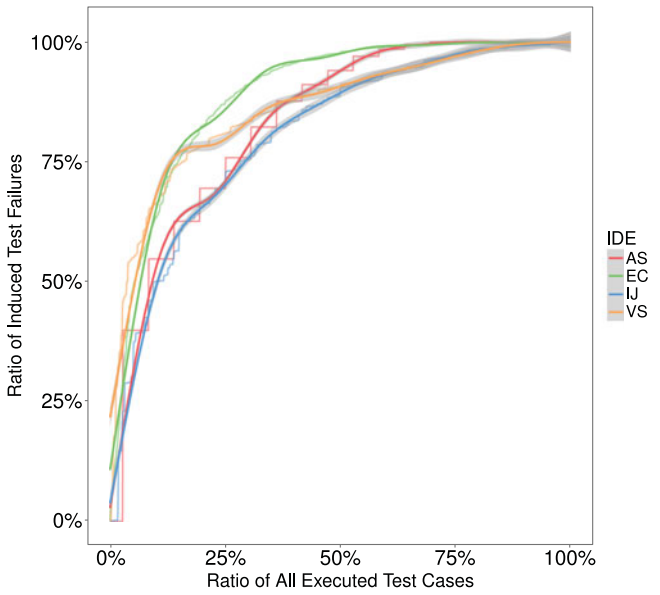

Fig. 11. Accumulated error responsibility of test cases per IDE. Based on 134 projects with ≥ 10 run test cases (EC: 112, IJ: 9, AS: 1, VS 12).

RQ3.2 Are All Test Cases Equally Responsible for Test Failures?

In this question, we regard all test cases that have ever been executed and observed. We then calculate and track how many times each of them failed, as described in detail in Section 3.2. Since we cannot track renames of files and, therefore, treat them as two different files, it is likely that the real error percentage for test cases is slightly higher. Fig. 11 depicts the results, showing that only 25 percent of test cases are responsible for over 75 percent of test failures in Eclipse and Visual Studio. In all IDEs, 50 percent of test cases are responsible for over 80 percent of all test errors. While slightly lower for IntelliJ-based IDEs, the failure and growth rate of the curve is similar across IDEs, suggesting a near-logarithmic growth.

As developers apparently often face test failures, we ask:

RQ3.3 How Do Developers React to a Failing Test?

For each failing test execution in our data sets, we generate a linearized stream of subsequently following intervals, as explained in Section 3.3. By counting and summing up developers' actions after each failing test for up to 3.3 minutes (200 seconds), we can draw a precise picture of how developers manage a failing test in Fig. 12. Across all IDEs, the most widespread immediate reaction in ~50 percent of cases within the first seconds is to read test code.[6] The second most common reaction, at stable 20 percent of reactions across the time, is to read production code.

The next most common reactions—switching focus away from the IDE (for example, to turn to the web browser), switching perspective in the IDE (for example to a dedicated debugging perspective), typing test code, and being inactive—appear in different order among IDEs. Typing

6. While writing this extension, we uncovered a bug in the analysis code to RQ3.3. The bug swapped the "Read Test Code" with the "Read Production Code" label. This lead us to wrongly claim in the original WatchDog paper [8] that developers dived into offending production code first, which was never the case.

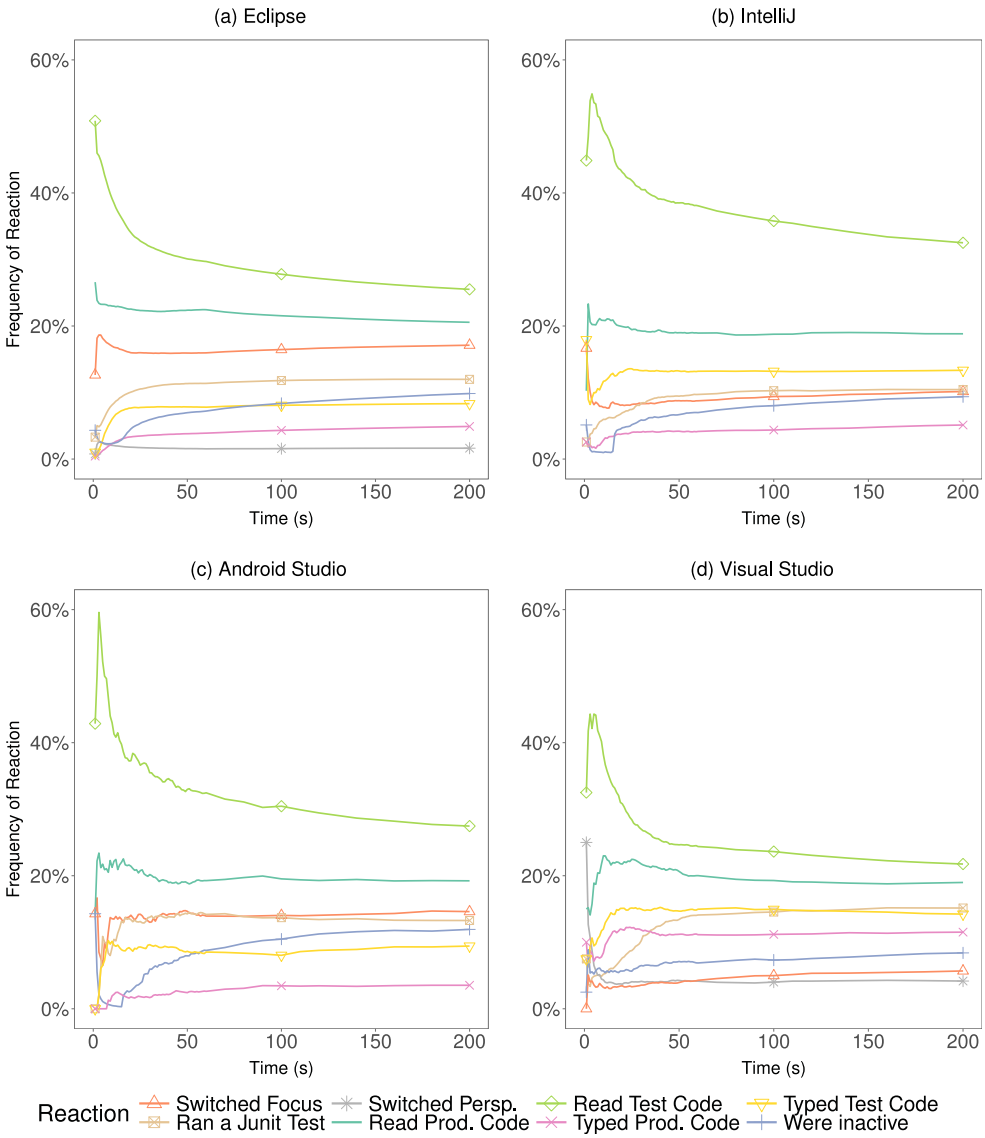

Fig. 12. Frequency of immediate reactions to a failing test over time, separated by IDE.

test code, however, is a more common reaction to a failing test in all IDEs than typing production code. Starting another test execution is a fairly common course of action within the first minute across all IDEs, reaching ~15 percent frequency. Switching perspective is only prevalent in the first seconds (see Fig. 12d), since it is an automated feature of Visual Studio (see Section 2.3.2). Altogether quitting the IDE almost never happens and is, therefore, not shown. After two minutes (120 seconds), the reactions trend asymptotically toward their overall distribution, with little variability.

The logical follow-up to RQ3.3 is to ask whether developers' reactions to a failing test are in the end successful, and:

RQ3.4 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 40,700 failing test executions were caused by 15,696 unique test classes according to their file name hash (EC: 13,371, IJ: 959, AS: 94, VS: 1,271). We never saw a successful execution of 32 percent (EC: 28 percent, IJ: 50 percent, AS: 46 percent, VS: 54 percent) of tests, and at least one successful execution of the others.

For the 10,701 failing tests that we know have been fixed later, we examine how long developers take to fix a failing test. Table 3 shows that a quarter of test repairs happen within less than a minute, half within 4 minutes, and 75 percent within 15 minutes.

One reason why in some cases the time between a failing and succeeding test might be so short is that developers did not actually have to make repairs to their tests. Instead, they might have just executed the tests without changes, since it might be flaky. A flaky test is a test that shows non-deterministic pass behavior [45], [46], meaning it (randomly) fails or succeeds. To answer this question, we ask for the IDE:

RQ3.5 Do Developers Experience Flaky Tests?

