Software Analytics: What's Next?

Tim Menzies

North Carolina State University, USA

Thomas Zimmermann

Microsoft Research, USA

Knowing what factors control software projects is very useful since humans may not understand those factors. Developers sometimes develop their own ideas about "good" and "bad" software, based on just a few past projects. Using software analytics, we can correct those misconceptions.

Software analytics lets software engineers learn about AI techniques. Once learned, those engineers can now build and ship innovative AI tools. That is, software analytics is the training ground for the next generation of AI-literate software engineers.

KEYWORDS: Software, software analytics, data mining

Tweets:

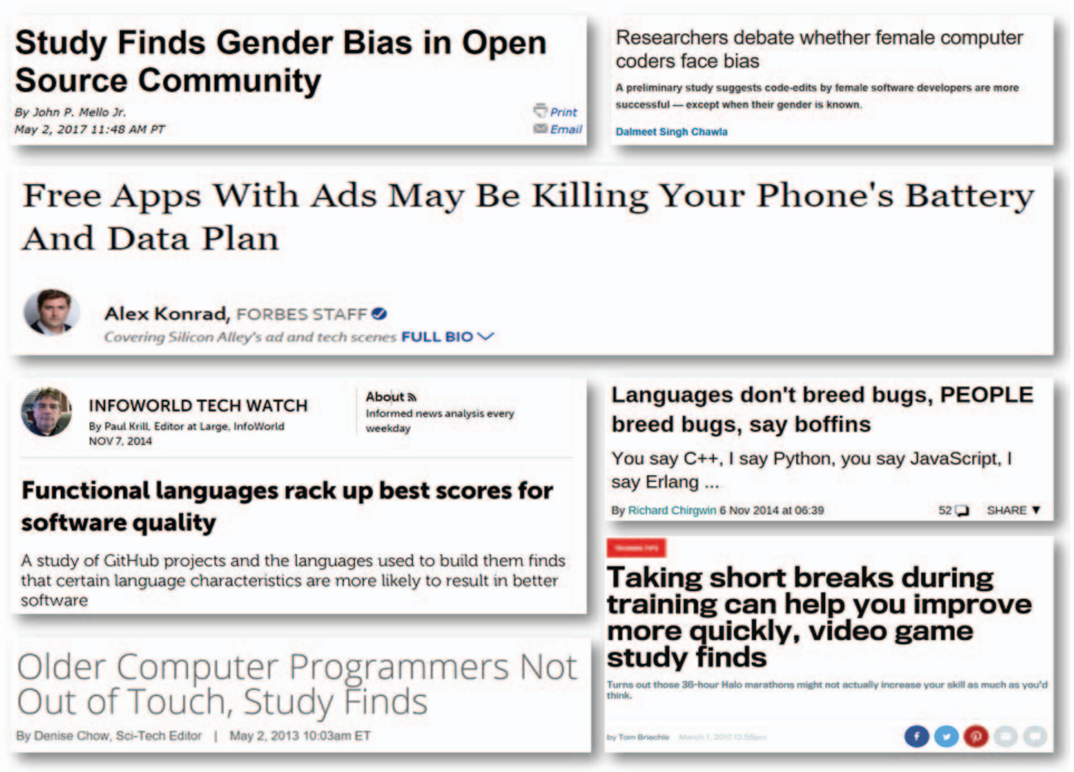
1. What do you know about your projects? Have you checked? #softwareanalytics has so many surprises for you.
2. In the data-rich 21st century, #softwareanalytics lets us retest the truisms of the past.
3. "The realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs." -- Maurice Wilkes. Let #softwareanaytics guide your bug removal.
4. Microsoft has 1000+ employees using software analytics. They do things that used to be called Phd-level research. Now we call that "standard operating procedure".
5. Software analytics: the training ground for the next generation of AI-literate software engineer.

Digital Object Identifier 10.1109/MS.2018.290111035 0740-7459/$26.00 2018 IEEE

1. Introduction

Software development is a complex process. Human developers may not always understand all the factors that influence their projects. Software analytics is an excellent choice for discovering, verifying, and monitoring the factors that affect software development

Software analytics distills large amounts of low-value data into small chunks of very high-value information[1]. Those "chunks" can reveal what factors matter the most for software projects. For example, Figure 1 lists some of the more prominent recent insights learned in this way.



