Empirical Software Engineering, Predictive Models, and Product Lines

Jeffrey C. Carver, Eduardo Santana de Almeida, Rafael Capilla, Leandro Minku, Marco Torchiano, and Alejandro Valdezate

THIS ISSUE’S COLUMN reports on papers presented at the 11th International Symposium on Empirical Software Engineering and Measurement (ESEM 17), 13th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 17), and 21st International Systems and Software Product Line Conference (SPLC 17). 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.

Tools and Delivery Speed

“Improving the Delivery Cycle: A Multiple-Case Study of the Toolchains in Finnish Software Intensive Enterprises,” by Simo Mäkinen and his colleagues, investigates how different categories of tools affected the speed of software delivery cycles in Finnish companies. On the basis of qualitative semistructured interviews of developers from 18 organizations working in different domains (for example, consulting, web services, telecommunications, and mobile games), the authors identified 15 categories of tools across the software lifecycle:

- requirements (elicitation, backlog management, and bug tracking),
- development (version control, build, continuous integration, and artifact repository),
- operations (provisioning/environments and deployment),
- testing (unit, UI, and acceptance),
- quality (performance and code review), and
- feedback (communication/feedback).

These tools ranged from commodity tools used in most organizations, to modestly used tools that were missing from some organizations, to uncommon tools that were used in only a few organizations.

The reasons for gaps in the toolchain in some organizations included alternate enactment of activities (for example, manual or outsourced), limited relevance in the domain (for example, lack of UIs in embedded systems), lack of knowledge, and company culture. A key finding was that companies aiming to deploy software more rapidly should strive for fewer manual steps. You can access this paper at bit.ly/PD_2018_May_1.

Release, Deployment, and Energy Use

“Estimating Energy Impact of Software Releases and Deployment Strategies: The KPMG Case Study,” by Roberto Verdecchia and his colleagues, reports on a study of industrial data to understand how various factors impacted a software application’s energy use. The authors used data from KPMG to study the impact of three factors:

- The release. They studied two functionally similar releases.
- The deployment strategy. They studied whether the database and web application were...
deployed on the same server or different servers.

- The **use case scenario**. The two scenarios involved users completing a common questionnaire or generating a report based on the questionnaire data.

The authors have provided an online replication package for interested readers (see bit.ly/PD_2018_May_2a).

All three factors significantly affected the underlying application’s energy consumption. But there was no consistent pattern, owing to the significant interactions among the factors. These findings indicate that although the release strategy, deployment strategy, and chosen use case are all important, there’s no absolute best approach for all circumstances. The key finding is that for each application and use case, the software providers should conduct a similar analysis to determine the best strategy. You can access this paper at bit.ly/PD_2018_May_2.

### Classifying Developers

“Characterizing Software Developers by Perceptions of Productivity,” by André Meyer and his colleagues, discusses a survey of 413 Microsoft developers. The authors aimed to characterize the developers by their perception of what it means to be productive.

On the basis of the survey (available at bit.ly/PD_2018_May_3a), Meyer and his colleagues identified six types of developers:

- The **social developer** feels productive when helping coworkers or collaborating. He or she likes to arrive early or work late and focus on a single task.
- The **lone developer** avoids disruptions and noise and feels productive when working on tasks (solving problems, fixing bugs, or coding features) quietly, with little or no interruption or social interaction.
- The **focused developer** feels productive when working efficiently on one task at a time.
- The **balanced developer** feels productive when working on relevant, clear, and familiar tasks and is less affected by disruptions. He or she likes to arrive early or stay late.
- The **leading developer** feels productive when participating in meetings and responding to emails.
- The **goal-oriented developer** feels productive when completing or making progress on a task and is more open to meetings and emails in case they help achieve the goal.

Knowledge of these productivity types can be useful for organizations that want to create the most productive work environments for different types of developers. You can access this paper at bit.ly/PD_2018_May_3.

### Estimating Issue Resolution Time

“Multi-objective Search-Based Approach to Estimate Issue Resolution Time,” by Wisam Al-Zubaidi and his colleagues, presents an approach to estimate the time required to resolve an issue. These estimates can be used to inform

- users about when a new functionality will become available,
- bug reporters about when a fix will be ready, or
- project managers about when new releases will be finished.

The authors’ approach uses data from previously closed issues to build tree-like models to estimate issue resolution time. It seeks to maximize accuracy and minimize tree size. The smaller trees are human-readable, thereby enabling software engineers to understand how the estimations are made.

Results from an evaluation of 8,260 issues from five open source projects show that this approach outperformed other common estimation models (linear regression, case-based reasoning, and random forests). In particular, this approach obtained a mean absolute error that was 24 to 40 percent smaller than that obtained by random forests (the best among those three other estimation models). Furthermore, this approach’s estimates were 50 to 60 percent more accurate than random guessing, on average. You can access this paper at bit.ly/PD_2018_May_4.

### Product Line Testing

“Product Line Engineering on the Right Side of the ‘V,’” by Susan Gregg and her colleagues, offers a practical testing approach for products developed using feature-based product line engineering (PLE). This approach reduces the cost and effort of testing activities by increasing testing accuracy and decreasing testing activity. It increases accuracy by using the PLE factory to configure test cases to exactly match the product assets to be tested. It decreases testing activity by testing inside the PLE factory wherever possible and appropriate.

The authors present five key principles for deciding whether to test something once, inside the factory, or once per product, outside the factory:

- Perform standalone testing inside.
Because testing occurs while the product line is continually evolving, the approach uses branching policies to define stable spaces for testing operations. It draws from the experience of the AEGIS Weapon System product line (a highly integrated naval-combat ship system). You can access this paper at bit.ly/PD_2018_May_5.

**Agile Product Lines**

“Agile Tames Product Line Variability: An Agile Development Method for Multiple Product Lines of Automotive Software Systems,” by Kengo Hayashi and his colleagues, describes stringent requirements’ role in the diversity of automotive software products when agility is necessary to deliver software. The authors devised APLE, an extended agile PLE method. APLE divides the application-engineering phase into a sequence of iterative processes and performs minor iteration loops in a project and major iteration loops across projects. With APLE, developers can manage the diversity of multiple product lines by iteratively reusing assets, which decreases the cost and complexity of maintaining those lines. Developers can reuse variation points several times while engineering multiple applications. The implementation of APLE in DENSO’s automotive division over 10 months and 22 two-week sprints decreased the complexity of handling multiple product lines. The resulting productivity volatility was narrowed down to lower than 20 percent, and mostly lower than 10 percent, indicating a stable development approach. You can access this paper at bit.ly/PD_2018_May_6.

**Microservices and Product Lines**

“Using Microservices and Software Product Line Engineering to Support Reuse of Evolving Multi-tenant SaaS,” by Leonardo Tizzei and his colleagues, reports on the integrated use of a microservice architecture and PLE activities to develop multitenant software as a service (SaaS). (Microservices are small, decentralized,
autonomous services that work together.) The authors used the Weather InSights Environment (WISE) to assess the extent to which their approach supports software reuse during the evolution of multitenant SaaS.

The evaluation of the evolution of multitenant SaaS refactorings of 13 reviews over 384 days showed that the tenants reused an average of 62 percent of the LOC, reducing maintenance effort and increasing scalability. Also, nonreused LOC increased by 200 percent in cases in which one tenant consumed features that other tenants hadn’t consumed. Overall, the study shows this approach’s success when existing functionality can be decomposed into small, independent services that are easier to maintain. You can access this paper at bit.ly/PD_2018_May_7.

References