It Will Never Work in Theory

http://neverworkintheory.org

November 11, 2021

References

[**Brown2020**] Chris Brown and Chris Parnin. Understanding the impact of GitHub suggested changes on recommendations between developers. In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 11 2020, DOI 10.1145/3368089.3409722.

Abstract: Recommendations between colleagues are effective for encouraging developers to adopt better practices. Research shows these peer interactions are useful for improving developer behaviors, or the adoption of activities to help software engineers complete programming tasks. However, in-person recommendations between developers in the workplace are declining. One form of online recommendations between developers are pull requests, which allow users to propose code changes and provide feedback on contributions. GitHub, a popular code hosting platform, recently introduced the suggested changes feature, which allows users to recommend improvements for pull requests. To better understand this feature and its impact on recommendations between developers, we report an empirical study of this system, measuring usage, effectiveness, and perception. Our results show that suggested changes support code review activities and significantly impact the timing and communication between developers on pull requests. This work provides insight into the suggested changes feature and implications for improving future systems for automated developer recommendations, such as providing situated, concise, and actionable feedback.

[**Catolino2019**] Gemma Catolino, Fabio Palomba, Damian A. Tamburri, Alexander Serebrenik, and Filomena Ferrucci. Gender diversity and women in software teams: How do they affect community smells? In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2019, DOI 10.1109/icse-seis.2019.00010.

Abstract: As social as software engineers are, there is a known and established gender imbalance in our community structures, regardless of their open-or closed-source nature. To shed light on the actual benefits of achieving such balance, this empirical study looks into the relations between such

balance and the occurrence of community smells, that is, sub-optimal circumstances and patterns across the software organizational structure. Examples of community smells are Organizational Silo effects (overly disconnected subgroups) or Lone Wolves (defiant community members). Results indicate that the presence of women generally reduces the amount of community smells. We conclude that women are instrumental to reducing community smells in software development teams.

[**Chattopadhyay2020**] Souti Chattopadhyay, Nicholas Nelson, Audrey Au, Natalia Morales, Christopher Sanchez, Rahul Pandita, and Anita Sarma. A tale from the trenches: cognitive biases and software development. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 6 2020, DOI 10.1145/3377811.3380330.

Abstract: Cognitive biases are hard-wired behaviors that influence developer actions and can set them on an incorrect course of action, necessitating backtracking. While researchers have found that cognitive biases occur in development tasks in controlled lab studies, we still don't know how these biases affect developers' everyday behavior. Without such an understanding, development tools and practices remain inadequate. To close this gap, we conducted a 2-part field study to examine the extent to which cognitive biases occur, the consequences of these biases on developer behavior, and the practices and tools that developers use to deal with these biases. About 70% of observed actions that were reversed were associated with at least one cognitive bias. Further, even though developers recognized that biases frequently occur, they routinely are forced to deal with such issues with ad hoc processes and sub-optimal tool support. As one participant (IP12) lamented: There is no salvation!

[**Danilova2021**] Anastasia Danilova, Alena Naiakshina, Stefan Horstmann, and Matthew Smith. Do you really code? designing and evaluating screening questions for online surveys with programmers. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2021, DOI 10.1109/icse43902.2021.00057.

Abstract: Recruiting professional programmers in sufficient numbers for research studies can be challenging because they often cannot spare the time, or due to their geographical distribution and potentially the cost involved. Online platforms such as Clickworker or Qualtrics do provide options to recruit participants with programming skill; however, misunderstandings and fraud can be an issue. This can result in participants without programming skill taking part in studies and surveys. If these participants are not detected, they can cause detrimental noise in the survey data. In this paper, we develop screener questions that are easy and quick to answer for people with programming skill but difficult to answer correctly for those without. In order to evaluate our questionnaire for efficacy and efficiency, we recruited several batches of participants with and without programming skill and tested the questions. In our batch 42% of Clickworkers stating that they have programming skill did not meet our criteria and we would recommend filtering these

from studies. We also evaluated the questions in an adversarial setting. We conclude with a set of recommended questions which researchers can use to recruit participants with programming skill from online platforms.

[**Devanbu2016**] Prem Devanbu, Thomas Zimmermann, and Christian Bird. Belief & evidence in empirical software engineering. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 5 2016, DOI 10.1145/2884781.2884812.

Abstract: Empirical software engineering has produced a steady stream of evidence-based results concerning the factors that affect important outcomes such as cost, quality, and interval. However, programmers often also have strongly-held a priori opinions about these issues. These opinions are important, since developers are highly trained professionals whose beliefs would doubtless affect their practice. As in evidence-based medicine, disseminating empirical findings to developers is a key step in ensuring that the findings impact practice. In this paper, we describe a case study, on the prior beliefs of developers at Microsoft, and the relationship of these beliefs to actual empirical data on the projects in which these developers work. Our findings are 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 and c) beliefs can vary with each project, but do not necessarily correspond with actual evidence in that project. Our findings suggest that more effort should be taken to disseminate empirical findings to developers and that more in-depth study the interplay of belief and evidence in software practice is needed.

