Effective and Ineffective Software Testing Strategies

Ayaan M. Kazerouni
Virginia Tech
Blacksburg, Virginia
ayaan@vt.edu

ABSTRACT

I propose a methodology for assessing software developers' testing habits, using IDE-interaction data and contextual source code data. The data is provided in the form of Enriched Event Streams from the Mining Software Repositories Conference's 2018 Mining Challenge. I aim to form the assessment using measures such as the temporal and change-size distance between solution code edits and test code edits, solution and test executions, etc. Once the measures have been developed, I will attempt to correlate them with intermediate project outcomes such as test and build failures. Questions I hope to answer are: Can we identify effective and ineffective testing strategies? Is it better to regularly write unit tests, or is it better to perform ad-hoc testing?

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Process validation*; *Software evolution*; Software notations and tools;

KEYWORDS

ACM proceedings, LaTeX, text tagging

ACM Reference Format:
Ayaan M. Kazerouni. 2017. Effective and Ineffective Software Testing Strategies. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, Article 4, 2 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

I want to usher in continuous integration for software development, not just deployment. Assessment of software tends to focus on postmortem evaluation of metrics like correctness, merge-ability, and code coverage. This is evidenced in the current practices of continuous integration and deployment that focus on software's ability to pass unit tests before it can be merged into a deployment pipeline. However, there is little attention or tooling spared for the assessment of the software development process itself. This assessment becomes both more challenging and more crucial to address as software complexity increases. Evaluating and responding to a software developer's skills like incremental development and testing could greatly increase productivity and improve the ability to write robust and tested code.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

Incremental development with regular testing is a known best practice of software engineering, and current methods for assessing these practices are severely lacking. If we could provide continuous assessment and feedback of a software developer's programming process, we may be able to help developer's stay 'on track' with regard to adhering to best practices. This project's potential long-term impacts are what make it important enough to pursue: if developers testing habits are continuously kept on track by an intelligent system, the code may see an increase in quality, leading to less bugs shipped out to production, which comes with obvious benefits.

1.1 Proposed Solution

I aim to develop models to quantify a software developer's programming and testing habits. By measuring the programming process, I can empirically evaluate its adherence to known best practices in software engineering. Since 'quantifying the programming process' is a rather large scope, I will initially focus the practices of incremental testing and development. I should note that when I say 'incremental testing' I do not refer to the strict practice of Test-Driven Development (TDD), where a developer writes failing tests before writing the solution code to make the tests pass. I refer to the practice of writing and testing code *temporally close to each other*, meaning the developer constantly tests code as s/he writes it. Also, an important part of this process is the regular execution of tests (or at the very least, the program itself) to ensure that things are working correctly.

The dataset provided by the MSR 2018 Mining Challenge [8] facilitates this kind of analysis. I aim to use the Enriched Event Streams (EES) provided to gain an empirical of the developers' testing habits. In particular, I would like to know how often developers write or run their tests, and if they run the entire test suite or not. The 'Events' data together with the 'Contexts' data can provide rich insight into the development process, allowing me to temporally interpolate program and test executions with program and test creation. I plan to use the data related to *Specific Tools* (Testing, Build) and *File Management* (Solutions, Edits) interaction events, according to the nomenclature in [8].

A proposed road-map for this project is as follows:

Outcome measures: I plan to develop 'outcome' measures from the EES data. Possible candidates include the number or rate of test failures, failed builds, and time spent on a project. The 'Contexts' data include GitHub users and projects. Therefore another good measure of 'project health' could be the project's star count on GitHub, using the project's popularity as a proxy for success.

Process assessment: Then, I will develop quantitative measures for the testing process on each project. The measures will probably be related to things such as time between solution programming and test programming, the growth rate of the test suite vs the

solution, the number of failed builds, or the number of failed test runs.

Discrimination: I will attempt to correlate these measures against the proposed outcome variables, helping to verify their effectiveness. The hope is that this work can help discriminate between *effective* and *ineffective* software development behaviors and contribute meaningful recommendations for developing and testing software.

