Improving Students' Testing Practices

Gina R. Bai
North Carolina State University, Raleigh, NC, USA
Advisor: Kathryn T. Stolee
Homepage: https://ginabai.github.io
rbai2@ncsu.edu

ABSTRACT

Software testing prevents and detects the introduction of faults and bugs during the process of evolving and delivering reliable software. As an important software development activity, testing has been intensively studied to measure test code quality and effectiveness, and assist professional developers and testers with automated test generation tools. In recent years, testing has been attracting educators' attention and has been integrated into some Computer Science education programs. Understanding challenges and problems faced by students can help inform educators the topics that require extra attention and practice when presenting testing concepts and techniques.

In my research, I study how students implement and modify source code given unit tests, and how they perceive and perform unit testing. I propose to quantitatively measure the quality of student-written test code, and qualitatively identify the common mistakes and bad smells observed in student-written test code. We compare the performance of students and professionals, who vary in prior testing experience, to investigate the factors that lead to high-quality test code. The ultimate goal of my research is to address the challenges students encountered during test code composition and improve their testing skills with supportive tools or guidance.

ACM Reference Format:
Gina R. Bai. 2020. Improving Students' Testing Practices. In *42nd International Conference on Software Engineering Companion (ICSE '20 Companion), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377812.3381401

1 INTRODUCTION

Software testing is crucial to help developers detect and fix bugs and faults in software systems. With the availability and pervasive adoption of convenient testing frameworks (e.g., JUnit) and agile development methodologies (e.g. Test-Driven Development), writing unit test cases is an increasingly common practice in industry [20, 37]. As an important software development activity, testing has been intensively studied on many aspects, such as fault localization and detection [35], mutation testing [3, 20], regression testing [16], test code quality and effectiveness [2, 5, 20, 36, 44], and test suite minimization [21, 22, 40]. Multiple automated test generation tools [10, 39] have been developed as well. Some researchers have also explored the unit testing practices of professional developers/testers, including their motivation, perception of unit testing, and automated tools they adopted [9, 41]. However, how computing students and novice testers perceive and perform testing, and the quality of novice-written test code remains under-studied.

Being considered as one of the most important skills an engineer should have, however, testing still does not receive its deserved attention in most Computer Science education programs [4]. For example, some programs do not offer or only offer software testing courses as electives [4]. Chan, et al. [8] reported that only 28% of testing team members received formal training in software testing from universities. Ng and colleagues [33] found that for most companies (61.4%), less than 20% of testers have received training in software testing through university studies. Lemos, et al. [29] claimed that some university instructors tended to lack the adequate understanding of basic software testing concepts, and hence were insufficient to help students produce more reliable code.

Both educators and researchers have pointed out that the lack of formal and systematic testing education from university Computer Science programs and the lack of sufficient professional training in software testing when the students join the workforce introduced a skills gap between the Computer Science graduates and the professional testers/developers [4, 8, 33]. To bridge the gap, educators have sought to establish and enhance students' testing skills in recent years. For instance, integrating testing to CS1 course [19, 26, 31, 43], proposing postgraduate level courses focused on testing [32], and introducing tools to support students learning testing [7, 15, 43].

Despite the considerable effort made by researchers and educators, insights on how and how well students test the source code given program specifications, what challenges students encounter when performing testing, if there are differences on testing strategies and performance between students and professionals, and how to address these issues to improve students' testing skills are still lacking. We hypothesize:

> By identifying the common mistakes and challenges students have during the testing process and investigating the factors that correlate to high-quality test code, we can help students overcome these barriers and improve their testing performance with appropriate supportive tools or guidance.

To evaluate our hypothesis, we ask the following five research questions:

- **RQ1:** What are the problem-solving strategies adopted by students during source code composition to pass the unit tests? **(Study-1)**

ICSE '20 Companion, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7122-3/20/05…$15.00
https://doi.org/10.1145/3377812.3381401

- **RQ2:** How does the design of unit tests (single assertion vs. multiple assertions) impact on students' programming performance and intention of persistence? **(Study-2)**
- **RQ3:** How do students perceive unit testing and what are the common mistakes and challenges observed during students' test code composition? **(Study-3)**
- **RQ4:** What are the differences between professional developers and students in terms of perception and performance on unit testing? **(Study-4)**
- **RQ5:** How can educators and practitioners guide students to perform high-quality testing? **(Study-5)**

We focus on how students perceive and perform unit testing. The expected contributions of my research include:

- Better understanding of how and how well students test source code given program specifications.
- Insights on testing activities that students need more assistance with.
- Applicable suggestions on unit test designs to assist students with reading and understanding the test code.
- Educational guidance to enhance students' testing skills.

