How (Much) Do Developers Test?

Moritz Beller, Georgios Gousios, Andy Zaidman
Delft University of Technology,
The Netherlands
{m.m.beller, g.gousios, a.e.zaidman}@tudelft.nl

***Abstract*—What do we know about software testing in the real world? It seems we know from Fred Brooks' seminal work "The Mythical Man-Month" that 50% of project effort is spent on testing. However, due to the enormous advances in software engineering in the past 40 years, the question stands: Is this observation still true? In fact, was it ever true? The vision for our research is to settle the discussion about Brooks' estimation once and for all: How much do developers test? Does developers' estimation on how much they test match reality? How frequently do they execute their tests, and is there a relationship between test runtime and execution frequency? What are the typical reactions to failing tests? Do developers solve actual defects in the production code, or do they merely relax their test assertions? Emerging results from 40 software engineering students show that students overestimate their testing time threefold, and 50% of them test as little as 4% of their time, or less. Having proven the scalability of our infrastructure, we are now extending our case study with professional software engineers from open-source and industrial organizations.**

I. Introduction

Understanding how developers test and how to better support them in practice is crucial for the design of next-generation Integrated Development Environments (*IDEs*) and testing tools. Important questions, that need to be answered to increase our understanding, are:

1) How much time is spent on engineering test code versus production code? Do developers' estimation on how much they test match reality?
2) How frequently do developers execute their tests? Do they do it after each change to production code, or only a few times a day?
3) How long does a test run executed in the IDE take? Is there a relation between the frequency and the length of execution?
4) What are the typical reactions to failing tests? Do developers solve actual defects in the production code, or do they merely relax their test assertions?

In his seminal work on the mythical man-month from 1975, Brooks estimates that 50% of the development time of a software product is dedicated to testing [1]. In the 40 years since, software engineering has changed dramatically. New programming languages, intelligent IDEs, a more agile way of working and test-driven development (*TDD*) are but a few of these advances. In general, practitioners and researchers have gained a broader understanding of the importance and benefits of software testing [2], expressed in its wide adoption in practice [2], [3].

Despite these fundamental changes, however, there is a surprising absence of research that follows up on Brooks' estimation and examines how much time developers devote to testing and production code. Even in recent literature, Brooks' rough 50% estimate is still widely referenced [4], [5]. It comes as no surprise then that the question of how much time needs to be spent on testing is one of the grand research challenges in empirical software engineering [6].

As developer testing, we understand any activity the developer undertakes in his IDE related to testing the program [7]. This usually consists of writing and executing unit tests, but is widely complemented by integration or system testing [8]. On a project level, developer testing is often complemented by manual testing, dedicated test teams and automated test generation. In contrast to these quality assurance methods, testing in the IDE cannot be quantified with a traditional time measurement approach, as it is intertwined with developing production code, especially when using TDD.

By observing the fine-grained steps in which developers construct software in their IDEs, we are able to provide deep insights into the aforementioned questions. Our approach contrasts the traditional repository mining, which focuses on the final result of fine-grained developer activities in the IDE, but does not provide accurate timing information about them.

In this paper, we present a novel approach to tackle our empirical questions on developer testing in a large-scale case study and report its first emerging results. To this end, we have created the WatchDog project.[1] While the data collection from developers across several open-source and commercial organizations is ongoing, student data from a software engineering course at TU Delft gives interesting first results: It shows that students use on average only 9% of their working time in the IDE for engineering tests. Moreover, they test three times less in reality than they think they do. In our ongoing work, we are complementing this study with a study on professional developers from commercial and open-source projects, and diving deeper into the reactions of individual developers to failing test cases.

II. Methods

To study our research questions, we could watch over the shoulders of developers and manually take notes on how they develop their software. However, when we want to perform the study on a larger scale, we need an automated way to reliably

[1]http://www.testroots.org/testroots_watchdog.html

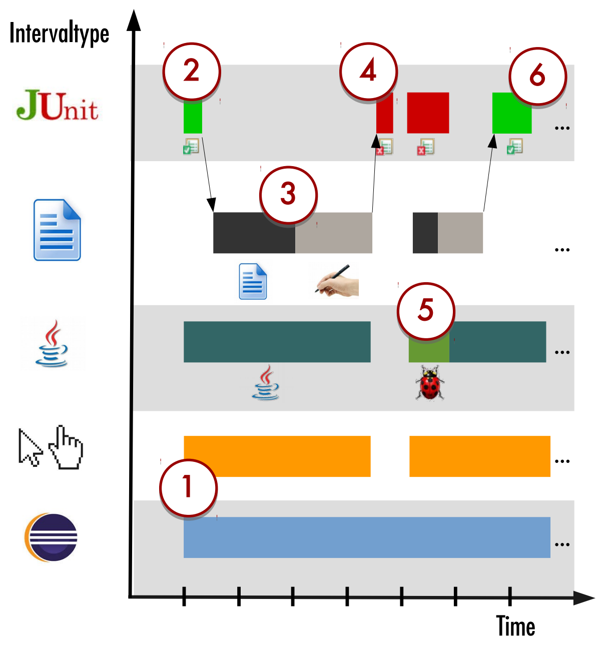

Fig. 1. Exemplary data showing an intuitive visualization of intervals.

