Toward an Empirical Theory of Feedback-Driven Development

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

ABSTRACT

Software developers today crave for feedback, be it from their peers or even bots in the form of code review, static analysis tools like their compiler, or the local or remote execution of their tests in the Continuous Integration (CI) environment. With the advent of social coding sites like GitHub and tight integration of CI services like Travis CI, software development practices have fundamentally changed. Despite a highly changed software engineering landscape, however, we still lack a suitable description of an individual's contemporary software development practices, that is how an individual code contribution comes to be. Existing descriptions like the v-model are either too coarse-grained to describe an individual contributor's workflow, or only regard a sub-part of the development process like Test-Driven Development. In addition, most existing models are pre- rather than de-scriptive. By contrast, in our thesis, we perform a series of empirical studies to describe the individual constituents of Feedback-Driven Development (FDD) and then compile the evidence into an initial framework on how modern software development works. Our thesis culminates in the finding that feedback loops are the characterizing criterion of contemporary software development. Our model is flexible enough to accommodate a broad bandwidth of contemporary workflows, despite large variances in how projects use and configure parts of FDD.

ACM Reference Format:
Moritz Beller. 2018. Toward an Empirical Theory of Feedback-Driven Development. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3183440.3190332

1 RESEARCH PROBLEM AND MOTIVATION

Today, developers can receive feedback on a piece of code they created from a variety of sources: the compiler, automated static analysis tools (ASATs), the CI server, local or remote test runs, peers who perform a code review, if necessary, a debugging session that includes remote logging information, application telemetry, or interactive dashboards. Arguably, even end users can give feedback to the developers, possibly via an automated bug monitoring system like Eclipse's automated error reporting. The goal of all these different feedback mechanisms is to enable developers to immediately improve the quality of their software. It is thus evident that feedback stands at the center of today's software development practices and its center stand various quality assurance mechanisms.

However, so far, research has not come up with a unifying theory of FDD, partly because of the challenge to define a theory despite the widely varying ways in which modern software development works across projects. A theory would first give us a common vocabulary based on an empirically-grounded understanding of how developers work today. This, in turn, enables us to better educate students on relevant concepts, allow projects and developers to easily identify strengths and weaknesses in their own workflows, and do more articulate research on topics important to practitioners today.

2 BACKGROUND AND RELATED WORK

A plethora of breakdowns of software engineers' working processes are in existence today, from structured, general process decompositions like the V-model [17], over more flexible guidelines like the agile manifesto [18] to practically process-free software creation paradigms like the chaos model [20]. These models, however, tend to focus less on an individual developer's workflow, but more on the general workflow of an entire project. Thus, they are of little help in describing an individual contributor's workflow. Other, partly more recent inventions like Test-Driven Development (TDD) or its offspring Behavior-Driven Development provide recommendations closer to a single developer, but they often focus on a somewhat limited aspect of the software development process, for example how to drive development via testing, which leaves out other feedback-cycles such as code review or static analysis. Thus, they cannot provide us a model capturing a more holistic code creation process on the level of an individual contribution. A common denominator of all these models is that they are prescriptive rather than descriptive: they argue that a certain methodology *should be* applied instead of studying what *is being* applied.

In contrast to these pre facto models, we built up our model of Feedback-Driven Development post factum on empirical evidence. We performed empirical analyses on the constituents of today's software development workflow first and then compiled this empirical evidence into a model of Feedback-Driven Development. Our model is thus a contemporary mirror of the development practices of software developers.

3 INTENDED SOLUTION APPROACH

The Research Path Schema (RPS) is an analytical framework that allows Software Engineering researchers to clearly communicate the principal setup of their research. It also describes a way to theory building via different research paths [22]. It defines a substantive, conceptual, and methodological domain. Our instantiation of the RPS in Figure 1 starts from the substantive domain "Quality Assurance Methods in OSS and Commercial Software" ①, makes observations by means of a large-scale case study ② and derives a

ACM ISBN 978-1-4503-5663-3/18/05…$15.00
https://doi.org/10.1145/3183440.3190332

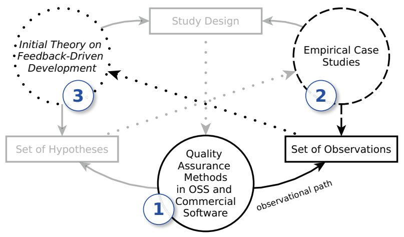
Study Design
Initial Theory on Feedback-Driven Development
Empirical Case Studies
Set of Hypotheses
Quality Assurance Methods in OSS and Commercial Software
Set of Observations
observational path
1
2
3

Figure 1: We followed the *observational path* in the RPS.

set of hypotheses on FDD that together form an initial theoretical framework ③.