2 RELATED WORK

My research is based on prior work that explores the quality and effectiveness of test code. However, few prior works studied the quality of student-written test cases.

Test Code Quality and Effectiveness:

Test code quality is usually evaluated with code coverage and mutation testing [2, 20, 36, 44]. Few focused on the quality of student-written test code [12, 13, 42] and students' perspectives on unit testing [18]. The metrics used in these studies are limited to code coverage, which is argued to be not a strong indicator for test effectiveness [24]. Athanasiou, et al. [5] introduced a complex model that assesses test code quality by combining source code metrics that reflect three main aspects of test code quality: completeness (metrics: code coverage, assertions-McCabe ratio), effectiveness (metrics: assertion density, directness) and maintainability (adjusted Software Improvement Group (SIG) quality model).

In my research, we quantitatively measure the quality of student-written test code, and qualitatively categorize the mistakes and bad smells [11] observed in the test code. We are also interested in learning the challenges that students encounter during testing.

Software Testing Education:

As Jones pointed out that, "there is a basic set of testing skills that every undergraduate should acquire" [27], the practice of software testing, such as unit testing, is suggested to be integrated into the Computer Science and Software Engineering curricula as part of the educational experience [1, 27]. Educators have suggested different approaches to introduce testing in Computer Science curricula, such as requiring students to turn in tests along with their solutions [14, 19], asking students to perform black-box testing of software seeded with errors [25, 33], and instructing students to conduct peer testing [18, 38].

Aniche, et al. [4] explored the challenges that students face when learning software testing. They reported the common mistakes into eight categories (ordered by their frequency): test coverage, maintainability of test code, understanding testing concepts, boundary testing, state-based testing, assertions, mock objects, and tools. The researchers discussed the testing topics that students find hardest to learn, for example, applying Modified Condition/Decision Coverage criteria to test complicated conditions, and deciding how much to test is enough. They also observed that the students are not sufficiently aware of the difficulty of testability.

In my research, we investigate how students perceive and perform unit testing. We explore the factors that lead to high-quality test code. We recruit students and professional testers that vary in unit testing experience to participate in these studies intending to explore if education and prior experience on testing impact students' testing performance and strategies.

3 RESEARCH

In the following, I discuss the experimental designs and evaluation plans of the studies.

3.1 Study-1: Test Code Comprehension - Problem Solving Strategies (ICPC 2019) [6]

3.1.1 Motivation. In Computer Science, it is a common practice for instructors to provide students with detailed program specifications and a set of unit tests to verify the functionality of student-written code. However, knowledge is lacking on how students comprehend program specifications and test code, and what are the tools and strategies they adopt to facilitate programming and satisfying the unit tests.

3.1.2 Methodology. We conducted an exploratory case study to reveal the tools and strategies students use during program implementation. Students were expected to compose regular expressions in Java that pass unit tests illustrating the intended behavior. We combined surveys and screen-capture videos to better understand students' overall performance on the tasks.

3.1.3 Results. Our findings indicate that the testing environments are not always sufficient for regular expressions. For example, the JUnit tests return only pass/fail for regular expression users and the Eclipse built-in debugger does not explain why the matching fails; meanwhile, web-based tools that support dynamic testing can visualize the matching results and explain the syntax of the regular expression. The results show that participants who consulted web-based tools passed more tests than those who did not consult web-based tools.

We also observed that, when being tasked to examine the strings against a given pattern and extract pertinent information from a string subject to a regular expression, participants spent a longer time reading the test cases (2.8 minutes vs. 1.9 minutes) and checked the test cases more frequently (5.4 times vs. 4.2 times) than being tasked to validate the entire content of a string against a regular expression on average.

3.2 Study-2: Test Code Comprehension - Design & Intention of Persistence (In Progress)

3.2.1 Motivation. There are several guidelines of JUnit test design that suggest testing one function per unit test [28], or even just one assertion in each test cases [34]; however, Ma'ayan reported that, with JUnit testing, there was 61% of all tests contained more

than a single assertion; meanwhile, about 78% of all `assertTrue` and `assertFalse` assertions do not contain a customized error message [30]. There is no empirical study to demonstrate if single-assertion JUnit tests benefit the users, especially students. Moreover, as positive feedback may be important to student persistence in introductory level Computre Science courses, we are interested to see if single-assertion JUnit tests, which are more likely to provide positive granular feedback than the multiple-assertions test cases do, could potentially increase the persistence intention of students in Computer Science.