Study Finds Gender Bias in Open Source Community
By John P. Mello Jr.
May 2, 2017 11:48 AM PT

Researchers debate whether female computer coders face bias
A preliminary study suggests code-edits by female software developers are more successful — except when their gender is known.
Dalmeet Singh Chawla

Free Apps With Ads May Be Killing Your Phone's Battery And Data Plan
Alex Konrad, FORBES STAFF
Covering Silicon Alley's ad and tech scenes FULL BIO

INFOWORLD TECH WATCH
By Paul Krill, Editor at Large, InfoWorld
NOV 7, 2014
About
Informed news analysis every weekday

Functional languages rack up best scores for software quality
A study of GitHub projects and the languages used to build them finds that certain language characteristics are more likely to result in better software

Languages don't breed bugs, PEOPLE breed bugs, say boffins
You say C++, I say Python, you say JavaScript, I say Erlang ...
By Richard Chirgwin 6 Nov 2014 at 06:39

Older Computer Programmers Not Out of Touch, Study Finds
By Denise Chow, Sci-Tech Editor | May 2, 2013 10:03am ET

Taking short breaks during training can help you improve more quickly, video game study finds
Turns out those 36-hour Halo marathons might not actually increase your skill as much as you'd think.
by Tom Briechle

Figure 1: Software analytics, in the press. From *Linux Insider, Nature, Forbes, InfoWorld, The Register, Live Science, and Men's Fitness* on surprising Software Analytics findings

Software analytics lets us "trust, but verify" human intuitions. If someone claims that "this or that" is important for a successful software project, analytics lets us treat that claim as something to be verified (rather than a sacred law that cannot be questioned). Also, once verified, analytics can act as a monitor to continually check if "this or that" is now overcome by subsequent developments.

Knowing what factors control software projects is very useful since, sometimes, human do not understand those factors. Anthropologists studying software projects[2] warn that developers usually develop their personal ideas about good and bad software

based on just a few past projects. All too often, these ideas are assumed to be apply to all projects, rather than just the few seen lately. This can lead to too much reuse of to many old, and now outdated, ideas. A recent study [3] of 564 software developers found that:
"(a) programmers do indeed have very strong beliefs on certain topics; (b) their beliefs are primarily formed based on personal experience, rather than on findings in empirical research; (c) beliefs can vary with each project, but do not necessarily correspond with actual evidence."

The good news is that, using software analytics, we can correct those misconceptions (for examples, see Figure 2). For example, after examining a project, certain coding styles could be seen as more bug prone (and should be avoided).

Prior to the 21st century, researchers often only had access to data from one or two projects. This meant theories of software development were built from limited data. But in the data-rich 21st century, researchers have access to all the data they need to test the truisms of the past. And what they've found is most surprising:

1. In stark contrast to the assumptions of much prior research, pre-and post-release failures are not connected [1].
2. Static code analyzers perform sinno better than simple statistical predictors [2].
3. The language construct "goto", as used in contemporary practice, is rarely considered harmful [3].
4. Strongly typed languages are not associated with successful projects [4].
5. Developer beliefs are rarely backed by any empirical evidence [5].
6. Test-driven development not any better than "test-last" [6].
7. Delayed issues are not exponentially more expensive to fix [7].
8. Most "bad smells" should not be fixed [8.9].

References

1. Norman E. Fenton and Niclas Ohlsson. 2000. Quantitative Analysis of Faults and Failures in a Complex Software System. IEEE Trans. Softw. Eng. 26, 8 (August 2000), 797-814. DOI=http://dx.doi.org/10.1109/32.879815
2. Foyzur Rahman, Sameer Khatri, Earl T. Barr, and Premkumar Devanbu. 2014. Comparing static bug finders and statistical prediction. In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014).
3. Meiyappan Nagappan, Romain Robbes, Yasutaka Kamei, Éric Tanter, Shane McIntosh, Audris Mockus, and Ahmed E. Hassan. 2015. An empirical study of goto in C code from GitHub repositories. FSE'15.
4. [Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in github. FSE'15.
5. Prem Devanbu, Thomas Zimmermann, and Christian Bird. Belief & evidence in empirical software engineering. In Proceedings of the 38th International Conference on Software Engineering, pages 108–119. ACM, 2016
6. D. Fucci, H. Erdogmus, B. Turhan, M. Oivo and N. Juristo, "A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last?," in IEEE Transactions on Software Engineering, 43(7), pp. 597-614, 1 2017.
7. Tim Menzies, William Nichols, Forrest Shull, and Lucas Layman. 2017. Are delayed issues harder to resolve? Revisiting cost-to-fix of defects throughout the lifecycle. Empirical Softw. Engg. 22, 4 (August 2017), 1903-1935.
8. Krishna, R., Menzies, T., & Layman, L. (2017). Less is more: Minimizing code reorganization using

XTREE. Information and Software Technology, 88, 53-66.

9. Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. . FSE'12 .

Figure 2: Why we need software analytics: much of our prior "knowledge" about software engineering now needs extensive revision.

The Art and Science of Analyzing Software Data
Christian Bird, Tim Menzies, Thomas Zimmermann
Perspectives on Data Science for Software Engineering
Edited by Tim Menzies, Laurie Williams, Thomas Zimmermann
Sharing Data and Models in Software Engineering
Tim Menzies, Ekrem Kocaguneli, Leandro Minku, Fayola Peters and Burak Turhan
IEEE Software
SOFTWARE ANALYTICS: SO WHAT?

FIGURE 3: Researchers in software analytics have collaborated extensively to record the state of the art in software analytics. For more on these books, please see goo.gl/ZeyBhb. For other material, see the July/September 2013 special issues on software analytics from IEEE Software.

There are many ways to distill that data including quantitative methods (that often use data mining algorithms) and/or qualitative methods (that use extensive human feedback to guide the data exploration). Previously we have offered extensive surveys of those methods (see Figure 3). The goal of this article is to update those surveys and offer some notes on current and future directions in software analytics.

2. Some History

As soon as people started programming, it became apparent that programming was an inherently buggy process. As recalled by Maurice Wilkes[4], speaking of his programming experiences from the early 1950s: "It was on one of my journeys between the EDSAC room and the punching equipment that 'hesitating at the angles of stairs' the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs." It took several decades to gather the experience required to quantify any kind size/defect relationship. In 1971, Fumio Akiyama[5] described the first known "size" law, saying the number of defects D was a function of the number of LOC. In 1976, Thomas McCabe argued that the number of LOC was less important than the complexity of that code[6].' He argued that code is more likely to be defective when his "cyclomatic complexity" measure was over 10.

Not only is programming an inherently buggy process, it's also inherently difficult. Based on data from dozens of projects, Barry Boehm[7] proposed in 1981 an estimator for development effort (that was exponential on program size) using a set of "effort multipliers” M_i inferred from the current project:

$$\text{effort} = a \star \text{KL0C}^{b} \star \prod M_i,$$

where $2.4 \le a \le 3$ and $1.05 \le b \le 1.2$. While useful in their home domains, it turns out that the results of Akiyama, McCabe and Boehm required extensive modification to work on other projects. One useful feature of the current generation of data mining algorithms is that it is now relatively fast and simple to learn for example defect or effort predictors for other projects, using whatever attributes they have available.

3. Current Status

Thousands of recent research papers all make the same conclusion: data from software projects contain useful information that can be found by data mining algorithms. We now know that with modest amounts of data collection, it is possible to:

1. Build powerful recommender systems for software navigation or bug triage; or
2. Make reasonably accurate predictions about software development effort and defects
3. For more details on that and other work, see material in Figure 3.

Those predictions can rival those made by much more complex methods (such as static code analysis tools, which can be hard to maintain). Various studies have shown the commercial merits of this approach; e.g. for one set of projects, software analytics could predict 87 percent of code defects, decrease inspection efforts by 72 percent and hence, reduces post-release defects by 44 percent[8]. Better yet, when multiple models are combined using ensemble learning, very good estimates can be generated (very low variance in their predictions) and these predictors can be incrementally updated to handle changing conditions[9]. Also, when is there too much data for a manual analysis, it is possible to automatically analyze data from hundreds to thousands of projects using software analytics[10].

Recent research explains why software analytics has been so successful: artifacts from software projects have an inherent simplicity. As Abram Hindle and colleagues[11] explain:

“Programming languages, in theory, are complex, flexible and powerful, but the programs that real people actually write are mostly simple and rather repetitive, and thus they have usefully predictable statistical properties that can be captured in statistical language models and leveraged for software engineering tasks.”

For example, here are just some of the many applications in this growing area on research:

1. patterns in the tokens of software can recognize code that is unexpected and bug prone;
2. sentiment analysis tools can gauge the mood of the developers, just by reading their issue comments[12.]
3. clustering tools can explore complex spaces like StackOverflow to automatically detect related questions[13].