collect usage data related to developer testing. For this reason, we have developed WatchDog, which we detail in this section. We then explain how we attracted developers to using it.

A. *WatchDog Infrastructure*

Starting with an initial prototype in 2012, we evolved WatchDog into an open-source and production-ready software solution[2] with a client-server architecture, which is designed to scale up to thousands of simultaneous users.

1) WatchDog Client: We implemented WatchDog as an Eclipse plugin, because the Eclipse Java Development Tools edition (JDT) is one of the most widely used IDEs for Java programming [9]. Thanks to its integrated JUnit support, the Eclipse JDT facilitates developer testing.

WatchDog instruments the Eclipse JDT environment and registers listeners for internal UI events related to programming behavior and test executions. We group coherent events as *intervals* which comprise its interval type, start and end time. Thanks to this abstraction, we can closely follow the typical workflow of a developer without observing hundreds of events per minute. Every time a developer reads, modifies or executes a test or production code class, WatchDog creates a new interval and enriches it with interval type-specific data.

Having installed WatchDog, developers register a user and the workspaces in which it should be active. The registration includes a survey which asks for an estimate of how much time they spend on testing, and how experienced a programmer they are.

Figure 1 shows an exemplary workflow: a developer starts Eclipse and WatchDog creates `EclipseOpen` and `UserActive` intervals (1). After that, the user executes the unit tests of the class he needs to change, triggering the creation of a `JUnitInterval` that is enriched with the test result "Passed" (2). Having browsed the source code of the file (3) to understand which parts need to change (triggering a `ReadingInterval`), the developer then performs the necessary changes. In such a `TypingInterval`, WatchDog stores categorical information on the opened file, like its source lines of code before and after editing, and whether it is production or test code. To rate a file as a test, we look for imports of Java test frameworks and their annotations (Junit, Mockito, or PowerMock), and "Test" in the file path. This way, we can correctly identify all tests that employ standard Java testing frameworks as test runners for their unit, integration, or system tests, and even test-related utility classes. A re-execution of the unit test shows it fails after the edit (4). The developer steps through the test with the debugger (5) and fixes the error. The final re-execution of the test (6) assures him of his success. All user intervals are backed by a timeout, so that we do not record intervals when the user is inactive. Intervals are locally cached, allowing offline work, and automatically sent as a Json stream.

2) WatchDog Server: The WatchDog server accepts this Json data via a REST API. After sanity checking, the intervals are stored in a NoSQL database. This infrastructure allows high extensibility up to thousands of clients and easy maintainability in case of changes in the client. Automated ping-services monitor the health of our web API, so we can immediately react if a problem occurs.

B. *Acquisition of WatchDog Users*

Our initial hope was that it is sufficient to advertise WatchDog through its website, Twitter, and Facebook. However, despite enormous efforts,[3] the social media campaign resulted in only two new users.

We identified three reasons for this: 1) Privacy-concerns. 2) A lack of an incentive to use WatchDog, as it does not give developers immediate insights. 3) An increasing popularity of other IDEs like IntelliJ IDEA.

As a result, we devised a new strategy on how to approach the Eclipse community as a whole and interested companies individually (mitigating reason 3). Furthermore, we implemented a new feature that displays basic statistics from the last development hour in the Eclipse IDE (2). While this increases the theoretical possibility that developers change their behavior, we could never fully exclude the Hawthorne effect [10], namely that developers change their behavior because they know they are being watched. At the same time, we are developing solutions to make it attractive for companies to deploy WatchDog (1), like user-agnostic data collection and project reports aimed at the management level.[4]

To calibrate our measurements, we asked 165 students participating in the 10-week course "Software Engineering Methods" (*SEM*) at TU Delft to install WatchDog. In the course, student teams of five to six implement a small game in

[2]https://github.com/TestRoots/watchdog

[3]E.g., we were retweeted in total over 60 times, http://goo.gl/AqE2zU

[4]http://files.figshare.com/1800935/sample_watchdog_report.pdf

Java. The course's weekly assignments focus on implementing new features while the students learn and apply sound software engineering principles. Students following SEM are familiar with testing and have to deliver a well-tested final product with 75% statement coverage (C0).