Following the research method described in Section 3.4, we measure the "test flakiness" per project, the percentage of tests that show non-deterministic behavior despite the fact that there are no changes to the project in the meantime, including changes to test, production, or configuration files. Table 3 shows that the mean flakiness value is 12.2 percent, with outliers of zero and 100 percent flaky test percentages.

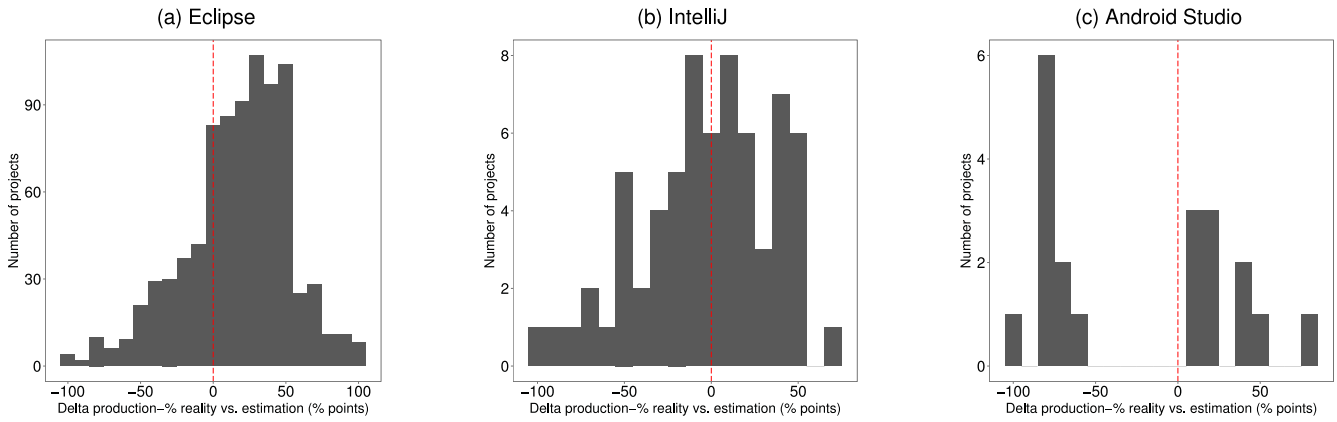

Fig. 13. Difference between estimated and actual time spent on testing split per IDE (no data for FeedBaG++).

5.4 RQ4: Do Developers Follow TDD in the IDE?

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

Our results reveal that sessions of only 43 developers match against a strict TDD definition, the top NFA in Fig. 7a (EC: 42, IJ: 0, AS: 0, VS: 1). This makes 1.7 percent of all developers, or 11.8 percent of developers who executed tests, see Section 5.1. In total, only 2.2 percent of sessions with test executions contain strict TDD patterns. Only one developer uses strict TDD in more than 20 percent of the development process on average. Seven of the 43 developers use TDD for at least 5% of their development. The remaining 35 developers use strict TDD in less than 5% of their intervals. Refactoring is the dominant phase in TDD: 39 of the 43 developers did some form of refactoring. At 69 percent, the majority of the intervals of the 43 developers are devoted to the refactoring phase of TDD (depicted in Fig. 8). Most developers who practiced strict TDD have a long programming experience: 23 declared an experience between 7 and 10 years.

Sessions from 136 developers match against the lenient TDD NFA in Fig. 7b (EC: 42, IJ: 18, AS: 3, VS: 3). This makes 5.6 percent of all developers, or 37 percent of developers who executed tests (EC: 15 percent, IJ: 38 percent, AS: 33 percent, VS: 19 percent), see Section 5.1. Sixteen developers use lenient TDD in more than 20 percent of their intervals, including the developer who has over 20 percent strict TDD matches. 28 developers use lenient TDD in more than 10 percent, but less than 20 percent of their intervals. 98 of the 136 developers who use lenient TDD also refactor their code according to the TDD refactoring process in Fig. 8. For them, 48 percent of intervals that match against the lenient TDD are due to refactoring. Of the 136 developers, 49 have little programming experience (0–2 years), 25 have some experience (3–6 years), and the majority of 59 is very experienced (> 7 years).

In our normalized data set, the results on the use of TDD are somewhat higher, with 6 percent of users following strict, and 22 percent following lenient TDD. The distribution of testing- and refactoring is similar to the Σ values.

However, even top TDD users do not follow TDD in most sessions. For example, the user with the highest TDD usage has one session with 69 percent compliance to TDD. On the other hand, in the majority of the remaining sessions, the developer did not use TDD at all (0 percent). We verified this to be common also for the other developers who partially used TDD. These low results on TDD are complemented by 574 projects where users *claimed* to use TDD, but in reality only 47 of the 574 *did* according to our definition.

5.5 RQ5: How Much Do Developers Test in the IDE?

In WatchDog clients, we asked developers how much time they spend on engineering tests. To compare survey answers to their actual development behavior, 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 mostly short test duration is negligible compared to the time spent on reading and typing (see Section 5.2). 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.

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 (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 data set, by *averaging* across project and *not* normalizing for the contributed development time (only multiplying each estimation with it).

Per-Project Measurement. Following Halkjelsvik et al. [47], Fig. 13 shows the relative directional error of estimations as a histogram of the differences between the measured production percentage and its estimation *per project*. A value of 0 means that 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 picture on the correctness of estimations is diverse. In Eclipse, developers tend to overestimate their testing effort by 17 percent-points, see Fig. 13a, where the median of the distribution is shifted to the right of 0, marked by the red line. While there are much fewer observations, the reverse is true for Fig. 13c with an error of -23.4 percent-points. At an average estimation

difference of -2.2 percent, IntelliJ developers seemed to be most accurate. Moreover, they have fewer extreme outliers than Eclipse (axes labels of Figs. 13a and 13b). However, the distribution of estimations in Fig. 13b shows that the average mean value can be deceiving, as the graph demonstrates a broad proliferation of evening-out estimations from -40 percent to +50 percent, but no spike at 0 percent. There are relatively few observations for Android Studio (20) and IntelliJ (67) in comparison to Eclipse. On a per project-base, the average mean time spent testing is 28 percent (EC: 27 percent, IJ: 38 percent, AS: 51 percent, VS: 27 percent). However, developers estimated a distribution of 51 percent on production code (EC: 56 percent, IJ: 64 percent, AS: 73 percent), and 49 percent on tests, so they overestimated the time spent on testing by 21 percent percentage points, or 1.75 times.

Averaged Measurement. When we do not normalize the data per project for our whole data set Σ, we find that all developers spend in total 89 percent of their time writing or reading production classes (EC: 89.3 percent, IJ: 98.5 percent, AS: 84.0 percent VS: 60.0 percent), and 11 percent of their time on testing (EC: 10.7 percent, IJ: 1.5 percent, AS: 16.0 percent, VS: 40.0 percent). These implausibly large differences to the normalized testing percentage of 28 percent and between the IDEs remind us to consider Σ_{CN} again. Its average mean test percentage of 26.2 percent confirms the per-project normalized measurement we reported above (28 percent). We therefore use these values in the discussion.

Moreover, reading and writing are relatively uniformly spread across test and production code: while developers read production classes for 96.6 percent of the total time they spend in them, they read tests longer, namely 96.9 percent of the total time they spend in them.

6 Discussion

In this section, we interpret the results to our research questions and put them in a broader perspective.

6.1 RQ1: Which Testing Patterns Are Common in the IDE?

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. The test detection rate in the Eclipse-based client is almost twice as high as in the other clients. A possible reason might be that we concentrated our testing advertisement efforts on Eclipse. An investigation of testing practices on the popular Continuous Integration (CI) server Travis CI showed a somewhat higher test rate at 69 percent for Java projects [15]. Reasons might be that testing is the central phase of CI [15], [48] and that projects that have set up Travis CI might be more mature in general. This frequency is closer to the 58 percent we found in our normalized data set. Moreover, our IDE observation does not mean that the projects contain no tests (a repository analysis might find that there exist some), but it does indicate that testing is not a prime activity of the registered WatchDog developers. Alarmingly, only 43 percent of the projects that claimed to have JUnit tests in the survey actually had intervals showing tests ("truth tellers"). For the other 57 percent, their developer did not execute, read, or modify any test in the observation period. The varying amount of data we received from users impacts this measure, since we are more likely to detect test activity within a large amount of general activity for one user than when we have little data overall. Our data distribution suggests that normalization should give us a more realistic picture, see Fig. 10a. Consequently, Σ_{CN} has a "truth teller" ratio of 73 percent. Since we likely overestimate tests, these two discoveries raise questions: Which value does testing have in practice? And, further, are (anonymous) developers' survey answers true and which measures are suitable to ensure correctness of our conclusions?

