Editor: **Jeffrey C. Carver**
University of Alabama
carver@cs.ua.edu

# Software Maintenance and Evolution and Automated Software Engineering

Jeffrey C. Carver and Alexander Serebrenik

**THIS ISSUE'S ARTICLE** reports on the 33rd International Conference on Software Maintenance and Evolution (ICSME 17) and 32nd International Conference on Automated Software Engineering (ASE 17). Feedback or suggestions are welcome. In addition, if you try or adopt any of the practices described in the column, please send Jeffrey Carver and the paper authors a note about your experiences.

## The Relationship between Personality and Project Success

"Personality and Project Success: Insights from a Large-Scale Study with Professionals," by Xin Xia and his colleagues, explores the relationship between the personalities of the team manager and members and project success, as scored by the team's company.[1] The authors analyzed data from 28 projects at two large Chinese IT companies, Insigma Global Service and Hengtian, and surveyed 346 professionals involved in these projects.

The participants completed the DISC (dominance, influence, steadiness, and compliance) personality test, which is based on team-building theories. The results showed that the teams with dominant managers, more influential members, and fewer dominant members had higher success scores.

On the basis of this finding, Xia and his colleagues recommend that teams should have at most 15 percent dominant members and at least 10 percent influential members. Furthermore, the authors observed that the projects with dominant managers achieved higher scores regarding internal issues, software quality, and customer satisfaction. Similarly, the percentage of influential members correlated positively with software quality and customer satisfaction. You can access this paper at bit.ly/PD_2018_March_1.

## Fixing Flaky Tests

In "Does Refactoring of Test Smells Induce Fixing Flaky Tests?," Fabio Palomba and Andy Zaidman explain how they try to fix flaky tests.[2] Owing to nondeterministic behavior, flaky tests exhibit both a passing and a failing result. They can hide real defects, making those defects more difficult to reproduce, and they reduce the tester's confidence in the quality of the system under test.

Palomba and Zaidman relate test flakiness to test smells previously identified in the literature. A study of 19,532 JUnit test methods from 18 open-source software systems showed that almost 45 percent of the test methods were flaky. The most frequent reasons for flakiness were incorrect use of concurrent constructs and dependency on external resources or networks. In addition, 61 percent of the flaky tests exhibited a test smell. Flakiness was tied directly to test smells 54 percent of the time; refactoring the tests eliminated the flakiness.

This study confirms the importance of adopting tools that can identify the presence of test smells, whose removal will help eliminate flakiness. This paper won an IEEE TCSE (Technical Council on Software Engineering) Distinguished Paper award. You can access it at bit.ly/PD_2018_March_2.


0740-7459/18/$33.00 © 2018 IEEE

## Technical Debt

At ICSME 17, three papers discussed technical debt—suboptimal decisions that create longer-term negative effects unless they're revisited and corrected.

"The Pricey Bill of Technical Debt: When and by Whom Will It Be Paid?," by Terese Besker and her colleagues, describes a web survey of 258 participants and follow-up interviews with 32 industrial software practitioners to identify when and by whom technical debt is repaid.[3] The results show that dealing with technical debt wasted an estimated 36 percent of development time. Most of this wasted time was spent understanding or measuring the technical debt. This empirical quantification of technical debt's impact enables practitioners to benchmark their projects. You can access this paper at bit.ly/PD_2018_March_3.

The second and third papers discussed *self-admitted technical debt* (SATD). SATD refers to situations in which a developer acknowledges, through comments, that one of his or her decisions or implementations is inadequate or suboptimal.

"An Empirical Study on the Removal of Self-Admitted Technical Debt," by Everton Maldonado and his colleagues, reports on a study of five large open source projects.[4] The results indicate that the developers removed most SATD. The person who inserted the SATD was more likely to remove it, and remove it more quickly, than others were. These observations point to the importance of developers' personal awareness and responsibility for their code. You can access this paper at bit.ly/PD_2018_March_4.

"Recommending When Design Technical Debt Should Be Self-Admitted," by Fiorella Zampetti and her colleagues, describes TEDIOUS (*Te*chnical *D*ebt *I*dentification *S*ystem), a machine-learning approach.[5] TEDIOUS identifies code fragments a developer should have marked as SATD. It leverages method-level features such as independent variables, including source code structural metrics, readability metrics, and warnings raised by static-analysis tools. It has achieved an average precision of 67 percent and a recall of 55 percent.

So, whereas Maldonado and his colleagues highlight the importance of a developer's awareness of SATD, Zampetti and her colleagues help increase this awareness through an automated tool. You can access this paper at bit.ly/PD_2018_March_5.