4. But What’s Next?

when we reflect over the last decade, several new trends stand out. For example, consider the rise of the data scientist in industry. Many organizations now pay handsomely to hire large teams of data scientists. For example, at the time of this writing, there are 1000+ Microsoft employees exploring project data using software analytics. These teams are performing tasks that, a decade, would have been called cutting-edge research. But now we call that work “standard operating procedure”.

Several new application areas have emerged such as green engineering. Fancy cellphones become hunks of dead plastic if they run out of power. Hence, taming power consumption is now a primary design concern. Software analytics is a tools for monitoring, and altering, power usage profiles[14].

Another emerging area is social patterns in software engineering. Our society divides users into different groups. By adopting the perspectives of different social groupings, it is possible to build better software[15]. Models built from social factors (like how rarely someone updates part of the code) can be more effective for predicting code quality prediction than code factors (like function size or number of arguments). For example, when studying software built in multiple countries, a good predictor for bugs is the complexity of the organizational chart (least bugs are introduced when people working on the same functions report to the same manager, even if they are in different countries[16]).

Software analytics also studies the interactions of developers using biometric sensors. Just as we mine software (and the social processes that develop them), so too can we mine data, collected at the millisecond level, from computer programmers. For example, using eye-tracking software or sensors for skin and brain activity, software analytics can determine what code is important or most difficult for developers[17].

When trading-off different goals, the data mining algorithms used in conventional software engineering often use hard-wired choices. These choices may be irrelevant or even antithetical to the concerns of the business users who are funding the analysis. For example, one way to maximize accuracy in unbalanced data sets (where, say, most of the examples are not defective) is to obsess on maximizing the true negative score. This can mean that, by other measures such as precision or recall, the learner fails. What is required are “learners” that guide the reasoning according to user-specified goal. Such “search-based software engineering” tools can implement other useful tasks such as automatically tuning the control parameters of a data mining algorithm[18] (which, for most developers, is a black art).

Other important trends show the maturity of software analytics. For decades now, there has been extensive discussions about reproducibility of software research-- much of it having the form “wouldn’t it be a good idea”. We can report here that, at least for quantitative software analytics based on data mining algorithms, such reproducibility is common practice. Many conferences in software engineering reward researchers with “badges”, if they place their materials online (see tiny.cc/acmbadges). Accordingly, researchers place their scripts and data in Github. Some even register those repositories with tools like Zenodo, in order to obtain unique and eternal digital object identifiers for their artifacts. For an excellent pragmatic discussion on organising repositories in order to increase reproducibility, see “Good Enough Practices for Scientific Computing”[19].

Another challenge are the security and privacy issues raised by software analytics. Legislation being enacted around the world (e.g., the European General Data Protection Regulation, GDPR) means that vendors collecting data will face large fines unless they address privacy concerns. One way to privatize data is to obfuscate it (a.k.a. mutate it) ito reduce the odds that an malevolent agent can identify who or what generated that data. But this raises a problem- the more we mutate data, the harder it becomes to learn an effective model from it. Recent results suggest that sometimes this problem can be solved[20], but much more work is required on how to share data without compromising confidentiality.

And every innovation also offers new opportunities. There is a flow-on effect from software analytics to other AI tasks outside of software engineering. Software analytics lets software engineers learn about AI techniques, all the while practicing on domains they understand (i.e. their own development practices). Once someone can apply data mining algorithms to their data, they can now build and ship innovative AI tools. While sometimes those tools solve SE problems (e.g. recommending what to change in source code), they can also be used on a wider range of problems. That is, we see software analytics as the training ground for the next generation of AI-literate software engineer working on applications such as image recognition; large-scale text mining; autonomous cars or drones; etc.

Finally, with every innovation come new challenges. In the rush to explore a promising new technology, we should not forget the hard won lessons of other kinds of research in SE. This point is explored more in Figure 4.

5. Better Software Analytics equals Better Science?

We end with this frequently asked question: “What is the most important technology a newcomer should learn to make them better at data science (in general) and software analytics (in particular)?” To answer this question, we need a workable definition of “science” which we take to mean a community of people collecting and curating and critiquing a set of ideas. In this community, everyone is doing each other the courtesy to try and prove this shared pool of ideas.