> Roughly half of projects and users do not practice testing in the IDE actively.

Only 17 percent of all projects comprise tests that developers can run in the IDE. The values across IDEs are relatively similar. We assume the real percentage is similar for Visual Studio, but shows lower due to the fact that tests are organized in their own project, see Section 2.3.2. For 27 percent of projects that have executable IDE tests developers never exercise the option to execute them. This gives a hint that testing might not be as popular as we thought [49]. Reasons might include that there are often no pre-existing 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 on a constant basis. The apparent lack of automated developer tests might be one factor for the bug-proneness of many current software systems.

Even for projects that 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 (15).

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

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 [50] and rely on it to trigger the execution of the tests on the CI server. In this case, 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, both anecdotal evidence and 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 CI server support in future releases to facilitate this new workflow of developers.

Every developer is familiar with the phrase "Oops, I broke the build" [51]. 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 [52]. 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], [53]. 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. Supporting this observation, Romano et al. found that, even when following TDD, developers "write quick-and-dirty production code to pass the tests, [and] do not update their tests often" [54].

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

6.2 RQ2: What Characterizes the Tests Developers Run?

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.

While still being fast, a notable exception to this are the tests run in Visual Studio, which took an order of magnitude longer. One reason for this could be that many C# tests might rely on additional base tests that take longer to setup. For example, the tests for FeedBaG++ require a specific base test of ReSharper, which takes 60 seconds to initialize. Another reason could be that Visual Studio facilitates debugging by running tests in the debugger automatically. Pausing on a breakpoint would be added to the tests' runtime.

We could generally 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 that takes 0.1 seconds and one that 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. Instead, our positive correlation values suggest that developers prefer tests that take slightly longer, for example because they assert more complex constructions. Thus, they might be more beneficial to developers than straight-forward, very short tests. In fact, short tests might be so limited in their coverage that developers might not find them useful enough to run them more often. This might be particularly relevant for testing mobile applications, where the typically longer running integration tests require developers to start up an Android Emulator. Our strong correlation for Android Studio suggests that developers prefer running such longer tests.

One reason for the generally 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 percent 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 99.5 percent of these test selections with two scenarios: developers either want to investigate a possibly failing test case in isolation (94.6 percent of test selections), or exclude such an irritating test case from a larger set of tests (4.9 percent). 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 [55].

6.3 RQ3: How Do Developers Manage Failing Tests?

One other possible explanation for the short time it takes tests to run in the IDE is that 65 percent of them fail (RQ3.1): 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.

This is a substantial difference to testing on Travis CI, where only 4 percent of Java builds fail due to failing tests [15]. For 32 percent of the failing test cases, we never saw a successful execution (RQ3.4). 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 specific scenario is rare, as observed at the commit-level by Pinto et al. [3]. Consequently, a substantial part of tests (up to 32 percent) 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.

This observation motivated us to explore which test cases failures typically stem from.

> Only 25 percent of test cases are responsible for 75 percent of test execution failures in the IDE.

This statement reminds us of the Pareto principle [56], the startling observation that, for many events, roughly 80 percent of the effects stem from 20 percent of the causes. The principle has been observed in Software Engineering in alike circumstances before, for example that 20 percent of the code contains 80 percent of its errors [57].

On the CI side, test executions are the main part of how fast a project builds [15]. To manage the problem of long and expensive builds, Herzig et al. built an elaborate cost model deciding which tests to skip [58]. A simulation of their model on Microsoft Windows and Office demonstrated that they would have skipped 40 percent of test executions. Using association rule mining based on recent historical data, such as test failure frequency, Anderson et al. demonstrated how they could reduce the duration of regression testing for another Microsoft product by also leaving out a substantial amount of tests [59]. Similarly, Fig. 11 shows that at least in Eclipse and Android Studio,

running the right 60 percent of test cases (and skipping 40 percent) results in catching all test failures. For IntelliJ and Visual Studio, the results are at a comparable ∼90 percent. Thus, if we can select them efficiently, we can skip executing ∼40 percent of test cases that always give a passing result in the IDE.

Both Microsoft studies have been performed on the build level, not as deep down as our findings in the "working mines of software development," the IDE. This observation on the build level trickles down to the IDE, where one would expect more changes than on the CI level. Moreover, it also shows that some tests never fail, even during the change-prone phases of development, reducing the value of such tests at least for bug-uncovering purposes.

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 test code.

All observed IDEs support this work flow by presenting the developer with the location of the test failure in the test class when double-clicking a failed execution. It is, thus, a conscious choice of the programmers to instead dive into production code (20 percent of reactions) that is being tested. Closing the IDE, perhaps out of frustration that the test fails, or opening the debug perspective to examine the test are very rare reactions. It is only prevalent in Fig. 12d because Visual Studio automatically switches to this perspective when running tests. Five seconds after a test failure, ∼15 percent 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 (mis-)using a language construct, the standard library, or other well-known APIs and frameworks. Researchers try to integrate answers from internet fora such as Stack Overflow into the IDE [60], to make this possibly interrupting context switch unnecessary.

> 12 percent of test case executions show a non-deterministic result.

Flaky tests are a phenomenon that has been studied on the repository [28] and build [45], [61] level. Luo et al. classified root causes of flaky tests. They found that asynchronous waiting, concurrency, and test order dependency problems represent 77 percent of test flakiness causes. Including all potential factors, we have calculated a flakiness score of on average 12 percent of test cases per project in the IDE. A study on the flakiness of tests run on the CI server Travis CI [62] found a similar flakiness rate of 12.8 percent [61]. This is another instance of a finding on a build server level that seems to directly translate to the IDE of individual developers. Moreover, the test flakiness of 12 percent fits well to an observed reaction of (re-)executing tests 10 seconds after the initial test failure in 15 percent of cases for most IDEs in Fig. 12.

> Findings on the CI level on test flakiness and error responsibility seem to trickle down to the IDE of individual developers.

6.4 RQ4: Do Developers Follow TDD?

TDD is one of the most widely studied software development methodologies [30], [31], [63].[7] Even so, little research has been performed on how widespread its use is in practice. In Section 3.5, we developed a formal technique that can precisely measure how strictly developers follow TDD. In all our 594 projects, we found only 16 developers that employed TDD for more than 20 percent of their changes. Similar to RQ1, we notice a stark contrast between survey answers and the observed behavior of developers, even in our normalized control data set. Only in 12 percent of the projects in which developers claimed to do TDD, did they actually follow it (to a small degree).

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

The developers who partially employed TDD in our data set were more experienced in comparison to the general population. We also found a higher TDD rate in our normalized data set, likely due to the fact that Σ_{CN} has more experienced users compared to Σ and TDD followship correlates with experience.

Two recent studies support these discoveries on TDD. Borle et al. found an almost complete lack of evidence for TDD adoption in the repositories of open source GitHub projects [64]. Romano et al. found that both novice and expert programmers apply TDD in a shallow fashion even in a controlled lab experiment dedicated to TDD [66]. As a cardinal difference to our field study Romano et al. found that "refactoring [...] is not performed as often as the process requires" [66], while we found developers devoting over 50 percent of their TDD development intervals to the readoption of code. A reason might be that refactoring is inevitable in most real-world software projects, but can perhaps be avoided in a lab assignment setting.

In the following, we discuss a number of possible reasons for the apparently small adoption of TDD in practice:

1) *There is no consensus on the usefulness and value of TDD.* While there have been many controlled experiments and case studies in which TDD was found to be beneficial, there seems to be an equally high number of studies that showed no, or even adverse effects [67], [68], [69]. Moreover, some of the pro-TDD studies contradict each other on its concrete benefits: For example, Erdogmus measured that the use of TDD leads to a higher number of tests and increases productivity [70], while in a case study at IBM, TDD did not affect productivity, yet decreased the number of defects [71]. Another study at IBM

7. A Google Scholar search for "Test Driven Development" returned 15,400 hits on May, 18th, 2016, while the much older "Cleanroom Software Engineering" only returned 1,350 hits and the popular "Code Review" 17,300 hits.

and Microsoft, done in part by the same authors, found that defects decreased drastically, yet productivity declined with the introduction of TDD [72]. In light of no clear evidence for TDD, developers might simply choose not to employ it.