3.2.2 Methodology. We adopt A/B testing in this study: single-assertion test cases versus test cases consist of multiple assertions. To evaluate students' implementation given the unit tests and description of expected program behaviors, we measure the source code from both groups via the number of passed assertions. A high pass rate indicates a good implementation.

To evaluate students' persistence intention and degree of confidence, we compare the differences among students' self-evaluated CS abilities and the likelihood of taking more CS courses or pursuing a CS degree between preliminary and post surveys.

3.3 Study-3: Test Code Composition - Students (In Progress)

3.3.1 Motivation. Studies have been done on test code quality [2, 5, 20, 36, 44]; however, few focused on the quality of student-written test code [12, 13, 18], and the metrics are limited to code coverage, which is argued to be not a strong indicator for test effectiveness [24]. In addition, how students perceive and write unit tests remain under-studied.

The goal of this study is to shed light on students' testing strategies, the challenges they encounter during test code composition, as well as the factors that correlate to high performance.

3.3.2 Methodology. We survey students to learn how they perceive unit testing, how they usually test their own code, what tools and information sources do they consult. We instruct participants to perform black-box testing and white-box testing on two Java programs respectively in the Eclipse IDE. Participants are instructed to test the program requirements as thoroughly as possible. When finished, participants are expected to complete a post survey and report the challenges they encountered and things they would like to receive help on during the study.

For measuring test code quality, in addition to code coverage, our metrics include the number of detected bugs, requirement coverage, and mutation score. Besides the quantitative measurements, we investigate the common mistakes and code smells in the test code. We adopt the test smells categorized by Deursen, et al. [11] and code smells defined by Fowler, et al. [17]. For example, redundant assertions, conditional test logic, and test names that do not reflect the test intention are potential test smells.

3.4 Study-4: Test Code Composition - Professionals (Proposed)

3.4.1 Motivation. After learning how and how well students test the given source code (**Study-3**), this study focuses on how professional developers/testers perform unit testing and explores if there is any difference between professionals and students in terms of testing performance and strategies. With a better understanding of the practices that correlate to higher test code quality, we are able to generalize the strategies or perceptions that may be helpful to students.

3.4.2 Methodology. We will invite professional developers/testers to participate in this study. They will be instructed to complete the same surveys and work on the same testing projects as we provide the students in **Study-3**. The same metrics will be applied to professional-written test code, and the results will be compared against the students' testing performance. We will explore the factors that lead to higher test code quality.

Our results will provide educational tips on unit testing practices for students to improve their testing skills and performance.

3.5 Study-5: Test Code Quality - Educational Guidance and Improvement (Proposed)

3.5.1 Motivation. At this stage, we will have learned the mistakes and challenges that students have during unit testing (**Study-3**) and the factors that correlate to high-quality test code (**Study-4**). Therefore, we will introduce a supportive tool or an educational guidance for students and novice testers to assist them in testing programs in a systematical style. For example, a tool that can simultaneously decompose specifications and select inputs using techniques such as boundary value testing for testers. We envision this tool to improve students' unit testing practices and hence partially reduce technology companies' training cost on new hires.

3.5.2 Methodology. To evaluate and validate the usefulness of the proposed tool/guidance, we will invite both students and professionals to participate in this study. We will divide the participants into three groups: 1) students with the proposed tool/guidance 2) students without the proposed tool/guidance, 3) professional developers without the proposed tool/guidance. Similar to prior studies, all groups will be instructed to perform black-box testing and white-box testing, and their performance will be measured and compared with the same metrics used in **Study-3** and **Study-4**. Feedback on their experience of this experiment will be collected to further improve the supportive tool or educational guidance.

4 TIMELINE

The author is a fourth year Computer Science PhD student at North Carolina State University working with Dr. Kathryn Stolee. The proposed research plan timeline is as follows:

Milestones	Expected Completion
Study-1	Completed: ICPC 2019
Study-2	Spring 2020 - Summer 2020*
Study-3	Spring 2020*
Study-4	Fall 2020 - Spring 2021*
Study-5	Fall 2021 - Spring 2022*
Defense	Spring 2022*

*Building on my current publication record in software engineering [6, 23, 45], the potential venues for submission include ISSTA, ICSE, FSE, ICPC, and other peer-reviewed conferences and journals.

