The Human Factor

Jeffrey C. Carver, Birgit Penzenstadler, Alexander Serebrenik, and Aiko Yamashita

IN THIS ISSUE, we report on papers from the 39th International Conference on Software Engineering and its collocated events. These papers focus on human factors in software engineering, with the last three dealing with open source software. Feedback and suggestions are welcome. In addition, if you try or adopt any of the practices mentioned in the column, please send Jeffrey Carver and the paper authors a note about your experiences.

**Brainstorming and Inclusiveness**

"From Diversity by Numbers to Diversity as Process: Supporting Inclusiveness in Software Development Teams with Brainstorming," by Anna Filippova and her colleagues, discusses how brainstorming affected the satisfaction of minority developers working in group settings. Software development is inherently collaborative, and its success depends on the ability of every team member to contribute. Unfortunately, minority team members, such as women, face greater challenges speaking up, and when they do, their input is often overlooked or dismissed. Brainstorming supports minority team members’ engagement and satisfaction by providing structure for all team members to voice ideas and by encouraging integration of every contribution.

To study brainstorming’s effects, Filippova and her colleagues studied participants from two hackathons that emphasized diversity and involved participants who were minorities in terms of gender, race, age, location, education, and development experience. In both events, participants worked in groups to identify problems and develop solutions. The authors didn’t impose brainstorming concepts on the groups; they used postevent surveys and interviews to identify groups that had used brainstorming.

The results showed that brainstorming had a direct positive impact on process satisfaction and an indirect positive impact on outcome satisfaction. These effects were stronger for the minority participants. That is, brainstorming had a more positive effect on minority participants’ satisfaction with both their group processes and outcomes. This research’s results suggest that brainstorming is a practical mechanism for supporting inclusion in diverse teams.

You can access this paper at bit.ly/PD_Sept_1.

**Dealing with Future Privacy Requirements**

"Privacy Requirements: Present and Future," by Pauline Anthonyksamy and her colleagues, highlights the change in thinking required to handle future privacy requirements. We’re moving from a priori scoping of privacy requirements to open, fluid information flow. This move means that future privacy requirements must cope
with system boundaries that have become more fluid.

Previously, research on privacy requirements focused on the legal aspects of extracting them from legal texts and policies and on verifying them through model checking and the formalization of regulatory rules. This approach makes clear what information is protected. However, it’s also restricting and limited in the information it provides because it assumes a static answer (for example, a user saying “I will never give away my International Mobile Equipment Identity,” even though some services might require that as a verification mechanism). In summary, the current focus is system-driven, syntactic, and attribute-driven.

The move toward systems that collaborate more closely with each other will require privacy requirements to shift to a cross-system (and cross-domain) focus. To properly support this shift, we need transparency, and we need to educate users so that they can make informed decisions about the tradeoffs between the gains from sharing information and the loss of privacy. We also need to shift toward a deeper understanding of privacy that considers derived attributes and synthetic data (for example, location privacy for mobile devices that takes into account speed and the previous known location). This paper is a call to action toward cross-domain privacy analysis, deep privacy, and user empowerment. You can access it at bit.ly/PD_Sept_2.

**Code Reviews and Social Relationships**

“Process Aspects and Social Dynamics of Contemporary Code Review: Insights from Open Source Development and Industrial Practice,” by Amiangshu Bosu and his colleagues, reports on the benefits organizations gain from code reviews, besides improving their software’s quality. To identify those benefits, the authors surveyed open source developers and Microsoft developers who had participated in 30 or more code reviews.

The code reviews had an important impact on the social relationships in an organization. Specifically, the second most common reason why code reviews were important (behind maintainability issues) was knowledge sharing. Respondents also indicated that the two most important factors in deciding whether to accept a code review request were the code author’s reputation and the reviewer’s prior relationship with the author. Finally, respondents indicated that the quality of the code under review (good or bad) affected the reviewer’s perception of the code author’s abilities and personal characteristics.

Bosu and his colleagues also identify similarities and differences between the open source developers’ and Microsoft developers’ experiences. Overall, the two groups of developers gave similar responses regarding how code review affected social relationships, although they emphasized different aspects. The open source developers focused on relationship building, whereas the Microsoft developers focused on knowledge dissemination. You can access this paper at bit.ly/PD_Sept_3 and the corresponding slides at bit.ly/PD_Sept_3a.

**Understanding Open Source Licenses**

“Do Software Developers Understand Open Source Licenses?,” by Daniel Almeida and his colleagues, examines how well developers understood the implications of using various open source licenses, either alone or in combination. In response to this paper, Peter Smith, principal software engineer at ACL, told us that “the Internet makes it incredibly easy for developers to integrate open source packages into their own software.” The use of such packages requires that organizations understand the licenses that govern components’ use. Because a license’s applicability depends on how a particular component is used, licensing issues can’t be delegated exclusively to managers or legal departments.

In a survey of 375 developers, Almeida and his colleagues posed various hypothetical scenarios (for example, “Can Sue make her application commercially available under MPL [Mozilla Public License] 2.0 if it makes use of a library distributed under GNU GPL [General Public License] 3.0?”). They then compared the developers’ responses to those of an intellectual-property lawyer who was an expert on open source licenses.

The developers understood the applicability and implications of single licenses, even when used in complex scenarios. Conversely, they had difficulty identifying the correct interactions in scenarios involving multiple licenses. Smith also said that “this [lack of understanding of multiple licenses] is important, especially as I’ve witnessed expensive software ‘rewrites’ as companies come to realize their mistake.” So, the need exists for tools that help developers identify license inconsistencies and that suggest potential refactorings. You can access this paper at bit.ly/PD_Sept_4.
Managing Docker

“An Empirical Analysis of the Docker Container Ecosystem on GitHub,” by Jürgen Cito and his colleagues, presents their analysis of a large sample of projects in the Docker ecosystem. The authors aimed to understand the ecosystem’s evolution and quality issues and determine guidelines for tool support to improve quality and drive ecosystem change. Docker, which lets developers package an application and its dependences as a self-contained unit, has become the de facto standard for software containerization.

After analyzing more than 70,000 GitHub Dockerfiles, Cito and his colleagues made these observations.

- **The Docker images were quite large.** By using a heavyweight OS as their base image, most Dockerfiles defeated containerization’s original purpose, which is to lower virtualization’s footprint. Reducing the image size could tremendously affect the build time for continuous integration or deployment in an orchestration system such as Kubernetes.
- **Build failures took too long.** Approximately one-third of the builds failed, taking an average of 90.5 seconds to notify developers of the failure. Compared with the results for Java projects built on Travis CI, this delay could be problematic if building the Docker image was part of the continuous-deployment or continuous-delivery pipeline.
- **Version pinning was problematic.** Over 25 percent of the quality issues arose from missing version information for dependencies.
- **Unstructured dependencies drove recurring changes.** Most changes in Docker resulted from dependencies that were stored in an unstructured manner.

The authors suggest these remedies:

- Reduce the image size by using a lighter-weight OS as the base.
- Integrate quality checks to issue version-pinning warnings as part of the build process.
- Use better abstractions to resolve the intricacies of different package managers.

You can access this paper at bit.ly/PD_Sept_5.

**References**