2) *Technical practicalities prohibit the use of TDD.* Some libraries or frameworks do not lend themselves for development in a TDD-fashion. As an example, few graphical toolkits allow development in a test-first manner for a UI.
3) *Time or cost pressure prohibits the use of TDD.* TDD is often associated with a slower initial development time, and the hope that the higher quality of code it produces offsets this cost in the longer run [73], [74]. At high pressure phases or for short-lived projects, a concise decision not to use TDD might be made.
4) *Developers might not see value in testing a certain functionality TDD-style.* We received anecdotal evidence from developers saying that they do not think the current piece of functionality they are working on mandates thorough testing, or "simply cannot be wrong."
5) *Developers skip certain phases of the required TDD process.* We received anecdotal evidence from developers saying that they "sometimes know the result of a test execution." Consequently, they might skip the mandatory test executions in Fig. 7.
6) *The application of TDD might be unnatural.* Instead of working toward a solution, TDD puts writing its specification first. This often requires developers to specify an interface without knowing the evolution and needs of the implementation. Romano et al. accordingly report that developers found the "red" test phase of TDD, in which developers are supposed to perform the above steps, particularly challenging and demotivating [66].
7) *Developers might not know how to employ TDD.* TDD is a relatively light-weight development methodology that is taught in numerous books [29], blog posts, articles, YouTube videos, and even part of the ACM's recommendations on a curriculum for undergraduate Software Engineering programs [75]. By contrast, Janzen and Saiedian noted that one common misconception among developers was that "TDD equals automated testing." [76] Since Beck defines TDD as "driv[ing] development with automated tests" [29], we believe practitioners have understood it correctly and that a lack of education on TDD or a wrong understanding of it is not a likely reason in most cases.

While TDD might be clear enough for all practitioners, for academic studies, we still miss a precise, formally agreed-upon definition. In fact, the lack of it might explain some of the variability in the outcomes of research on the benefits of TDD. We hope that our precise definition of TDD in terms of automata from Section 3.5 can help future research on a technical level.

> We need to convene on a generally agreed-upon, formal definition of TDD.

In his 2014 keynote at Railsconf and subsequent blog posts [77], [78], Heinemeier Hansson sparked a debate on the usefulness and adoption of TDD, leading to a series of broadcast discussions together with Fowler and Beck on the topic "Is TDD dead?" [79]. Since our WatchDog results seemed relevant to their discussion, we approached Beck, Fowler, and Heinemeier Hansson with our paper [8] to uncover if we made any methodological mistakes, for example that our model of TDD might be erroneous. Fowler and Heinemeier Hansson replied that they were generally interested in the results of the study and identified the potential sampling bias also discussed in Section 7.4. Regarding the low TDD use, Fowler stated that he would not be surprised if developer testing of any kind remains uncommon.

6.5 RQ5: How Much Do Developers Test?

The question of how much time software engineers put into testing their application was first asked (and anecdotally answered) by Brooks in 1975 [7]. In contrast to our study, Brooks' numbers cover the entire development and not only software developers themselves. Nowadays, it is widely believed that "testing takes 50 percent of your time." While their estimation was remarkably on-par with Brooks' general estimation (average mean 50.5 percent production time to 49.5 percent test time, median 50 percent) in Fig. 2, WatchDog developers tested considerably less than they thought they would at only 28 percent of their time, overestimating the real testing time nearly two-fold. The time developers spend testing is relatively similar across all IDEs, with the only apparent outlier of Android Studio (51 percent). Mobile application developers might indeed spend more time testing since the Android framework facilitates unit, integration, and UI testing ("Android Studio is designed to make testing simple. With just a few clicks, you can set up a JUnit test that runs on the local JVM or an instrumented test that runs on a device" [80]), or our developer sample from Android Studio might be too small. We need more research to better understand this phenomenon and the reasons behind it.

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

In comparison, students tested 9 percent of their time [9], 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.

The ability to accurately predict the effort and time needed for the main tasks of a software project (such as testing) is important for its coordination, planning, budgeting and, finally, successful on-time completion. In a comprehensive review of the research on human judgments of task completion durations, Halkjelsvik and Jørgensen merged the two research lines of effort prediction from engineering domains and time-duration estimation from psychology [47]. Their results showed that duration predictions frequently (more than 60 percent of predictions) fall outside even a 90 percent confidence interval given by the estimators, meaning that it is normal for predictions to be as inaccurate as observed in our study. While engineers generally seem to overestimate the duration of small tasks, they underestimate larger tasks. As testing is the smaller activity in comparison to production

code for most projects (~25 percent:75 percent of work time overall), this observation fits the measured overestimation of testing effort in our study. There might be a tendency to underestimate difficult and overestimate easy tasks, particularly in software development projects [81]. As developers often dislike testing and consider it "tedious" [82], [83], this might be a contributing factor to our observed overestimation of testing. In a study on software maintenance tasks by Hatton [84], developers consistently overestimated the duration of small change requests, while they consistently underestimated larger ones. Many developers might perceive testing as the smaller task in relation to the seemingly endless complexity of coming up with a working production implementation. Consequently, Hatton's findings could help explain why our participants overestimated it. Similar to our study, Halkjelsvik and Jørgensen report that working professional engineers, while still not accurate, were better in their predictions than students [47].

A prime reason for developers' inaccurate estimations might be that predicting is an inherently difficult task, especially in fast-changing domains like software development. Another reason for the inaccuracy of predictions could be that people remember the time previous tasks took incorrectly [85]. When participants had written records of their past prediction performances, however, they became quite accurate [86]. Highly overestimating the testing effort of a software product can have adverse implications on the quality of the resulting product. Software developers should, therefore, be aware of how much they test, and how much their perception deviates from the actual effort they invest in testing in the IDE. WatchDog supplies developers with both immediate and accumulated statistics, hopefully allowing them to make more precise estimations and better planning in the future.

In conjunction with RQ1 and RQ3, our discrepancy between survey answers and real-world behavior casts doubt on whether we can trust untriaged answers from developers in surveys, especially if the respondents are unknown to the survey authors.

> Objectively observed behavior in the IDE often contradicted survey answers on developers' self-estimation about testing and TDD, showcasing the importance of data triangulation.

6.6 A Note on Generality and Replicability

Long-running field studies and studies that generalize over multiple factors, such as IDEs or languages, are rare in software engineering [5], [12], [87], [88], because building and maintaining the necessary tools requires significant time efforts over prolonged periods of time. Moreover, we show that it is possible to re-cycle data that was originally not intended for this study by including the FeedBaG++ client. This paper demonstrates that even controversial, unexpected results such as our original observations on testing patterns [9], can generalize across different state-of-the art IDEs. Our larger study—comprising ten times more data and three more IDEs—confirmed most of the observations we drew from a much shorter, less resource-intense 5-month study in only Eclipse.

> If argued correctly, even a relatively small number of observations in one environment can generalize to similar contexts in Software Engineering.

We needed to normalize only relatively few of our results, leaving most of our observations straight-forward to derive from our data. However, for some research questions, for example to counter the appearance that developers might test 20 times longer in one IDE than in another (see RQ5, Section 5.5), normalization was critical. Since filter criteria always induce a bias, this paper also shows how an observational field study can use unfiltered and easy-to-interpret and replicate data and combine it with the smaller normalized data sample where necessary.

Our mixed-methods study also showcased the problem of reporting survey answers without further triaging. While normalizing the data improved their credibility, even with it, there was a considerable mismatch between developers' actions and their surveyed answers and beliefs. A diverse set of factors, including psychological ones, seems to play a key role in this.

6.7 Toward a Theory of Test-Guided Development

Combining results from RQ1–RQ5, we find that most developers spend a substantial amount of their time working on codified tests, in some cases more than 50 percent. However, this time is shorter than expected generally and specifically by the developers themselves. Many of the tests developers work on cannot be executed in the IDE and could, therefore, not provide immediate feedback. There are relatively short development phases when programmers execute IDE-based tests heavily, followed by periods when they invoke almost no tests.