III. Preliminary Results

In this section, we present the initial analysis of 388,669 intervals we received from 40 SEM students working on 24 different projects over the course of 10 weeks. In total, we observed 4.2 work years of development in the IDE, assuming the OECD average number of working hours in the Netherlands in 2013.[5]

Table I presents our preliminary results on variables that have been aggregated per project. From top to bottom, the table has four sections: overall project statistics, general user behavior in the IDE, user behavior with respect to testing, and statistics of test executions. Its mini-histograms provide an intuition of their distribution. Q_x denotes the x-quantile.

In general, we see that users spent on average 4 hours of work time per day, in which, on most projects, they read code rather than write it (70% reading time). Both variables "reading" and "writing time" are normally distributed across projects (a Shapiro-Wilks test accepts the null hypothesis at $p < 0.05$ in both cases), with relatively small deviations ($\sigma = 15.09$); this indicates a fairly consistent developer behavior in the activities reading and writing code. All findings we report in the following are initial observations, though, and we need to validate them with a larger set of professional developers before they should be used to draw conclusions.

1) How much time is spent on engineering test code versus production code? Do developers' estimation on how much they test match reality?

The core question of this early research effort is how much time software engineers spend on testing versus developing production code in their IDE. Table I shows that SEM students spend on average 91% of their time on production code and only 9% on test code (both reading and writing). The low effort on testing is surprising, as SEM requires 75% C0 coverage. It seems that reaching 75% coverage is possible even when working less than a tenth of the development time on tests. In fact, 50% of the students spent more than 96% of their time on production code. This result strengthens research that questions the value of coverage-based criteria [11].

Almost all students spent less than 50% of their time on testing, most of them considerably less. Brooks' 50%-hypothesis is therefore not accurate for the observed SEM projects. This raises the question whether his hypothesis holds for real-world projects, where additional testing forms might be practiced.

Comparing the actual to the estimated test-production time distribution, we see that the 5% and 95% quantiles are relatively similar (56% to 66%, 90% to 100%) – supporting the intuition that developers who do not test at all, or who test a lot, can identify their exceptional behavior easily. However, more alarmingly, the vast majority of students consistently overestimated the time they spend on testing threefold (9% instead of an estimated 27%). We urgently need to investigate this disturbing finding with open-source and industrial developers, as it could be one of the explaining factors for the large amount of bugs in today's software.

2) How frequently do developers execute their tests? Do they do it after each change to production code, or only a few times a day?

Students executed their tests on quite regular intervals (4.5/day). Together with the average recording time per day, this means students executed their tests roughly every 50 minutes on average. Judging from our own development experience, we had expected more frequent test executions. Explanations could be that the undergraduate students are relatively new to object-orientation and the complex Eclipse IDE, which slows down development and hence decreases the number of necessary test executions. More programming practice could help here. Students might also underestimate the value of their tests. A lesson for teachers could be to make students more aware of the benefits of frequent testing.

3) How long does a test run executed in the IDE take? Is there a relation between the frequency and the length of execution?

Because the SEM projects are developed from scratch within 10 weeks, we expect relatively short test execution times. The median test duration of 1.2 seconds strengthens our expectation. At the same time, there is no significant correlation between the test duration and the number of test executions. A reason could be that the difference between short (0.0s) and long test executions (13.5s) is relatively unimportant in practice: Both give almost immediate feedback to the programmer. We hypothesize this might change for projects with more evolved tests that take longer to run.

4) What are the typical reactions to failing tests? Do developers solve actual defects in the production code, or do they merely relax their test assertions?

Most students experienced one failing test case per day, substantial outliers were rare. Together with the short working time per day (median 2.5 hours), this poses the question whether students mostly made small, incremental improvements to their software.

We can visualize a developer's test and development behavior similarly to figure 1 to answer this question. From our data, we can see whether developers try to debug into the failing test case (as in our example in section II-A1), how often they re-execute the offending test case, and whether they change the test itself or related production code. We have produced such graphs from the student data and expect to gain deeper insights when we examine them in comparison to a broader population of professional software engineers.

