Editor: **Giuliano Antoniol**
Polytechnique Montréal

Editor: **Steve Counsell**
Brunell University

Editor: **Phillip Laplante**
Pennsylvania State University

Leaders of Tomorrow on the Future of Software Engineering

A Roundtable

Nine rising stars in software engineering shared their perspectives on the future of software engineering at the Leaders of Tomorrow Symposium at the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering. Here they describe how software engineering research will evolve, highlighting emerging opportunities and groundbreaking solutions. They predict the rise of end-user programming, the monitoring of developers through neuroimaging and biometrics sensors, analysis of data from unstructured documents, the mining of mobile marketplaces, and changes to how we create and release software. Enjoy! *—Massimiliano Di Penta, Ahmed E. Hassan, and Thomas Zimmermann, symposium chairs*

In the Future, Everyone Will Be a Programmer for 15 Minutes

Felienne Hermans

IN THE PAST, software engineering has focused mainly on professional developers: people employed to build, test, and maintain software. However, many people program not as a job but as a means to an end. These workers, often called *end-user programmers*, write queries, small scripts, or spreadsheets to support their daily jobs. The number of end-user programmers in the US alone is estimated at 11 million, compared to only 2.75 million professional programmers[1] Given the popularity of introductory-programming initiatives such as code.org and end users' widespread adoption of programming languages such as Python and R, we can assume that a new generation of professionals will emerge who can perform some programming to reach their professional goals.

So, supporting end-user programmers in building reliable software will be an even bigger opportunity for research in the near future.

Because end-user programmers create programs to support their own domain of expertise, the programs they create are, by definition, not meant for others to use. However, a core problematic aspect of end-user programming is that sometimes the created artifacts evolve from personal solutions to programs used by many colleagues. When that happens, the end users, who often didn't expect this situation and are unprepared for it, must suddenly take on the challenges of professional developers, such as testing and maintaining their creations.

0740-7459/16/$33.00 © 2016 IEEE

This problem offers several interesting research directions for software engineering. When the boundaries between professional and end-user programmers blur, so will the boundaries between the traditional IDE for professionals and end-user tools such as spreadsheets, LabVIEW, or Matlab. Equipping those tools with techniques for testing, measuring, and maintaining artifacts will be an exciting new challenge to help everyone be a programmer, even if for just 15 minutes.

Reference

1. C. Scaffidi, M. Shaw, and B.A. Myers, "Estimating the Numbers of End Users and End User Programmers," *Proc. 2005 IEEE Symp. Visual Languages and Human-Centric Computing* (VL/HCC 05), 2005, pp. 207–214.

FELIENNE HERMANS is an assistant professor of software engineering at the Delft University of Technology. Contact her at f.f.j.hermans@tudelft.nl.

Tailoring Tool Support to Developers' Needs

Janet Siegmund

SOFTWARE IS DESIGNED, implemented, and maintained by human developers. Their minds have inherent limitations that can be cognitive, such as in terms of working memory or programming skills, or affective, such as tiredness or irritability. To produce high-quality software, developers need support to overcome these limitations. A future challenge of software-engineering research will be to provide such support that's tailored to the task at hand and takes into account developers' state of mind.

> To understand developers in detail, researchers have adopted neuroimaging methods.

To this end, researchers need to understand developers in detail. To succeed, researchers have adopted neuroimaging methods to monitor developers' state of mind. For example, my colleagues and I employed functional magnetic resonance imaging to observe developers as they comprehended source code.[1] Igor Crk and Timothy Kluthe used electroencephalography to quantify programmer expertise,[2] and Takao Nakagawa and his colleagues used functional near-infrared spectroscopy to assess developers' cognitive loads.[3] Thomas Fritz and his colleagues combined psychophysiological measures to predict perceived task difficulty for developers.[4]

Measurement devices will become cheaper, smaller, and wearable, such that developers can wear them for their everyday tasks, similar to the activity trackers many of us wear today. So, we'll be able to tell, for instance, when

- a developer's cognitive load reaches a critical level and he or she needs extra support,
- a developer is in a state of high concentration and shouldn't be disturbed, or
- a developer is tired and shouldn't work on safety-critical tasks.

With such monitoring, IDEs can support developers by, for example, displaying or hiding additional information.