Test and production code evolution in general is only loosely coupled. This corroborates with our finding that no developer follows TDD continuously and that it, thus, seems to be a rather idealistic software development method that a small percentage of developers occasionally employs with overall little adoption in practice. We call the development practice of loosely guiding one's development with the help of tests, as the majority of developers does, relying on testing to varying degrees, *Test-Guided Development* (TGD). We argue that TGD is closer to the development reality of most programmers than TDD.

Two insights from our study, test flakiness and test failure rate, seem to be almost identical in the context of CI, showing the strong connection to individual developer testing in the IDE. However, there are also significant differences, namely that CI provides no fast feedback loop to developers, by taking on average 20 minutes, several orders of magnitudes longer than a typical IDE test execution [15]. Test failures are much more infrequent in Java builds than in test executions in the IDE. We, therefore, argue that it plays a different, complimentary role to testing in the IDE. Due to its different and less immediate nature, CI testing cannot (fully) explain the observed low values on developer testing.

7 Threats to Validity

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

7.1 Limitations

Our study has two main limitations, scope and a lack of value judgments, which we describe in the following.

Scope Definition. An endemic limitation of our study is that we can only capture what happens inside the IDE. Conversely, if developers perform work outside the IDE, we cannot record it. Examples for such behavior include pulling-in changes through an external version-control tool, such as `git` or `svn`, or modifying a file with an external editor. To reduce the likelihood and impact of such modifications, we typically limit analyses of our research questions, for example RQ3.5 regarding test flakiness, to one IDE session only.

Naturally, for RQ5, we cannot detect work on a whiteboard or thought processes of developers, which are generally hard 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.

> Our conclusions are drawn from the precisely-defined and scoped setting of codified developer testing in IDEs. To draw a holistic picture of the state of testing, we need more multi-faceted research in environments including dedicated testers.

Value Judgments. If we want to gain insight into whether more developer testing manifests in an improvement for the project, we would need to define a suitable outcome measure, for example bugs per release. One could then, for example, compare the testing effort in days across several releases, and identify whether there is a correlation. However, different projects would have different, possibly contradicting, definitions of the outcome measure: A self-managed server project in the cloud might pay no attention to bugs per release, as releases are short-lived and upgrading the software is essentially cost free. On the other hand, a server installed at a customer that cannot be reached from the outside might have this metric as its only priority. We have not defined a uniform outcome measure because (1) we could not define a sensible uniform outcome measure across all participating projects of their different priorities, (2) many developers preferred to stay anonymous, and (3) do not have or (4) would not have given us access to this highly sensible data. One can argue that if a project reaches its desired outcome with the limited amount of testing we generally found in this study, this is better than having to spend a lot of effort on testing, it in principle wastes resources without contributing to the project's functionality. This remains a fruitful future area for deep studies on a small set of projects.

> This paper does not contain an outcome measurement. As such, all statements are comparative to the respective groups and non-judgmental. A relative high (or low) description does not mean imply "good" or "bad."

7.2 Construct Validity

Construct validity concerns errors caused by the way we collect data. For capturing developers' activities we use WatchDog and FeedBaG++ (described in Section 2.1), which we thoroughly tested with end-to-end, integration, and developer tests. Moreover, 40 students had already used WatchDog before the start of our data collection phase [9]. Similarly, FeedBaG++ had been deployed at a company during 2015 [23] before we made it publicly available in 2016. 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 development sessions, which we replicated in Eclipse, IntelliJ, and Visual Studio. We then ran our analysis pipeline and ensured the analyzed results were comparable.

When we compare data across IDEs, it is paramount that the logic that gathers and abstracts this data (to intervals) works in the same way. WatchDog's architecture with its mutually shared core guarantees this by design (see Section 2.1). Moreover, we had a professional software tester examine WatchDog.

To ensure the correctness of the transformation from FeedBaG++ events to WatchDog intervals, we implemented an extensive test suite for the transformation on the FeedBaG++ side and created a debugger that visualizes intervals similarly to the diagram shown in Fig. 5. We used this visualization for an analysis of several manually defined usage scenarios, in which we verified that the generated intervals are accurate and that they reflect the actually recorded interactions. Moreover, we recorded artificial mini-scenarios with FeedBaG++, transferred them to WatchDog intervals and ran parts of the analysis pipeline, for example for the recognition of TDD behavior, effectively creating end-to-end tests.

7.3 Internal Validity

Internal validity regards threats inherent to our study.

Our study subject population shows no peculiarity (see Section 4.2), such as 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 (Fig. 10a). To control for the possible effects of a distorted distribution, we created a normalized data set Σ_{CN}, which showed little practical difference to our main sample. Moreover, contrary to the idea of conducting an open field study, we run the risk of arbitrarily selecting for certain behavior by sampling. Since WatchDog and FeedBaG++ are freely available, we cannot control who installs it. Due to the way we advertise it (see Section 4.1), our sample might be biased toward developers who are actively interested in testing.

In the wizard in Fig. 2 for RQ5, the default slider position to estimate between production and test effort was set to 50 percent. This could be a reason for why we received an estimation of 51 percent:49 percent. To mitigate this, WatchDog users had to move the slider before they were allowed to progress the wizard, forcing them to think about their own distribution.

The Hawthorne effect [89] poses a similar threat: 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 4.1, it was necessary to give users an incentive to install WatchDog. Without the preview functionality, we would likely not have had any users. To measure the potential impact of our immediate IDE statistics (see Fig. 3), we tracked how often and how long developers consulted it via the `WatchDogView` interval. In total, only 192 of the 2,200 Eclipse developers opened the view in total 720 times in 422 of 39,855 possible sessions (1 percent), with a median open time of 2.4 minutes per user. This is similar with 58 times for 181 developers in Σ_{CN}. We believe that these measures demonstrate that developers did not constantly monitor their WatchDog recorded testing behavior, otherwise the numbers would be significantly higher. That users engage with reports about their behavior only for a short amount of time is not unique to our study: Meyer et al. found similar numbers when presenting developers with a report of their productivity [90]. Even the commercial RescueTime only had user engagement lengths of on average five seconds per day [91]. Our long observation period is another suitable countermeasure to the Hawthorne effect, as developers might change their behavior for a day, but unlikely for several months.

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

7.4 External Validity

Threats to external validity concern the generalizability of our results. While we observed 161 years of development worktime (collected in 14,266,683 intervals originating from 2,443 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 is 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 mainly very experienced programmers follow TDD to some extent in Section 5.4.

By capturing not only data from Eclipse, but also IntelliJ, Android Studio, and Visual Studio, we believe to have sufficiently excluded the threat that a certain behavior might be IDE-specific. While we have data from two programming languages (Java and C#), other programming language communities, especially non-object-oriented ones, might have different testing cultures and use other IDEs that might not facilitate testing in the same way the Eclipse, IntelliJ, and Visual Studio IDEs do. Hence, their results might deviate from the relatively mature and test-aware Java and C# communities.

Finally, the time we measure for an activity such as testing in the IDE does not equal the effort an organization has to invest in it overall. Arguments against this are that developer testing per hour is as expensive as development (since both are done by the same set of persons), and that time is typically the critical resource in software development [47]. An in-depth investigation with management data such as real project costs is necessary to validate this in practice. To exclude the risk of a different understanding of the word testing, we specifically asked developers about JUnit testing, i.e., automated, codified developer tests (see the description in Fig. 2).

8 Related Work

In this section, we first describe tools and plugins that are methodically similar to WatchDog, and then proceed with a description of related research.

8.1 Related Tools and Plugins

A number of tools have been developed to assess development activity at the sub-commit level. These tools include Hackystat [92], Syde [93], Spyware [94], CodingTracker [95], DFlow [96], the "Change-Oriented Programming Environment,"[8] the "Eclipse Usage Data Collector,"[9] QuantifiedDev,[10] Codealike,[11] and RescueTime.[12] However, none of these focused on time-related developer testing.

Hackystat with its Zorro extension was one of the first solutions that aimed at detecting TDD activities [97], [98], similar to the education-oriented TDD-Guide [99] and the prototype TestFirstGauge [100]. In contrast to WatchDog, Hackystat did not focus on the IDE, but offered a multitude of sensors, from bug trackers such as Bugzilla to build tool such as `ant`. One of Hackystat's challenges that we addressed with WatchDog was attracting a broader user base that allowed the recording and processing of their data.