We can also quantify the changes in `TypingIntervals` in terms of how much code they modify. However, to be able

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

TABLE I
DESCRIPTIVE STATISTICS FOR ALL SEM PROJECTS (EACH VARIABLE AGGREGATED PER PROJECT)

Variable	Unit	$Q_{0.05}$	Mean	Median	$Q_{0.95}$	Histogram
Recorded intervals	count	1874	9890	7176	23040	
Average recorded time	hours / day	0.0	4.0	2.5	14.3	
Java reading time	%	48	70	70	91	
Java writing time	%	9	30	30	52	
Production code time (estimate)	%	56	73	75	90	
Production code time (actual)	%	66	91	96	100	
Test code time (estimate)	%	10	27	25	44	
Test code time (actual)	%	0	9	4	34	
Test executions	1 / day	0.0	4.5	3.0	17.1	
Test failures	1 / day	0.0	1.9	1.0	8.2	
Test duration	seconds	0.0	3.4	1.2	13.5	

to answer the question whether test assertions are relaxed, we would have to record the textual differences to see if "assert" statements are entirely removed or if their test conditions are made easier to reach (e.g. by asserting on a range of values instead of a precise value). Due to privacy-concerns, we do not record this information at present.

IV. Related Work

A number of tools have been developed to collect and present development activity at the sub-commit level. These tools include Syde [12], Spyware [13], and the "Change-Oriented Programming Environment"[6] from Oregon State University. Other related projects are the "Eclipse Usage Data Collector"[7] and QuantifiedDev.[8] However, none of these focuses on time-related developer testing.

V. Summary & Future Challenges

In our initial investigation with 40 students, we have seen that half of them spend 4% or less of their working time on testing. This clearly contradicts Fred Brooks' 40-year-old observation that approximately 50% of software engineering effort is spent on testing. Perhaps even more striking is that our 40 student participants were spending three times less effort on engineering tests than they thought they would.

Nevertheless, we understand that student data does not speak for professional software engineers, and we acknowledge a call to arms to carry out this research in a wide variety of open source and industrial settings. We are working on actively attracting developers to participate in our research using a variety of strategies, such as approaching developer conferences, targeting Eclipse-specific market places and publishing in developer magazines. If our initial findings are confirmed among professional developers, they might be an explanatory factor for the observed bug-proneness of current software, and change the way developers test their software.

In follow-up research, we aim to go a step further in that we also want to record how developers react to failing test cases. This will allow us to get insight into whether developers fix the actual defect, simply relax their test assertions or maybe delete the failing test altogether at a more fine-grained level than previously observed by Pinto et al. [14].

[6]http://cope.eecs.oregonstate.edu
[7]https://eclipse.org/epp/usagedata
[8]http://www.quantifieddev.org

Acknowledgment

We gratefully acknowledge The Netherlands Organisation for Scientific Research (NWO) for funding TestRoots. For their input on the project, we thank Alberto Bacchelli, Daniel Pelsmaeker, Thomas Kinnen, Fabian Streitel, Martin Pöhlmann, Anand Sawant, and the students from SEM.

References

[1] F. Brooks, *The mythical man-month*. Addison-Wesley, 1975.
[2] G. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.
[3] D. Janzen and H. Saiedian, "Does test-driven development really improve software design quality?" *Software, IEEE*, vol. 25, no. 2, 2008.
[4] B. Wang, F. Madani, X. Wang, L. Wang, and C. White, "Design structure matrix," in *Planning and Roadmapping Technological Innovations*. Springer, 2014.
[5] P. Runeson, C. Andersson, and M. Höst, "Test processes in software product evolutiona qualitative survey on the state of practice," *Journal of software maintenance and evolution*, vol. 15, no. 1, 2003.
[6] A. Begel and T. Zimmermann, "Analyze this! 145 questions for data scientists in software engineering." in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2014.
[7] G. Meszaros, *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, 2006.
[8] T. Xie, J. de Halleux, N. Tillmann, and W. Schulte, "Teaching and training developer-testing techniques and tool support," in *Proceeding of the International Conference on Object oriented programming systems languages and applications (OOPSLA Companion)*. ACM, 2010.
[9] P. Muntean, C. Eckert, and A. Ibing, "Context-sensitive detection of information exposure bugs with symbolic execution," in *Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices*. ACM, 2014.
[10] J. Adair, "The hawthorne effect: A reconsideration of the methodological artifact." *Journal of applied psychology*, vol. 69, no. 2, 1984.
[11] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, "Test code quality and its relation to issue handling performance," *IEEE Transactions on Software Engineering*, vol. 40, no. 11, 2014.
[12] L. Hattori and M. Lanza, "Syde: a tool for collaborative software development," in *Proceedings of the International Conference on Software Engineering - Volume 2 (ICSE)*, 2010.
[13] R. Robbes and M. Lanza, "Spyware: a change-aware development toolset," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2008.
[14] L. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2012.