The work I plan to do is related to my PhD research, which is related to computer science and software engineering education. There is a wide consensus in the software engineering community that regular software testing is good, but to the best of my knowledge there is a lack of tools aimed at helping developers test effectively. A primary reason for this is that there is currently no way to assess the software development process. To properly assess regular testing and incremental development, fine-grained data is needed, and the EES data set fulfills this need. The contribution of this project will be the analysis undertaken to quantify a developer's software development process. While the metrics for software *process* will be self-developed, they will be verified against metrics for software *quality* such as the number of test failures.

There are numerous potential applications of this research including tools that can be integrated into common IDEs. These tools would continuously collect development process data and, using the analysis hypothesized in this proposal, alert the developer when they are in danger of straying from best practices (e.g. 'That was a large change, you should probably run your tests before moving on.'). Contributions of this work include an empirical understanding of how product code and solution code co-evolve in successful projects vs less successful projects.

2 RELATED WORK

There is a significant body of work revolving around the collection and analysis of data related to the programming process, and the assessment of software testing practices [1–7]. Many of the projects described below include the collection of data using custom IDE-based plugins. Some of the projects described are pedagogical in nature, and some not.

The Empirical Software Engineering group at Microsoft Research lists and ranks 145 research questions in software engineering, consulting with 1,500 Microsoft engineers [1]. Their list includes the areas *Development practices*, *Testing practices*, and *Software development process*, indicating that this is an area of research that both researchers and industry professionals are interested in.

Beller, et al. use WatchDog [1] to track IDE-interaction events across a number of IDEs in an effort to characterize the testing process. Their findings include that the majority of projects and users do not practice testing actively, and that tests and production code do not co-evolve gracefully [2]. WatchDog can provide reports on a developer's programming habits, but these reports provide statistics on the amount of test code written vs solution code, leaving the developer to judge their own testing habits.

Carter et al. model the novice programming practice as a Programming State Model [3]. They collect debugging and code-writing behavior data from Visual Studio and reduce the event stream to a series of state transitions, where each transition includes the *before* state (or *unknown*) and the *after* state. They found some success with predicting project outcomes using this model [3]. However, there is little mention of the assessment of testing practices.

My own research focuses on software engineering education. I have helped develop and maintain *DevEventTracker* [6], a custom Eclipse plugin that collects development event data from students at Virginia Tech. I use this data to model the student programming process to assess their incremental development and time management skills. We do this by developing metrics that attempt to reduce a student's event stream to a vector of metrics that accurately represent different aspects of the programming process [7].

[1] https://testroots.org

REFERENCES

[1] Andrew Begel and Tom Zimmermann. 2014. Analyze This! 145 Questions for Data Scientists in Software Engineering. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM. https://www.microsoft.com/en-us/research/publication/analyze-this-145-questions-for-data-scientists-in-software-engineering-2/

[2] 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.

[3] Adam Scott Carter and Christopher David Hundhausen. 2017. Using Programming Process Data to Detect Differences in Students' Patterns of Programming. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. ACM, New York, NY, USA, 105–110. https://doi.org/10.1145/3017680.3017785

[4] Adam S. Carter, Christopher D. Hundhausen, and Olusola Adesope. 2015. The Normalized Programming State Model: Predicting Student Performance in Computing Courses Based on Programming Behavior. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research (ICER '15)*. ACM, New York, NY, USA, 141–150. https://doi.org/10.1145/2787622.2787710

[5] R. L. Glass, R. Collard, A. Bertolino, J. Bach, and C. Kaner. 2006. Software testing and industry needs. *IEEE Software* 23, 4 (July 2006), 55–57. https://doi.org/10.1109/MS.2006.113

[6] Ayaan M. Kazerouni, Stephen H. Edwards, T. Simin Hall, and Clifford A. Shaffer. 2017. DevEventTracker: Tracking Development Events to Assess Incremental Development and Procrastination. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17)*. ACM, New York, NY, USA, 104–109. https://doi.org/10.1145/3059009.3059050

[7] Ayaan M. Kazerouni, Stephen H. Edwards, and Clifford A. Shaffer. 2017. Quantifying Incremental Development Practices and Their Relationship to Procrastination. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. ACM, New York, NY, USA, 191–199. https://doi.org/10.1145/3105726.3106180

[8] Sebastian Proksch, Sven Amann, and Sarah Nadi. 2018. Enriched Event Streams: A General Dataset for Empirical Studies on In-IDE Activities of Software Developers. In *Proceedings of the 15th Working Conference on Mining Software Repositories*.