8.2 Related Research

To investigate the presence or absence of tests, Kochar et al. mined 20,000 open-source projects and found that 62 percent contained unit tests [101]. LaToza et al. [102] surveyed 344 software engineers, testers and architects at Microsoft, with 79 percent of the respondents indicating that they use unit tests. Our findings indicate that only 35 percent 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.

In a study on GitHub using a repository-mining approach, Borle et al. found that a mere 3.7 percent of over 250,000 analyzed repositories could be classified to be using TDD [64]. This result strengthens our observed low TDD use in IDE sessions.

Pham et al. [103] 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 [83].

Zaidman et al. [4] and Marsavina et al. [53] 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, Zaidman and Marsavina found that writing

8. http://cope.eecs.oregonstate.edu
9. https://eclipse.org/epp/usagedata
10. https://www.youtube.com/watch?v=7QKWo5SulP8
11. https://codealike.com
12. https://rescuetime.com

test code is phased: after a longer period of production code development, developers switch to test code. Marinescu et al. [104] observed that test coverage usually remains constant, because already existing tests execute part of the newly added code. Feldt [105] 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 is related to the augmentation of test suites. Additionally, Athanasiou et al. investigated the quality of developer tests, noting that completeness, effectiveness, and maintainability of tests tend to vary among the observed projects [106].

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], [53], [101], [105] or (2) purely by means of a survey or interview [83], [102], [103], [107]. 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 alone.

9 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 developers' working environment to extract developer activity. We complemented and contrasted these objective observations with surveys of said developers. We found that automated developer testing (at least in the IDE) is not as popular as often assumed, that developers do not test as much as they believe they do, and that TDD is not a popular development paradigm. We called the concept of loosely steering software development with the help of testing *Test-Guided Development*.

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.
4) A generalized investigation of developer testing patterns across four IDEs in two programming languages.

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. Our work suggests that different tools and languages that are conceptually similar might not impact the practice as much as individuals often think, since we found few differences between data originating from them.

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 only partially matched 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. On a positive note, we also found that even relatively small samples from one population group might generalize well.

Acknowledgments

We owe our biggest gratitude to the hundreds of WatchDog users. Moreover, we thank Maryi Arciniegas-Mendez, Alan Richardson, Nepomuk Seiler, Shane McIntosh, Michaela Greiler, Diana Kupfer, Lars Vogel, Anja Reuter, Marcel Bruch, Ian Bull, Katrin Kehrbusch, Maaike Beliën, and the anonymous reviewers. We thank Andreas Bauer for help with the WatchDog transformer. This work was funded by the Dutch Science Foundation (NWO), project TestRoots (016.133.324) and the German Federal Ministry of Education and Research (BMBF) within the Software Campus project *Eko*, grant no. 01IS12054, and by the DFG as part of CRC 1119 CROSSING.

References

[1] P. Runeson, "A survey of unit testing practices," *IEEE Softw.*, vol. 23, no. 4, pp. 22–29, Jul./Aug. 2006. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MS.2006.91

[2] A. Begel and T. Zimmermann, "Analyze this! 145 questions for data scientists in software engineering," in *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 12–13. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568233

[3] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proc. Symp. Found. Softw. Eng.*, 2012, pp. 33:1–33:11.

[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 Softw. Eng.*, vol. 16, no. 3, pp. 325–364, 2011. [Online]. Available: http://dx.doi.org/10.1007/s10664–010-9143-7

[5] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *Proc. Int. Conf. Softw. Eng. Workshop Future Softw. Eng.*, 2007, pp. 85–103. [Online]. Available: http://doi.acm.org/10.1145/1253532.1254712

[6] M. V. Mäntylä, J. Itkonen, and J. Iivonen, "Who tested my software? testing as an organizationally cross-cutting activity," *Softw. Quality J.*, vol. 20, no. 1, pp. 145–172, 2012.

[7] F. Brooks, *The Mythical Man-Month*. Reading, MA, USA: Addison-Wesley, 1975.

[8] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their IDEs," in *Proc. 10th Joint Meet. Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2015, pp. 179–190.

[9] M. Beller, G. Gousios, and A. Zaidman, "How (much) do developers test?" in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 559–562.

[10] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen, "Mining software repositories to study co-evolution of production & test code," in *Proc. 1st Int. Conf. Softw. Testing Verification Validation*, 2008, pp. 220–229.

[11] M. Beller, I. Levaja, A. Panichella, G. Gousios, and A. Zaidman, "How to catch 'em all: WatchDog, a family of IDE plug-ins to assess testing," in *Proc. 3rd Int. Workshop Softw. Eng. Res. Ind. Practice*, 2016, pp. 53–56.

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

[13] S. Proksch, S. Nadi, S. Amann, and M. Mezini, "Enriching IN-IDE process information with fine-grained source code history," in *Proc. 24th Int. Conf. Softw. Anal. Evolution Reengineering*, 2017, pp. 250–260.

[14] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Reading, MA, USA: Addison-Wesley, 2007.

[15] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub," in *Proc. 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 356–367.

[16] R. L. Glass, R. Collard, A. Bertolino, J. Bach, and C. Kaner, "Software testing and industry needs," *IEEE Softw.*, vol. 23, no. 4, pp. 55–57, Jul./Aug. 2006. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MS.2006.113

[17] A. Bertolino, "The (Im)maturity level of software testing," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 5, pp. 1–4, Sep. 2004. [Online]. Available: http://doi.acm.org/10.1145/1022494.1022540

[18] J. Rooksby, M. Rouncefield, and I. Sommerville, "Testing in the wild: The social and organisational dimensions of real world practice," *Comput. Supported Cooperative Work*, vol. 18, no. 5/6, pp. 559–580, Dec. 2009. [Online]. Available: http://dx.doi.org/10.1007/s10606-009-9098-7

[19] TestRoots WatchDog. [Online]. Available: https://github.com/TestRoots/watchdog, Accessed on: Jun. 6, 2017.

[20] P. Muntean, C. Eckert, and A. Ibing, "Context-sensitive detection of information exposure bugs with symbolic execution," in *Proc. Int. Workshop Innovative Softw. Develop. Methodologies Practices*, 2014, pp. 84–93.

[21] G. Inc. and the Open Handset Alliance, "Download Android studio and SDK tools," Accessed on: May 31, 2017.

[22] ReSharper Plugin Gallery. [Online]. Available: https://www.jetbrains.com/resharper/, Accessed on: Jun. 6, 2017.

[23] S. Amann, S. Proksch, S. Nadi, and M. Mezini, "A study of visual studio usage in practice," in *Proc. 23rd IEEE Int. Conf. Softw. Anal. Evolution Reengineering*, 2016, pp. 124–134.

[24] M. Beller, N. Spruit, and A. Zaidman, "How developers debug," *PeerJ Preprints*, vol. 5, 2017, Art. no. e2743v1.

[25] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[26] Apache Maven Conventions. [Online]. Available: http://maven.apache.org/guides/getting-started, Accessed on: Jun. 6, 2017.

[27] J. C. Munson and S. G. Elbaum, "Code churn: A measure for estimating the impact of code change," in *Proc. Int. Conf. Softw. Maintenance*, 1998, Art. no. 24. [Online]. Available: http://dlib.computer.org/conferen/icsm/8779/pdf/87790024.pdf

[28] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 643–653.

[29] K. Beck, *Test Driven Development—by Example*. Reading, MA, USA: Addison Wesley, 2003.

[30] H. Munir, K. Wnuk, K. Petersen, and M. Moayyed, "An experimental evaluation of test driven development vs. test-last development with industry professionals," in *Proc. Int. Conf. Eval. Assessment Softw. Eng.*, 2014, pp. 50:1–50:10. [Online]. Available: http://doi.acm.org/10.1145/2601248.2601267

[31] Y. Rafique and V. B. Misic, "The effects of test-driven development on external quality and productivity: A meta-analysis," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 835–856, Jun. 2013.

[32] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual*, 2nd ed. London, U.K.: Pearson Higher Education, 2004.

[33] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Englewood Cliffs, NJ, USA: Prentice Hall, 2007.

[34] 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.

[35] J. L. Devore and N. Farnum, *Applied Statistics for Engineers and Scientists*. Pacific Grove, CA, USA: Duxbury, 1999.

[36] W. G. Hopkins, *A New View of Statistics*. 1997. [Online]. Available: http://sportsci.org/resource/stats/, Accessed on: Mar. 27, 2017.