To further increase their generalizability, we perform our studies in a large-scale fashion, typically on hundreds of projects or developers. This brings with it a number of complexities, from recruiting study participants over gathering large amounts of data to processing it. A point of criticism against large-scale analyses is that findings would sacrifice deep for broad understanding. However, single-project or small-scale analyses cannot uncover general phenomena and thus fail to quantify how widespread or impactful a certain issue is. Large-scale analyses help us single out individual problems from issues that plague Software Engineering as a craft.

The scale and nature of these studies ② requires a heavily tool-supported approach. We did this by not only mining existing software repositories like GIT in a traditional way, but also by creating our own new data sources: TRAVISTORRENT [7, 8], a TRAVIS CI "build log treasure trove" [2] and WATCHDOG [4–6, 9, 11], a family of IDE plugins that collects telemetry information from developer by instrumenting their IDEs. These techniques are scalable, robust, and updateable, causing minimal interference with the usual work habits of developers, thus increasing the realism in our studies. Compared to a (physical) onlooker, inducing the Hawthorne and trail effects [1, 14], our approach reduces these biases.

To analyze the data we gathered, we employ methods from the fields of data visualization, descriptive statistics, statistical hypothesis testing, and probability theory [19]. We enrich these methods with explanatory methods borrowed and adapted from the social sciences and known under the umbrella of Grounded Theory [16]. We employ a series of mixed-methods studies that combine several of the above techniques to answer a research question.

4 RESULTS AND CONTRIBUTIONS

Figure 2 sketches the FDD development workflow alongside its quality assurance methods typically found in today's software development projects. We take here the technical perspective of how a code contribution progresses from its initial inception ② steered by a project vision, an issue tracker task, or discussions ① to its final rejection or acceptance into the code base ⑥. A merged pull request, patch or otherwise integrated contribution equates an accepted code change in Figure 2 It is customary for contributions in the making to go through a cyclical review process until they reach a pre-defined acceptance criterion. Consequently, most projects explicitly allow reworking and perfecting contributions after their initial submission ⑦. The precise order of quality assurance checks in Figure 2 may deviate from project to project and even feedback cycle to feedback cycle. The Incremental Change process [21] complements FDD by describing what happens at the "Code Creation"

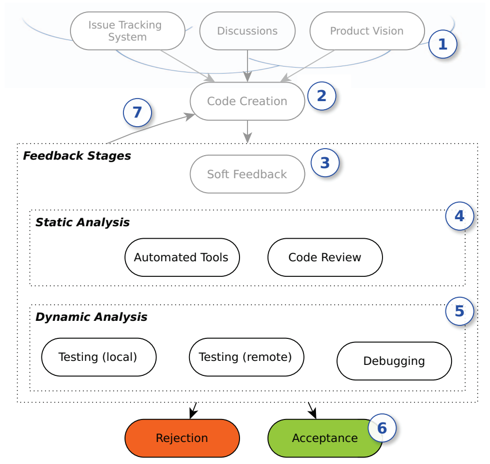

Figure 2: Studied (black) and related (gray) FDD concepts.

stage ②. Outside the scope of our technical investigation of FDD also falls the analysis of "soft feedback channels" ③ like STACK OVERFLOW. We divide the quality assurance methods we studied into Static Analysis ④, which examines program artifacts or their source code without executing them [23], and Dynamic Analysis ⑤, which relies on information gathered from their execution.

By performing a history analysis on the configuration files of ASATs, we found that most of the over 100 state-of-the-art projects on GITHUB only use one ASAT, that is typically only slightly customized and normally does not evolve throughout a project's life time [3]. To help developers unleash the potential of multiple ASATs, we created the tool UAV [15]. On the intersection between manual code review and ASATs, we discovered the last line effect [12, 13], the startling realization that the last line or statement in a micro-clone is more likely to contain a fault than any of the previous lines. We singled out psycho-cognitive reasons, most likely a working term memory overload, to be the prime cause and created an ASAT to integrate the detection of faulty micro-clones in an automated FDD loop. Overall, our studies suggest that integration of ASATs in the FDD cycle is lacking behind the integration of dynamic analysis.