ACKNOWLEDGMENTS

Special thanks to my Ph.D. advisor Kathryn T. Stolee for her continuous support and patience. This material is based upon work supported by the NSF under Grant #1645136 and Grant #1749936.

REFERENCES

[1] ACM, 2013. Computer Science Curricula Recommendations: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science. https://www.acm.org/education/curricula-recommendations. (ACM, 2013).

[2] T. L. Alves and J. Visser. 2009. Static Estimation of Test Coverage. In *International Working Conference on Source Code Analysis and Manipulation*. 55–64.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche. 2005. Is Mutation an Appropriate Tool for Testing Experiments?. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*. ACM, New York, NY, USA, 402–411.

[4] Maurício Aniche, Felienne Hermans, and Arie van Deursen. 2019. Pragmatic Software Testing Education. In *Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, New York, NY, USA, 414–420.

[5] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman. 2014. Test Code Quality and Its Relation to Issue Handling Performance. *IEEE Transactions on Software Engineering* 40, 11 (Nov 2014), 1100–1125.

[6] Gina R. Bai, Brian Clee, Nischal Shrestha, Carl Chapman, Cimone Wright, and Kathryn T. Stolee. 2019. Exploring Tools and Strategies Used During Regular Expression Composition Tasks. In *Proceedings of the 27th International Conference on Program Comprehension (ICPC '19)*. IEEE Press, Piscataway, NJ, USA, 197–208.

[7] Michael K. Bradshaw. 2015. Ante Up: A Framework to Strengthen Student-Based Testing of Assignments. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. ACM, New York, NY, USA, 488–493.

[8] F. T. Chan, T. H. Tse, W. H. Tang, and T. Y. Chen. 2005. Software testing education and training in Hong Kong. In *Fifth International Conference on Quality Software (QSIC'05)*. 313–316.

[9] Ermira Daka and Gordon Fraser. 2014. A Survey on Unit Testing Practices and Problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 201–211.

[10] Ermira Daka, José Miguel Rojas, and Gordon Fraser. 2017. Generating Unit Tests with Descriptive Names or: Would You Name Your Children Thing1 and Thing2?. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 57–67.

[11] Arie Deursen, Leon M.F. Moonen, A. Bergh, and Gerard Kok. 2001. *Refactoring Test Code.* Technical Report. Amsterdam, The Netherlands, The Netherlands.

[12] Stephen H. Edwards. 2003. Improving Student Performance by Evaluating How Well Students Test Their Own Programs. *J. Educ. Resour. Comput.* 3, 3, Article 1 (Sept. 2003).

[13] Stephen H. Edwards and Zalia Shams. 2014. Do Student Programmers All Tend to Write the Same Software Tests?. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITiCSE '14)*. ACM, New York, NY, USA, 171–176.

[14] Stephen H. Edwards, Zalia Shams, Michael Cogswell, and Robert C. Senkbeil. 2012. Running Students' Software Tests Against Each Others' Code: New Life for an Old "Gimmick". In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12)*. ACM, New York, NY, USA, 221–226.

[15] S. Elbaum, S. Person, J. Dokulil, and M. Jorde. 2007. Bug Hunt: Making Early Software Testing Lessons Engaging and Affordable. In *29th International Conference on Software Engineering (ICSE'07)*. 688–697.

[16] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 235–245.

[17] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.

[18] Alessio Gaspar, Sarah Langevin, Naomi Boyer, and Ralph Tindell. 2013. A Preliminary Review of Undergraduate Programming Students' Perspectives on Writing Tests, Working with Others, & Using Peer Testing. In *Proceedings of the 14th Annual ACM SIGITE Conference on Information Technology Education (SIGITE '13)*. ACM, New York, NY, USA, 109–114.

[19] Michael H. Goldwasser. 2002. A Gimmick to Integrate Software Testing Throughout the Curriculum. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE '02)*. ACM, New York, NY, USA, 271–275.

[20] G. Grano, F. Palomba, and H. C. Gall. 2019. Lightweight Assessment of Test-Case Effectiveness using Source-Code-Quality Indicators. *IEEE Transactions on Software Engineering* (2019), 1–1.