[37] S. Amann, S. Proksch, and S. Nadi, "FeedBaG: An interaction tracker for visual studio," in *Proc. 24th Int. Conf. Program Comprehension*, 2016, pp. 1–3.

[38] Let's Develop With TestRoots' WatchDog. [Online]. Available: http://youtu.be/-06ymo7dSHk, Accessed on: Jun. 7, 2017.

[39] Eclipse Marketplace: WatchDog Plugin. [Online]. Available: https://marketplace.eclipse.org/content/testroots-watchdog, Accessed on: Jun. 6, 2017.

[40] IntelliJ Marketplace: WatchDog Plugin. [Online]. Available: https://plugins.jetbrains.com/plugin/7828-watchdog, Accessed on: Jun. 6, 2017.

[41] ReSharper Plugin Gallery: FeedBaG++ Plugin. [Online]. Available: https://resharper-plugins.jetbrains.com/packages/KaVE.Project/, Accessed on: Jun. 6, 2017.

[42] Code Trails Marketplace: WatchDog Plugin. [Online]. Available: http://www.codetrails.com/blog/test-analytics-testroots-watchdog, Accessed on: Jun. 6, 2017.

[43] KAVE Datasets: Interaction Data, Mar. 1, 2017. [Online]. Available: http://www.kave.cc/datasets/, Accessed on: Jun. 6, 2017.

[44] G. Rothermel and S. Elbaum, "Putting your best tests forward," *IEEE Softw.*, vol. 20, no. 5, pp. 74–77, Sep./Oct. 2003.

[45] J. Bell, G. Kaiser, E. Melski, and M. Datattreya, "Efficient dependency detection for safe Java test acceleration," in *Proc. 10th Joint Meet. Found. Softw. Eng.*, 2015, pp. 770–781.

[46] F. Palomba and A. Zaidman, "Does refactoring of test smells induce fixing flaky tests?" in *Proc. Int. Conf. Softw. Maintenance Evolution*, 2017, pp. 1–12.

[47] T. Halkjelsvik and M. Jørgensen, "From origami to software development: A review of studies on judgment-based predictions of performance time," *Psychological Bulletin*, vol. 138, no. 2, 2012, Art. no. 238.

[48] C. Vassallo, et al., "A tale of CI build failures: An open source and a financial organization perspective," in *Proc. Int. Conf. Softw. Maintenance Evolution*, 2017, pp. 183–193.

[49] A. Patterson, M. Kölling, and J. Rosenberg, "Introducing unit testing with BlueJ," *ACM SIGCSE Bulletin*, vol. 35, no. 3, pp. 11–15, Jun. 2003. [Online]. Available: http://doi.acm.org/10.1145/961290.961518

[50] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *Proc. Work. Conf. Mining Softw. Repositories*, 2014, pp. 202–211.

[51] E. Derby, D. Larsen, and K. Schwaber, *Agile Retrospectives: Making Good Teams Great*. Raleigh, NC, USA: Pragmatic Bookshelf, 2006.

[52] V. Hurdugaci and A. Zaidman, "Aiding software developers to maintain developer tests," in *Proc. Eur. Conf. Softw. Maintenance Reengineering*, 2012, pp. 11–20.

[53] C. Marsavina, D. Romano, and A. Zaidman, "Studying fine-grained co-evolution patterns of production and test code," in *Proc. Int. Work. Conf. Source Code Anal. Manipulation*, 2014, pp. 195–204. [Online]. Available: http://dx.doi.org/10.1109/SCAM.2014.28

[54] S. Romano, D. Fucci, G. Scanniello, B. Turhan, and N. Juristo, "Findings from a multi-method study on test-driven development," *Inf. Softw. Technol.*, vol. 89, pp. 64–77, 2017.

[55] M. Gligoric, S. Negara, O. Legunsen, and D. Marinov, "An empirical evaluation and comparison of manual and automated test selection," in *Proc. 29th ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2014, pp. 361–372.

[56] A. Bookstein, "Informetric distributions, part I: Unified overview," *J. Amer. Soc. Inf. Sci.*, vol. 41, no. 5, 1990, Art. no. 368.

[57] R. S. Pressman, *Software Engineering: A Practitioner's Approach*. Basingstoke, U.K.: Palgrave Macmillan, 2005.

[58] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The art of testing less without sacrificing quality," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 483–493.

[59] J. Anderson, S. Salem, and H. Do, "Improving the effectiveness of test suite through mining historical data," in *Proc. 11th Work. Conf. Mining Softw. Repositories*, 2014, pp. 142–151. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597084

[60] 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 *Proc. Work. Conf. Mining Softw. Repositories*, 2014, pp. 102–111.

[61] Inozemtseva, Laura Michelle McLean, "Data science for software maintenance," Ph.D. dissertation, 2017. [Online]. Available: http://hdl.handle.net/10012/11753

[62] M. Beller, G. Gousios, and A. Zaidman, "TravisTorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration," in *Proc. 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 447–450.

[63] D. Janzen and H. Saiedian, "Test-driven development: Concepts, taxonomy, and future direction," *IEEE Comput.*, vol. 38, no. 9, pp. 43–50, Sep. 2005.

[64] A. Hindle, N. Borle, and M. Feghhi, "Analysis of test driven development on sentiment and coding activities in GitHub repositories." 2016. [Online]. Available: https://doi.org/10.7287/peerj.preprints.1920v2

[65] S. Romano, D. D. Fucci, G. Scanniello, B. Turhan, and N. Juristo, "Results from an ethnographically-informed study in the context of test-driven development," in *Proc. 20th Int. Conf. Eval. Assessment Softw. Eng.*, 2016, Art. no. 10.

[66] S. Romano, D. Fucci, G. Scanniello, B. Turhan, and N. Juristo, "Results from an ethnographically-informed study in the context of test driven development," in *Proc. 20th Int. Conf. Eval. Assessment Softw. Eng.*, 2016, pp. 10:1–10:10. [Online]. Available: http://doi.acm.org/10.1145/2915970.2915996

[67] J. W. Wilkerson, J. F. Nunamaker Jr, and R. Mercer, "Comparing the defect reduction benefits of code inspection and test-driven development," *IEEE Trans. Softw. Eng.*, vol. 38, no. 3, pp. 547–560, May/Jun. 2012.

[68] A. Oram and G. Wilson, *Making Software: What Really Works, and Why we Believe It*. Sebastopol, CA, USA: O'Reilly Media, 2010.

[69] S. Kollanus, "Test-driven development-still a promising approach?" in *Proc. 7th Int. Conf. Quality Inf. Commun. Technol.*, 2010, pp. 403–408.

[70] H. Erdogmus, M. Morisio, and M. Torchiano, "On the effectiveness of test-first approach to programming," *IEEE Trans. Soft. Eng.*, vol. 31, no. 3, pp. 226–237, 2005.

[71] L. Williams, E. M. Maximilien, and M. Vouk, "Test-driven development as a defect-reduction practice," in *Proc. 14th Int. Symp. Softw. Rel. Eng.*, 2003, pp. 34–45.

[72] N. Nagappan, E. M. Maximilien, T. Bhat, and L. Williams, "Realizing quality improvement through test driven development: Results and experiences of four industrial teams," *Empirical Softw. Eng.*, vol. 13, no. 3, pp. 289–302, 2008.

[73] M. M. Müller and F. Padberg, "About the return on investment of test-driven development," in *Proc. 5th Int. Workshop Econ.-Driven Softw. Eng. Res.*, 2003, Art. no. 26.

[74] G. Canfora, A. Cimitile, F. Garcia, M. Piattini, and C. A. Visaggio, "Evaluating advantages of test driven development: A controlled experiment with professionals," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng.*, 2006, pp. 364–371.

[75] Joint Task Force on Computing Curricula, IEEE Computer Society, and Association for Computing Machinery, "Curriculum guidelines for undergraduate degree programs in software engineering." [Online]. Available: http://www.acm.org/binaries/content/assets/education/se2014.pdf, Accessed on: Jun. 6, 2017.

[76] D. S. Janzen and H. Saiedian, "Does test-driven development really improve software design quality?" *IEEE Softw.*, vol. 25, no. 2, pp. 77–84, Mar./Apr. 2008.

[77] D. H. Hansson, "TDD is dead. Long live testing." [Online]. Available: http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html, Accessed on: Jun. 6, 2017.