By this definition, most data science (and much software analytics) is not science. Many developers use software analytics tools to produce conclusions and that’s the end of the story. Those conclusions are not registered and monitored. There is nothing that checks if old conclusions are now out of date (e.g., using anomaly detectors). There are no incremental revisions that seek minimal changes when updating old ideas.

If software analytics really wants to be called a “science”, then it needs to be more than just a way to make conclusions about the present. Any scientist will tell you that all ideas should be checked, rechecked, and incrementally revised. Data science methods like software analytics should be a tool for assisting in complex discussions about on-going issues.

Which is a long-winded way of saying that the technology we most need to better understand software analytics and data science is… science

SHORT BIO

Tim Menzies (Ph.D., UNSW, http://menzies.us) is a full Professor in CS at North Carolina State University. Currently, he works on software engineering, automated software engineering, and foundations of software science. In the past, he has been a lead researcher on projects for NSF, NIJ, DoD, NASA, USDA, as well as joint research work with private companies. For 2002 to 2004, he was the software engineering research chair at NASA's software Independent Verification and Validation Facility.

Thomas Zimmermann (Ph.D., Saarland University, http://thomas-zimmermann.com/) is a Senior Researcher at Microsoft where analyzes data for a living and collects unicorns in his office. Currently, he works on productivity of software developers and data scientists at Microsoft. In the past, he analyzed data from digital games, branch structures, and bug reports. He is a Distinguished Member of the ACM and a Senior Member of IEEE and the Computer Society.

Not everything that happens in software development is visible in a software repository [1] which means that no data miner can learn a model of these subtler factors. Qualitative case studies and ethnographies that can give comprehensive insights to particular cases and surveys that offer industry insights beyond what can be measured from software repositories. Due to their manual workload, such methods have issues with scaling up to cover many projects. Hence we see a future where software is explored in alternating cycles of qualitative and quantitative methods (where each can offer surprises that prompt further work in the other [2]).

Another issue with software analytics is the problem of "context" [3]. That is, models learned for one project may not apply to another. Much progress has been made in recent years in learning context-specific models [4], or building operators to transfer data/models learned from one project to another [5]. But this is a clearly an area that needs more work.

Yet another other issue is how "actionable" are our conclusions. Software analytics aims at obtain actionable insights from software artifacts that help practitioners accomplish tasks related to software development, systems. For software vendors, managers, developers and users, such comprehensible insights are the core deliverable of software analytics [6]. Sawyer et al. comment that actionable insight is the key driver for businesses to invest in data analytics initiatives [7]. But not all the models generated via software analytics are "actionable"; i.e. they cannot be used to make effective changes to a project. For example, software analytics can deliver models that humans find it hard to read or understand or audit. Examples of those kinds of models include anything that uses complex and/or arcane mathematical internal forms such as deep learning neural networks, Naive Bayes classifiers or Random forests. Hence some researchers in this field explore methods to re-express complex models in terms of simpler ones [8].

1. Aranda, J., & Venolia, G. (2009, May). The secret life of bugs: Going past the errors and omissions in software repositories. ICSE'09
2. Di Chen, Kathryn T. Stolee, Tim Menzies. Replicating and Scaling up Qualitative Analysis using Crowdsourcing: A Github-based Case Study. 2018. arXiv:1702.08571
3. Kai Petersen, Claes Wohlin. 2009. Context in industrial software engineering research. In *ESEM'09.*
4. Tore Dybå, Dag IK Sjøberg, and Daniela S. Cruzes. "What works for whom, where, when, and why? On the role of context in empirical software engineering." ESEM'12
5. Menzies, T. (2013). Guest editorial BEST PAPERS from the 2011 conference on predictive models in software engineering (PROMISE). *Information and Software Technology*, *55*(8), 1477-1478.
6. Krishna, R., & Menzies, T. (2018). Bellwethers: A Baseline Method For Transfer Learning. *IEEE*

Digital Object Identifier 10.1109/MS.2018.290111035 0740-7459/$26.00 2018 IEEE

Transactions on Software Engineering.

7. Shiang-Yen Tan, Taizan Chan. 2016. Defining and conceptualizing actionable insight: a conceptual framework for decision-centric analytics. Australasian Conference on Information Systems, 2015
8. R. Sawyer. 2013. AI's Impact on Analyses and Decision Making Depends on the Development of Less Complex Applications. In Principles and Applications of Business Intelligence Research. IGI Global, 83–95.
9. Hoa Khanh Dam, Truyen Tran, and Aditya Ghose. 2018. Explainable Software Analytics. arXiv preprint arXiv:1802.00603 (2018).