> The projects with dominant managers achieved higher scores regarding internal issues, software quality, and customer satisfaction.

## Locating Appropriate APIs

"APIBot: Question Answering Bot for API Documentation," by Yuan Tian and her colleagues, explains how they addressed the daunting task of finding information about APIs by constructing APIBot, a question-answering (QA) bot.[6] For example, if a developer asks, "How can I convert all the characters in a string to lowercase?," APIBot will return "The method `toLowerCase()` converts all the characters in this string to lowercase, using the rules of the default locale."

Tian and her colleagues note that applying well-established general-purpose QA systems to API documentation poses three key challenges:

- The QA process must consider domain-specific patterns and software-specific terms.
- Much of the semantics of an API's documentation is hidden in its implicit structure.
- General-purpose QA bots require much manually created training data that's not necessarily available for API documentation.

To address these challenges, APIBot introduces novel components on top of SiriusQA, a general-purpose QA system. In an empirical evaluation of APIBot on 92 API questions, APIBot achieved a Hit@5 score of 0.706 (the correct answer was among the top five answers returned about 70 percent of the time).

The main challenge for APIBot is to increase its accuracy to 90 percent or higher—the level expected by most practitioners who responded to a post-study survey. You can access this paper at bit.ly/PD_2018_March_6.

## Regular Expression Formatting

"Exploring Regular Expression Comprehension," by Carl Chapman and his colleagues, examines the

## ABOUT THE AUTHORS

**JEFFREY C. CARVER** is a professor in the University of Alabama's Department of Computer Science. Contact him at carver@cs.ua.edu.

**ALEXANDER SEREBRENIK** is an associate professor in Eindhoven University of Technology's Department of Mathematics and Computer Science. Contact him at a.serebrenik@tue.nl.


most comprehensible ways to represent regular expressions (*regexes*).[7] Regexes are commonly used in such diverse applications as search engines, database queries, and network security. However, it's not always clear which of the many ways to represent a collection of strings is the most understandable and maintainable. For example, to represent numbers smaller than 1,000, `[1-9][0-9]?[0-9]?`, `[1-9][0-9]{0,2}`, or `[1-9]\d{0,2}` match the same language, but which is best?

The authors measured regex comprehension on 42 pairs of regexes, using 180 participants and an empirical study of nearly 14,000 regexes and their features. The results showed that

- using an explicit range (for example, `[0-9]`) is often more understandable than a default character class (for example, `[\d]`);
- the larger the deterministic finite automaton corresponding to a regex was (up to size eight), the more understandable the regex was; and
- you can use a combination of community standards and understandability metrics to identify smelly and nonsmelly representations of regexes.

These findings provide insight for developers working with regexes to help them improve understandability, reduce faults, improve the effectiveness of code review when changes are made, and reduce maintenance costs. You can access this paper at bit.ly/PD_2018_March_7.

## References


1. X. Xia et al., "Personality and Project Success: Insights from a Large-Scale Study with Professionals," *Proc. 33rd Int'l Conf. Software Maintenance and Evolution* (ICSME 17), 2017, pp. 318–328.
2. F. Palomba and A. Zaidman, "Does Refactoring of Test Smells Induce Fixing Flaky Tests?," *Proc. 33rd Int'l Conf. Software Maintenance and Evolution* (ICSME 17), 2017, pp. 1–12.
3. T. Besker, A. Martini, and J. Bosch, "The Pricey Bill of Technical Debt: When and by Whom Will It Be Paid?," *Proc. 33rd Int'l Conf. Software Maintenance and Evolution* (ICSME 17), 2017, pp. 13–23.
4. E. Maldonado et al., "An Empirical Study on the Removal of Self-Admitted Technical Debt," *Proc. 33rd Int'l Conf. Software Maintenance and Evolution* (ICSME 17), 2017, pp. 238–248.
5. F. Zampetti et al., "Recommending When Design Technical Debt Should Be Self-Admitted," *Proc. 33rd Int'l Conf. Software Maintenance and Evolution* (ICSME 17), 2017, pp. 216–226.
6. Y. Tian et al., "APIBot: Question Answering Bot for API Documentation," *Proc. 32nd Int'l Conf. Automated Software Eng.* (ASE 17), 2017, pp. 153–158.
7. C. Chapman, P. Wang, and K.T. Stolee, "Exploring Regular Expression Comprehension," *Proc. 32nd Int'l Conf. Automated Software Eng.* (ASE 17), 2017, pp. 405–416.