[21] Alex Groce, Josie Holmes, and Kevin Kellar. 2017. One Test to Rule Them All. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 1–11.

[22] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The Art of Testing Less Without Sacrificing Quality. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 483–493.

[23] Nasif Imtiaz, Justin Middleton, Joymallya Chakraborty, Neill Robson, Gina Bai, and Emerson Murphy-Hill. 2019. Investigating the Effects of Gender Bias on GitHub. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 700–711.

[24] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 435–445.

[25] Ursula Jackson, Bill Z. Manaris, and Renée A. McCauley. 1997. Strategies for Effective Integration of Software Engineering Concepts and Techniques into the Undergraduate Computer Science Curriculum. In *Proceedings of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education (SIGCSE '97)*. ACM, New York, NY, USA, 360–364.

[26] David Janzen and Hossein Saiedian. 2008. Test-driven Learning in Early Programming Courses. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08)*. ACM, New York, NY, USA, 532–536.

[27] E. L. Jones. 2001. An experiential approach to incorporating software testing into the computer science curriculum. In *31st Annual Frontiers in Education Conference. Impact on Engineering and Science Education. Conference Proceedings (Cat. No.01CH37193)*, Vol. 2. F3D–7.

[28] Koskela, L. 2013. *Effective Unit Testing: A Guide for Java Developers.* Manning.

[29] Otavio Lemos, FÃąbio Silveira, Fabiano Ferrari, and Alessandro Garcia. 2017. The Impact of Software Testing Education on Code Reliability: An Empirical Assessment. *Journal of Systems and Software* (03 2017).

[30] Dor D. Ma'ayan. 2018. The Quality of Junit Tests: An Empirical Study Report. In *Proceedings of the 1st International Workshop on Software Qualities and Their Dependencies (SQUADE '18)*. ACM, New York, NY, USA, 33–36.

[31] Will Marrero and Amber Settle. 2005. Testing First: Emphasizing Testing in Early Programming Courses. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '05)*. ACM, 4–8.

[32] M. D. Mohamed Suffian, S. Ibrahim, and M. R. Abdullah. 2014. A proposal of post-graduate programme for software testing specialization. In *2014 8th. Malaysian Software Engineering Conference (MySEC)*. 342–347.

[33] S. P. Ng, T. Murnane, K. Reed, D. Grant, and T. Y. Chen. 2004. A preliminary survey on software testing practices in Australia. In *2004 Australian Software Engineering Conference. Proceedings.* 116–125.

[34] Roy Osherove. 2009. *The Art of Unit Testing: With Examples in .Net* (1st ed.). Manning Publications Co., Greenwich, CT, USA.

[35] Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. 2008. Finding Errors in .Net with Feedback-directed Random Testing. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. ACM, 87–96.

[36] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2016. Automatic Test Case Generation: What if Test Code Quality Matters?. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 130–141.

[37] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. 2012. Understanding Myths and Realities of Test-suite Evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 33, 11 pages.

[38] James Robergé and Candice Suriano. 1994. Using Laboratories to Teach Software Engineering Principles in the Introductory Computer Science Curriculum. In *Proceedings of the Twenty-fifth SIGCSE Symposium on Computer Science Education (SIGCSE '94)*. ACM, New York, NY, USA, 106–110.

[39] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2015. Automated Unit Test Generation During Software Development: A Controlled Experiment and Think-aloud Observations. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 338–349.

[40] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. 1998. An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites. In *Proceedings of the International Conference on Software Maintenance (ICSM '98)*. IEEE Computer Society, Washington, DC, USA, 34–.

[41] P. Runeson. 2006. A survey of unit testing practices. *IEEE Software* 23, 4 (July 2006), 22–29.

[42] Lilian Passos Scatalon, Jeffrey C. Carver, Rogério Eduardo Garcia, and Ellen Francine Barbosa. 2019. Software Testing in Introductory Programming Courses: A Systematic Mapping Study. In *Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, NY, USA, 421–427.

[43] Jaime Spacco and William Pugh. 2006. Helping Students Appreciate Test-driven Development (TDD). In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 907–913.

[44] J. Voas. 1997. How assertions can increase test effectiveness. *IEEE Software* 14, 2 (Mar 1997), 118–119.

[45] P. Wang, G. R. Bai, and K. T. Stolee. 2019. Exploring Regular Expression Evolution. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 502–513.