References

1. J. Siegmund et al., "Understanding Understanding Source Code with Functional Magnetic Resonance Imaging," *Proc. 36th Int'l Conf. Software Eng.* (ICSE 14), 2014, pp. 378–389.
2. I. Crk and T.X. Kluthe, "Toward Using Alpha and Theta Brain Waves to Quantify Programmer Expertise," *Proc. 36th Ann. Int'l Conf. IEEE Eng. in Medicine and Biology Soc.*, 2014, pp. 5373–5376.
3. T. Nakagawa et al., "Quantifying Programmers' Mental Workload during Program Comprehension Based on Cerebral Blood Flow Measurement: A Controlled Experiment," *Companion Proc. 36th Int'l Conf. Software Eng.* (ICSE 14), 2014, pp. 448–451.
4. T. Fritz et al., "Using Psycho-physiological Measures to Assess Task Difficulty in Software Development," *Proc. 36th Int'l Conf. Software Eng.* (ICSE 14), 2014, pp. 402–413.

JANET SIEGMUND is a postdoc in human factors in software engineering at the University of Passau. Contact her at siegmunj@fim.uni-passau.de.

Biometric Sensors Will Boost Developer Productivity

Thomas Fritz

PRODUCING GREAT SOFTWARE as fast as the market demands requires great, productive developers. Yet, what does it mean for developers to be productive, and how can we best help them be productive? To answer these questions, software engineering researchers have been and still are looking predominantly at software developers' output, such as applications, source code, or test cases. This focus on outputs misses an essential part of software development: the individual developer.

Biometric (psychophysiological) data offers the opportunity to better understand the individual developer and what he or she experiences while working. In particular, biometric sensors let us measure in real time a developer's physiological features that can be linked to his or her cognitive and emotional states. For instance, changes in pupil diameter sensed through an eye tracker can tell us something about the mental workload a change task imposes.

My vision is to integrate biometric sensing into a developer's work and leverage the data to boost productivity and provide better, real-time support—for instance, by reducing the number of interruptions at inopportune moments or preventing errors from entering the code. Currently, researchers are only beginning to analyze and understand how to interpret and use such data. Many challenges remain, such as sensor invasiveness, the privacy concerns arising from collecting and using such personalized data, or the real-time cleaning and analysis of fine-grained biometric data. Yet, measuring aspects of an individual developer rather than his or her output will provide a tremendous opportunity to amplify the human in the process and revolutionize software development.

THOMAS FRITZ is an assistant professor in the University of Zurich's Department of Informatics. Contact him at fritz@ifi.uzh.ch.

Mining Unstructured Data

Gabriele Bavota

UNSTRUCTURED DATA REFER to information that isn't organized or stored according to a precise schema or structure. Mining unstructured data (MUD) has become popular in software engineering, owing mainly to the high amount of such data in software repositories (for example, issue trackers' discussions and code comments) and, more generally, on the Web (for example, question-and-answer websites). Although researchers have already exploited MUD to support an incredibly large number of software engineering tasks (for example, artifact summarization and bug triage), much more can be done.

First, when talking about MUD in software engineering, people tend to think of textual information spread among software repositories. However, unstructured data are also, by definition, multimedia content, such as images and videos. Our community almost ignores these data, but they embed information that complements what's available in textual data. For example, a video tutorial can show how a developer interacts with an IDE, something difficult to embed in a textual tutorial. Mining such content could open whole new research directions.

Second, MUD techniques are often applied out of the box to support software engineering tasks. For instance, researchers have used text summarization to provide an overview of the main responsibilities in a class. However, the generated summaries aren't customized on the basis of who will read them and which task to support. A newcomer in charge of writing the class documentation might need information that differs from what an experienced developer testing the class would need. So, summaries that are consumer-related (who will consume them) and task-related (what they'll be used for) could lead to the generation of more useful pieces of knowledge. Such summarization is just one of the MUD applications that could be rethought in a consumer- or task-related fashion.

What else to expect? Given the growing amount of unstructured data out there, definitely a lot!

Biometric data offers the opportunity to better understand the individual developer.

GABRIELE BAVOTA is an assistant professor at the Free University of Bozen-Bolzano, Faculty of Computer Science, Software Engineering Research Group (SERG). Contact him at gabriele.bavota@unibz.it.