We found that "Test-Guided Development" best describes most developers' local testing practices [4–6], as they do not follow strict processes like TDD rigorously and tend to overestimate their testing efforts in the IDE twofold. With our family of WATCHDOG plugins [9], we studied the testing habits of more than 2,400 Java and C# developers in four different IDEs over the course of 2.5 years. Results suggest that testing practices largely generalize across (imperative) programming languages and IDEs. Most local testing is immediate, with a much shorter feedback loop than running the entire test suite, which is usually offloaded to the CI. Accordingly, an analysis of our TRAVISTORRENT [8] "build log treasure trove" (TRAVIS CI) shows that remote testing is the central phase of CI, causing more CI build failures than all other reasons combined [7]. Normally, contributions thus need to be reworked if they failed the testing stage of FDD ⑦. Finally, debugging is a somewhat opaque topic to developers not nearly as automated and streamlined as the other FDD quality assurance techniques: many developers still employ crude printf techniques, for lack of better knowledge or tools [10].

REFERENCES

[1] John G Adair. 1984. The Hawthorne effect: A reconsideration of the methodological artifact. *Journal of applied psychology* 69, 2 (1984), 334.

[2] Moritz Beller. [n. d.]. Become a Travis CI Log Miner in the MSR Mining Challenge 2017! ([n. d.]). https://blog.travis-ci.com/2017-01-16-travis-ci-mining-challenge/.

[3] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 470–481.

[4] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. 2017. Developer Testing in The IDE: Patterns, Beliefs, And Behavior. *IEEE Transactions on Software Engineering* PP, 99 (2017), 1–1. https://doi.org/10.1109/TSE.2017.2776152 To appear. Pre-print: http://ieeexplore.ieee.org/document/8116886/.

[5] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, How, and Why Developers (Do Not) Test in Their IDEs. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 179–190.

[6] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2015. How (Much) Do Developers Test?. In *Proceedings of the 37th International Conference on Software Engineering (ICSE), NIER Track*. IEEE, 559–562.

[7] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*.

[8] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *Proceedings of the Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*.

[9] Moritz Beller, Igor Levaja, Annibale Panichella, Georgios Gousios, and Andy Zaidman. 2016. How to Catch 'Em All: WatchDog, a Family of IDE Plug-Ins to Assess Testing. In *3rd International Workshop on Software Engineering Research and Industrial Practice (SER&IP 2016)*. IEEE, 53–56.

[10] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. 2018. On the Dichotomy of Debugging Behavior Among Programmers. In *40th International Conference on Software Engineering, ICSE 2018, Gothenborg, Sweden*.

[11] Moritz Beller, Niels Spruit, and Andy Zaidman. 2017. How developers debug. *PeerJ Preprints* 5 (2017), e2743v1.

[12] Moritz Beller, Andy Zaidman, and Andrey Karpov. 2015. The Last Line Effect. In *23rd International Conference on Program Comprehension (ICPC)*. ACM, 240–243.

[13] Moritz Beller, Andy Zaidman, Andrey Karpov, and Rolf A. Zwaan. 2017. The last line effect explained. *Empirical Software Engineering* 22, 3 (01 Jun 2017), 1508–1536. https://doi.org/10.1007/s10664-016-9489-6

[14] David A Braunholtz, Sarah JL Edwards, and Richard J Lilford. 2001. Are randomized clinical trials good for us (in the short term)? Evidence for a "trial effect". *Journal of clinical epidemiology* 54, 3 (2001), 217–224.

[15] Tim Buckers, Clinton Cao, Michiel Doesburg, Boning Gong, Sunwei Wang, Moritz Beller, and Andy Zaidman. 2017. UAV: Warnings from multiple Automated Static Analysis Tools at a glance. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*. 472–476. https://doi.org/10.1109/SANER.2017.7884656

[16] Kathy Charmaz and Linda Liska Belgrave. 2007. *Grounded theory*. Wiley Online Library.

[17] Kevin Forsberg and Harold Mooz. 1992. The relationship of systems engineering to the project cycle. *Engineering Management Journal* 4, 3 (1992), 36–43.

[18] Martin Fowler and Jim Highsmith. 2001. The agile manifesto. *Software Development* 9, 8 (2001), 28–35.

[19] Will G Hopkins. 1997. *A new view of statistics*. http://sportsci.org/resource/stats/. Accessed March 27, 2017.

[20] LBS Raccoon. 1995. The chaos model and the chaos cycle. *ACM SIGSOFT Software Engineering Notes* 20, 1 (1995), 55–66.

[21] Vaclav Rajlich and Prashant Gosavi. 2004. Incremental change in object-oriented programming. *IEEE software* 21, 4 (2004), 62–69.

[22] Klaas-Jan Stol and Brian Fitzgerald. 2015. Theory-oriented software engineering. *Science of Computer Programming* 101 (2015), 79–98.

[23] BA Wichmann, AA Canning, DL Clutterbuck, LA Winsborrow, NJ Ward, and DWR Marsh. 1995. Industrial perspective on static analysis. *Software Engineering Journal* 10, 2 (1995), 69–75.