ICSE Highlights

Jeffrey C. Carver and Alexander Serebrenik

THIS ISSUE’S COLUMN reports on papers from the 2017 International Conference on Software Engineering. Feedback and suggestions are welcome. In addition, if you try or adopt any of the practices included in the column, please send Jeffrey Carver and the paper authors a note about your experiences.

Connecting Source Code to Runtime Activity

“Context-Based Analytics—Establishing Explicit Links between Runtime Traces and Source Code,” by Jürgen Cito and his colleagues, discusses an approach for more easily identifying the source code associated with specific runtime activity. Developers have many available data sources for investigating the source of runtime problems. So, the process is often costly, and developers must use multiple tools to analyze each data source. Context-based analytics reduces the steps required to establish an explicit link between runtime behavior and the underlying code. This approach constructs a graph based on an application context model to make explicit information that was previously implicit.

Cito and his colleagues evaluated their approach on two runtime problems from IBM’s Active Deploy service, which lets developers release new versions of their cloud-based software with no downtime. The authors’ approach reduced the analysis steps by 48 percent and reduced the number of program traces developers had to inspect by 40 percent. You can access this paper at bit.ly/PD_Nov_1.

Analyzing Aggregation Techniques

“The Use of Summation to Aggregate Software Metrics Hinders the Performance of Defect Prediction Models,” by Feng Zhang and his colleagues, addresses the challenge of predicting defect locations on the basis of system measurements. Defect prevention techniques face a gap between the granularity of the measurements—that is, methods—and the desired granularity of predictions—that is, files. Although method-level prediction metrics exist, their accuracy is suspect. So, practitioners see file-level prediction as more accurate and useful. To provide file-level predictors, defect prediction techniques aggregate method-level metrics to the file level. The most common aggregation technique is summation.

Zhang and his colleagues describe how different aggregation techniques affect defect prediction. On the basis of an analysis of 255 open source systems, the authors conclude that summation isn’t the best technique. Summation performed best in only 11 percent of the projects, whereas a combination of all the aggregation techniques performed best in 40 percent of the projects. For effort-aware defect prediction—ranking files so that the most defects can be found with the least effort—aggregation measures based on central tendency performed the best.

The key lesson from this paper is that practitioners should be cautious about using only one defect prediction technique, especially if it relies on summation. You can access this paper at bit.ly/PD_Nov_2.

Software Development and Energy Efficiency

“Awakening Awareness on Energy Consumption in Software Engineering,” by Erik Jagroep and his colleagues, describes how feedback during development regarding energy consumption and performance created awareness among the stakeholders of two commercial software products. Members of the information and communications technology industry commonly try to understand and improve hardware’s energy impact. In contrast, industry has less practical knowledge of, and places less emphasis on, reducing software’s energy impact.

In the case studies described in this paper, Jagroep and his colleagues provided direct feedback to project stakeholders regarding how development choices affected energy efficiency and performance. To present energy-related information
to stakeholders, the authors used an energy dashboard that compactly displays key metrics and how those metrics changed from the previous software version. Overall, these efforts to create awareness of energy consumption encouraged the project stakeholders to discuss these topics and use that information while making project decisions. Organizational policies are necessary to maintain this awareness over time.

One case study participant, software architect Jason Hewitt, said, “My background was in mobile-phone software, where energy consumption was obviously very important. In these small, closed systems it was possible to demonstrate a clear correlation between software running on the system and the energy consumption. In software engineering we have lost this relationship.” You can access this paper at bit.ly/PD_Nov_3.

**Experimentation in Continuous Deployment**

“Characterizing Experimentation in Continuous Deployment: A Case Study on Bing,” by Katja Kevic and her colleagues, offers insights gained from two-and-a-half years of data from experiments on new software features. Using continuous deployment, developers of web-based software can conduct small-scale experiments to test new features with a subset of their user base. These experiments help the development organization make decisions about which features to fully deploy.

Kevic and her colleagues analyzed 21,230 experiments conducted in the Bing project to characterize those experiments’ important features. From this analysis, they made three interesting observations. First, the experiments took on average 42 days, including multiple iterations of one- to two-week runs. Second, one-third of the experiments were ultimately deployed to all users, with that rate varying across component type. Finally, the changes that were ultimately deployed to all users were significantly larger in terms of files, changesets, and contributors. You can access this paper at bit.ly/PD_Nov_4.

Practitioners should be cautious about using only one defect prediction technique.

**Feature Freeze and Continuous Delivery**

“Towards Continuous Delivery by Reducing the Feature Freeze Period: A Case Study,” by Eero Laukkanen and his colleagues, reports on Ericsson’s experiences with how feature freeze affects development. Organizations that follow a more traditional release cycle often institute a feature freeze period before each release, during which developers don’t change the code. Ericsson was struggling with this process and desired to move toward continuous delivery.

The first key observation was that the actual feature freeze practice differed from the intended practice. When the feature freeze period was too long, developers tended to not respect it. Also, as the feature freeze date approached, developers felt additional pressure to complete features, resulting in an increased development rate.

To address some of these issues and to move closer to continuous delivery, Ericsson changed its testing process in two ways. First, it used test automation to reduce the freeze period by 56 percent. Second, it changed testing to be more continuous rather than having developers wait for the feature freeze period. Although these changes had mixed results across teams, overall, the reduced freeze period resulted in fewer changes during the freeze, a lower commit rate as the freeze date approached, and fewer changes near the release date. You can access this paper at bit.ly/PD_Nov_5.

**References**


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.