Mining Mobile-App Markets

Meiyappan Nagappan

MOBILE APPS MIGHT differ considerably from traditional software owing to the varieties of mobile devices. This difference might be why interest in mobile apps has increased among both academic and industrial researchers. Academic researchers have shown continued interest in solving classic software engineering problems such as test input generation, software quality, and software reuse for mobile apps. The next logical step is to broaden these solutions to work on cross-platform apps.

On the industrial side, data analytics companies provide mobile-app developers with tools for collecting and analyzing both runtime and user data. These tools let developers know when and how users are using or not using an app. Knowing this, developers can focus on the features that bring in the most revenue.

Another key difference between mobile apps and traditional software is that mobile apps are distributed through centralized app markets that make it easy for developers to release mobile apps. So, the next big idea in software engineering research for mobile apps is mining mobile-app markets.

Through mobile-app markets, researchers have centralized access to not only the apps themselves but also the metadata surrounding them (such as release notes and the security permissions needed) and user feedback (through user reviews and ratings). Some advances in this area have already occurred—for example, understanding the relationship between ratings and downloads and understanding how ads affect both developers and users. Yet, numerous challenges (including mining data continuously in an evolving market and storing and sharing the data) and opportunities (including determining what features to add or how to make an app more competitive) remain to be explored.

MEIYAPPAN (MEI) NAGAPPAN is an assistant professor in the Rochester Institute of Technology's Department of Software Engineering. Contact him at mei@se.rit.edu.

Energy-Aware Software Development

Abram Hindle

THE FUTURE OF software engineering is energy-aware software development. Software sustainability will become a buzzword. Managers will request that their developers address software energy concerns. Programmers, now responsible for sustainable development, will rely on cloud energy measurement services. IDEs will warn programmers of any dangers to software energy consumption that their changes pose to their product. Their product's energy consumption profile will be tracked over time to enable careful regression testing and analysis. Third parties will measure, rank, and certify the end product according to "green star": Software Application Energy Consumption Ratings (SAECRs), an Energy Star derivative for ranking software energy consumption.[1]

Users will study app energy ratings, rank apps by their energy efficiency, and use such rankings to make app-purchasing decisions. Researchers will employ methodologies (green mining) that carefully evaluate energy consumption by measuring multiple versions of the system under test. This will aim to improve generality and avoid erroneous attribution of energy consumption to nonrepresentative code. Researchers will collaborate on a large shared corpus of software energy runs, enabling development of both general and specific models. These models will be employed by energy-aware plugins in IDEs.

So, the future of software engineering will be energy aware among all stakeholders: users, programmers, managers, and product owners.

Reference

1. C. Zhang A. Hindle, and D.M. German, "The Impact of User Choice on Energy Consumption," *IEEE Software*, vol. 31, no. 3, 2014, pp. 69–75; http://softwareprocess.es/2015/user_test-ieee-software-web.pdf.

ABRAM HINDLE is an assistant professor in the University of Alberta's Department of Computing Sciences. Contact him at hindle1@ualberta.ca.

Software Quality Assurance 2.0: Proactive, Practical, and Relevant

Yasutaka Kamei

FUMIO AKIYAMA FIRST attempted to find the relationship between

size-based metrics and the number of bugs by analyzing actual development histories in 1971.[1] Since then, researchers have devised many ways to help developers with software quality assurance (SQA). For example, my colleagues and I have proposed techniques that automatically identify software changes that have a high risk of inducing defects, by mining the risk that prior changes caused. I see SQA research evolving in at least three directions.

First, SQA approaches must be more proactive. Many of them have been reactive—they determine only what will happen after release. We can devise tools that not only predict risky areas but also generate tests (and possibly fixes) for them. We can also devise techniques that warn developers, even before they modify the code, that they're working with risky code that has had specific types of defects in the past.

Second, we should evaluate our SQA techniques in practical settings instead of using only traditional measures such as precision and recall. Many SQA studies show that their approaches are 5 percent better than a baseline in terms of precision and recall. However, what does such precision mean for developers? Future SQA research needs to show what that precision means and whether it's practically effective.

Finally, we should tackle the challenge that new markets raise. One such market is mobile apps. We use smartphones every day and update apps from online stores (such as Google Play). Mobile apps play a significant role in our daily life, and their characteristics differ from those of conventional applications studied in the past. My colleagues and I anticipate new markets to be an area of significant growth.