[78] D. H. Hansson, "Test-induced design damage." [Online]. Available: http://david.heinemeierhansson.com/2014/test-induced-design-damage.html, Accessed on: Jun. 6, 2017.

[79] D. H. Hansson, K. Beck, and M. Fowler, "Is TDD dead?" [Online]. Available: https://youtu.be/z9quxZsLcfo, Accessed on: Apr. 13, 2016.

[80] Android Studio Documentation: Test Your App. [Online]. Available: https://developer.android.com/studio/test/index.html, Accessed on: Jun. 12, 2017.

[81] T. Connolly and D. Dean, "Decomposed versus holistic estimates of effort required for software writing tasks," *Manage. Sci.*, vol. 43, no. 7, pp. 1029–1045, 1997.

[82] I. Ciupa, "Test studio: An environment for automatic test generation based on design by contracttm," Master's thesis, Chair of Software Engineering, Eidgenössische Technische Hochschule Zürich, Switzerland, 2004.

[83] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann, "Software developers' perceptions of productivity," in *Proc. Int. Symp. Found. Softw. Eng.*, 2014, pp. 19–29. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635892

[84] L. Hatton, "How accurately do engineers predict software maintenance tasks?" *IEEE Comput.*, vol. 40, no. 2, pp. 64–69, Feb. 2007.

[85] M. M. Roy, N. J. Christenfeld, and C. R. McKenzie, "Underestimating the duration of future events: Memory incorrectly used or memory bias?" *Psychological Bulletin*, vol. 131, no. 5, 2005, Art. no. 738.

[86] M. M. Roy, S. T. Mitten, and N. J. Christenfeld, "Correcting memory improves accuracy of predicted task duration," *J. Exp. Psychology: Appl.*, vol. 14, no. 3, 2008, Art. no. 266.

[87] M. Jørgensen and D. Sjøberg, "Generalization and theory-building in software engineering research," in *Proc. Int. Conf. Empirical Assessment Softw. Eng.*, 2004, pp. 29–36.

[88] D. I. Sjoberg, T. Dyba, and M. Jorgensen, "The future of empirical methods in software engineering research," in *Proc. Future Softw. Eng.*, 2007, pp. 358–378.

[89] J. G. Adair, "The Hawthorne effect: A reconsideration of the methodological artifact," *J. Appl. Psychology*, vol. 69, no. 2, pp. 334–345, 1984.

[90] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann, "Software developers' perceptions of productivity," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 19–29.

[91] E. I. Collins, A. L. Cox, J. Bird, and D. Harrison, "Social networking use and rescuetime: The issue of engagement," in *Proc. ACM Int. Joint Conf. Pervasive Ubiquitous Comput.: Adjunct Publication*, 2014, pp. 687–690.

[92] P. M. Johnson, et al., "Beyond the personal software process: Metrics collection and analysis for the differently disciplined," in *Proc. 25th Int. Conf. Softw. Eng.*, 2003, pp. 641–646.

[93] L. Hattori and M. Lanza, "Syde: A tool for collaborative software development," in *Proc. Int. Conf. Softw. Eng.*, 2010, pp. 235–238.

[94] R. Robbes and M. Lanza, "SpyWare: A change-aware development toolset," in *Proc. Int. Conf. Softw. Eng.*, 2008, pp. 847–850.

[95] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A comparative study of manual and automated refactorings," in *Proc. 27th Eur. Conf. Object-Oriented Program.*, 2013, pp. 552–576.

[96] R. Minelli, A. Mocci, M. Lanza, and L. Baracchi, "Visualizing developer interactions," in *Proc. Working Conf. Softw. Vis.*, 2014, pp. 147–156.

[97] H. Kou, P. M. Johnson, and H. Erdogmus, "Operational definition and automated inference of test-driven development with Zorro," *Automated Softw. Eng.*, vol. 17, no. 1, pp. 57–85, 2010.

[98] P. M. Johnson, "Searching under the streetlight for useful software analytics," *IEEE Softw.*, vol. 30, no. 4, pp. 57–63, Jul./Aug. 2013.

[99] O. Mishali, Y. Dubinsky, and S. Katz, "The TDD-Guide training and guidance tool for test-driven development," in *Agile Processes in Software Engineering and Extreme Programming*. Berlin, Germany: Springer, 2008, pp. 63–72.

[100] Y. Wang and H. Erdogmus, "The role of process measurement in test-driven development," in *Proc. 4th Conf. Extreme Program. Agile Methods*, 2004, pp. 32–42.

[101] P. Kochhar, T. Bissyande, D. Lo, and L. Jiang, "An empirical study of adoption of software testing in open source projects," in *Proc. Int. Conf. Quality Softw.*, 2013, pp. 103–112.

[102] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," in *Proc. Int. Conf. Softw. Eng.*, 2006, pp. 492–501.

[103] R. Pham, S. Kiesling, O. Liskin, L. Singer, and K. Schneider, "Enablers, inhibitors, and perceptions of testing in novice software teams," in *Proc. Int. Symp. Found. Softw. Eng.*, 2014, pp. 30–40.

[104] P. D. Marinescu, P. Hosek, and C. Cadar, "Covrig: A framework for the analysis of code, test, and coverage evolution in real software," in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 93–104.

[105] R. Feldt, "Do system test cases grow old?" in *Proc. Int. Conf. Softw. Testing Verification Validation*, 2014, pp. 343–352.

[106] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, "Test code quality and its relation to issue handling performance," *IEEE Trans. Softw. Eng.*, vol. 40, no. 11, pp. 1100–1125, Nov. 2014. [Online]. Available: http://dx.doi.org/10.1109/TSE.2014.2342227
[107] M. Greiler, A. van Deursen, and M. Storey, "Test confessions: A study of testing practices for plug-in systems," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 244–254.

Moritz Beller received the MSc degree with distinction from the Technical University of Munich, Germany. He is working toward the PhD degree at the Delft University of Technology, The Netherlands. His primary research domain is evaluating and improving the feedback developers receive from dynamic and static analysis using empirical methods. He is the main author of TravisTorrent, which provides free and open Continuous Integration analytics. He is a member of the IEEE. More on https://www.inventitech.com.

Georgios Gousios received the MSc degree from the University of Manchester and the PhD degree from Athens University of Economics and Business, both with distinction. He is an assistant professor with the Delft University of Technology, The Netherlands. His research interests include fields of distributed software development processes, software analytics, software testing, research infrastructures, and mining software repositories. He is the main author of the GHTorrent data collection and curation framework, the Alitheia Core repository mining platform and several open source tools. He is a member of the IEEE.

Annibale Panichella is a research associate in the Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg. His research interests include security testing, evolutionary testing, search-based software engineering, textual analysis, and empirical software engineering. He serves and has served as program committee member of various international conference (e.g., ICSE, GECCO, ICST and ICPC) and as reviewer for various international journals (e.g., the *IEEE Transactions on Software Engineering*, the *ACM Transactions on Software Engineering and Methodology*, the *IEEE Transactions on Evolutionary Computation*, the *Empirical Software Engineering*, the *Software Testing, Verification & Reliability*) in the fields of software engineering and evolutionary computation. He is a member of the IEEE.

Sebastian Proksch is working toward the doctoral degree at TU Darmstadt in the group of Prof. Mira Mezini. His research is focused on the structured development of recommender systems in software engineering and includes work on static analyses, mining software repositories, and human factors in software engineering. The work is driven by the idea to combine different sources of input data to improve the quality of recommender systems and their evaluation. He is a member of the IEEE.

Sven Amann is working toward the doctoral degree at TU Darmstadt, Germany. His primary research domain is API-misuse detection using static analyses and machine-learning techniques applied to examples mined from large code repositores and code search engines. He is founder and project lead of the MUBench benchmark suite. He is a member of the IEEE. More on http://sven-amann.de.

Andy Zaidman received the MSc and PhD degrees in computer science from the University of Antwerp, Belgium, in 2002 and 2006, respectively. He is an associate professor with the Delft University of Technology, The Netherlands. His main research interests include software evolution, program comprehension, mining software repositories, and software testing. He is an active member of the research community and involved in the organization of numerous conferences (WCRE'08, WCRE'09, VISSOFT'14 and MSR'18). In 2013, he was the laureate of a prestigious Vidi career grant from the Dutch science foundation NWO. He is a member of the IEEE.