Figure 4: Some open issues with software analytics (and some suggestions on how to address them).

References

[1] Zhang, D., Han, S., Dang, Y., Lou, J. G., Zhang, H., & Xie, T. (2013). Software analytics in practice. IEEE software, 30(5), 30-37.

[2] Carol Passos, Ana Paula Braun, Daniela S. Cruzes, and Manoel Mendonca. Analyzing the impact of beliefs in software project practices. In ESEM'11, 2011.

[3] Prem Devanbu, Thomas Zimmermann, and Christian Bird. Belief & evidence in empirical software engineering. In Proceedings of the 38th International Conference on Software Engineering, pages 108–119. ACM, 2016

[4] M. Wilkes, Memoirs of a Computer Pioneer, MIT Press, 1985.

[5] F. Akiyama, "An Example of Software System Debugging," Information Processing, vol. 71, 1971, pp. 353–359.

[6] T. McCabe, "A Complexity Measure," IEEE Trans. Software Eng., vol. 2, no. 4, 1976, pp. 308–320. B. Boehm, Software Engineering Economics, Prentice-Hall, 1981.

[7] B. Boehm, Software Engineering Economics, Prentice-Hall, 1981.

[8] Misirli et al. "AI-Based Defect Predictors: Applications and Benefits in a Case Study", AI Magazine June 2011

[9] Minku, L. L., & Yao, X. (2013). Ensembles and locality: Insight on improving software effort estimation. Information and Software Technology, 55(8), 1512-1528.

[10] R. Krisha, A. Agrawal, A. Rahman, A. Sobran, T.Menzies, What is the Connection Between Issues, Bugs, and Enhancements? (Lessons Learned from 800+ Software Projects), ICSE-SEIP, 2018.

[11] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In Proceedings of the 34th International Conference on Software Engineering (ICSE '12).

[12] Alessandro Murgia, Parastou Tourani, Bram Adams, and Marco Ortu. 2014. Do developers feel emotions? an exploratory analysis of emotions in software artifacts. MSR'14

[13] B. Xu, D. Ye, Z. Xing, X. Xia, G. Chen and S. Li, "Predicting semantically linkable knowledge in developer online forums via convolutional neural network," ASE'16

[14] Hindle, A., Wilson, A., Rasmussen, K., Barlow, E. J., Campbell, J. C., & Romansky, S. (2014, May). Greenminer: A hardware based mining software repositories software energy consumption framework. In Proceedings of the 11th Working Conference on Mining Software Repositories (pp. 12-21). ACM.

[15] Burnett, Margaret, Simone Stumpf, Jamie Macbeth, Stephann Makri, Laura Beckwith, Irwin Kwan, Anicia Peters, and William Jernigan. "GenderMag: A method for evaluating software's gender inclusiveness." Interacting with Computers 28, no. 6 (2016): 760-787.

[16] Bird C, Nagappan N, Devanbu PT, Gall HC, Murphy B. Does distributed development affect software quality? An empirical case study of Windows Vista. In: Proceedings of the 31st international conference on software engineering, ICSE 2009, May 16–24. Vancouver, Canada

Digital Object Identifier 10.1109/MS.2018.290111035 0740-7459/$26.00 2018 IEEE

17 Thomas Fritz, Andrew Begel, Sebastian C. Müller, Serap Yigit-Elliott, and Manuela Züger. 2014. Using psycho-physiological measures to assess task difficulty in software development. In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014). ACM, New York, NY, USA, 402-413. DOI: https://doi.org/10.1145/2568225.2568266

18 Tantithamthavorn, C., McIntosh, S., Hassan, A. E., & Matsumoto, K. (2016, May). Automated parameter optimization of classification techniques for defect prediction models. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on* (pp. 321-332). IEEE.

19 Wilson G, Bryan J, Cranston K, Kitzes J, Nederbragt L, Teal TK (2017) Good enough practices in scientific computing. PLoS Comput Biol 13(6): e1005510. https://doi.org/10.1371/journal.pcbi.1005510.

20 Li, Z., Jing, X. Y., Zhu, X., Zhang, H., Xu, B., & Ying, S. (2017). On the Multiple Sources and Privacy Preservation Issues for Heterogeneous Defect Prediction. IEEE Transactions on Software Engineering.