Reference

1. Fumio Akiyama, "An Example of Software System Debugging," *Proc. IFIP Congress 71*, vol. 1, 1971, pp. 353–359.

YASUTAKA KAMEI is an associate professor in Kyushu University's Graduate School and Faculty of Information Science and Electrical Engineering. Contact him at kamei@ait.kyushu-u.ac.jp.

Software Engineering for the Web

Ali Mesbah

THE WEB PROVIDES a unique software engineering platform. Its benefits include instantaneous upgrades for all users and "write once, run anywhere" through universal access and execution from any Internet-connected device. Because of these benefits, any application that can be written as a Web application eventually will be. Web-based services such as GitHub and Stack Overflow have already revolutionized how developers write software. This trend will likely continue.

> Any application that can be written as a Web application eventually will be.

The Web is still in its infancy and is continuously and rapidly evolving. Unlike traditional software, Web apps are heterogeneous (JavaScript, CSS, HTML, and so on). The dynamic interdependencies between these languages, and their distributed asynchronous client-server nature, pose many challenges for developers. This is where software engineering research can play an important role.

As Web languages such as JavaScript become more prominent, IDE support becomes essential for developing and maintaining large-scale applications in practice. However, current software analysis techniques have serious limitations in helping developers understand, write, analyze, and maintain Web code. Current research on Web app code smell detection, refactoring support, and code completion is scarce, and industrial tools available to Web developers have limited capabilities. As it stands today, we're not even able to extract proper control-flow and call graphs from Web code.

A promising research direction is the ability to handle multiple languages in Web analysis. Recent examples of such research include inferring cross-language slicing[1] and detecting inconsistencies between JavaScript and DOM (Document Object Model).[2] We need interlanguage analyses that can handle Web application code. Also, hybrid approaches that combine static and dynamic techniques will probably prove more useful for this domain.

References

1. H.V. Nguyen, C. Kästner, and T.N. Nguyen, "Cross-Language Program Slicing for Dynamic Web Applications," *Proc. 10th Joint Meeting Foundations of Software Eng.* (ESEC/FSE 15), 2015, pp. 369–380.
2. F. Ocariza, K. Pattabiraman, and A. Mesbah, "Detecting Inconsistencies in JavaScript MVC Applications," *Proc. 2015 ACM/IEEE Int'l Conf. Software Eng.* (ICSE 15), 2015, pp. 325–335.

ALI MESBAH is an assistant professor in the University of British Columbia's Electrical and Computer Engineering department. Contact him at amesbah@ece.ubc.ca.

Roll Your Own Release (RYOR)

Bram Adams

RELEASE ENGINEERING involves

- integrating developers' individual code changes into a coherent whole,
- automatically building and testing these changes, and
- deploying and releasing official releases to their target audience.

Over the last five years, release engineering has been revolutionized by a move toward ever shorter cycle times. Major companies such as Google, Facebook, and Mozilla have been reducing the time between consecutive releases from months to weeks, days, or even hours. Such short, regular releases bring features to users faster, gathering feedback more quickly.

Whereas companies traditionally have relied on a central team of release engineers and their automation to coordinate release-engineering activities, companies increasingly are moving toward a roll-your-own release (RYOR) strategy. For example, companies such as Netflix give their developers the power (tools) to automatically build and test a new feature in an environment similar to the production environment. The developers can also, on their own initiative, perform canary releases to certain groups of users. RYOR is a necessity to fully enable new paradigms such as microservices, which decompose a system into many small, independent (and interdependent) Web services.

Now, where's the catch? Well, traditional release engineers play a unique role. They have a global view of a system's architecture and quality and have a knack for identifying risky changes that could jeopardize an upcoming release. On the other hand, developers are biased toward their own features. To neutralize this natural bias, RYOR requires help from researchers—for example, to

- automatically predict integration conflicts and the effort to resolve them,
- identify backward-compatibility problems,
- profile and optimize the build system and tests, or
- determine the best deployment and release strategy for a certain feature.

To top things off, developers need this support ... inside their IDE!

BRAM ADAMS is an assistant professor at Polytechnique Montréal, where he heads the Lab on Maintenance, Construction, and Intelligence of Software. Contact him at bram.adams@polymtl.ca.