[**Dias2021**] Edson Dias, Paulo Meirelles, Fernando Castor, Igor Steinmacher, Igor Wiese, and Gustavo Pinto. What makes a great maintainer of open source projects? In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2021, DOI 10.1109/icse43902.2021.00093.

Abstract: Although Open Source Software (OSS) maintainers devote a significant proportion of their work to coding tasks, great maintainers must excel in many other activities beyond coding. Maintainers should care about fostering a community, helping new members to find their place, while also saying "no" to patches that although are well-coded and well-tested, do not contribute to the goal of the project. To perform all these activities masterfully, maintainers should exercise attributes that software engineers (working on closed source projects) do not always need to master. This paper aims to uncover, relate, and prioritize the unique attributes that great OSS maintainers might have. To achieve this goal, we conducted 33 semi-structured interviews with well-experienced maintainers that are the gatekeepers of notable projects such as the Linux Kernel, the Debian operating system, and the GitLab coding platform. After we analyzed the interviews and curated a list of attributes, we created a conceptual framework to explain how these attributes are connected. We then conducted a rating survey with 90 OSS contributors. We noted that "technical excellence" and "communication" are the most recurring attributes. When grouped, these attributes fit into four

broad categories: management, social, technical, and personality. While we noted that "sustain a long term vision of the project" and being "extremely careful" seem to form the basis of our framework, we noted through our survey that the communication attribute was perceived as the most essential one.

[**Ding2021**] Zhen Yu Ding and Claire Le Goues. An empirical study of OSS-fuzz bugs. In *Proc. International Conference on Mining Software Repositories (MSR)*. IEEE, 5 2021, DOI 10.1109/msr52588.2021.00026.
Abstract: Continuous fuzzing is an increasingly popular technique for automated quality and security assurance. Google maintains OSS-Fuzz: a continuous fuzzing service for open source software. We conduct the first empirical study of OSS-Fuzz, analyzing 23,907 bugs found in 316 projects. We examine the characteristics of fuzzer-found faults, the lifecycles of such faults, and the evolution of fuzzing campaigns over time. We find that OSS-Fuzz is often effective at quickly finding bugs, and developers are often quick to patch them. However, flaky bugs, timeouts, and out of memory errors are problematic, people rarely file CVEs for security vulnerabilities, and fuzzing campaigns often exhibit punctuated equilibria, where developers might be surprised by large spikes in bugs found. Our findings have implications on future fuzzing research and practice.

[**Ferreira2021**] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. Containing malicious package updates in npm with a lightweight permission system. In *Proc. International Conference on Software Engineering (ICSE)*, 2021, DOI 10.1109/ICSE43902.2021.00121.
Abstract: The large amount of third-party packages available in fast-moving software ecosystems, such as Node.js/npm, enables attackers to compromise applications by pushing malicious updates to their package dependencies. Studying the npm repository, we observed that many packages in the npm repository that are used in Node.js applications perform only simple computations and do not need access to filesystem or network APIs. This offers the opportunity to enforce least-privilege design per package, protecting applications and package dependencies from malicious updates. We propose a lightweight permission system that protects Node.js applications by enforcing package permissions at runtime. We discuss the design space of solutions and show that our system makes a large number of packages much harder to be exploited, almost for free.

[**Flint2021**] Samuel W. Flint, Jigyasa Chauhan, and Robert Dyer. Escaping the time pit: Pitfalls and guidelines for using time-based git data. In *Proc. International Conference on Mining Software Repositories (MSR)*. IEEE, 5 2021, DOI 10.1109/msr52588.2021.00022.
Abstract: Many software engineering research papers rely on time-based data (e.g., commit timestamps, issue report creation/update/close dates, release dates). Like most real-world data however, time-based data is often dirty. To date, there are no studies that quantify how frequently such data is

used by the software engineering research community, or investigate sources of and quantify how often such data is dirty. Depending on the research task and method used, including such dirty data could affect the research results. This paper presents the first survey of papers that utilize time-based data, published in the Mining Software Repositories (MSR) conference series. Out of the 690 technical track and data papers published in MSR 2004–2020, we saw at least 35% of papers utilized time-based data. We then used the Boa and Software Heritage infrastructures to help identify and quantify several sources of dirty commit timestamp data. Finally we provide guidelines/best practices for researchers utilizing time-based data from Git repositories.

[**Fritz2010**] Thomas Fritz, Jingwen Ou, Gail C. Murphy, and Emerson Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *Proc. International Conference on Software Engineering (ICSE)*. ACM Press, 2010, DOI 10.1145/1806799.1806856.
Abstract: The size and high rate of change of source code comprising a software system make it difficult for software developers to keep up with who on the team knows about particular parts of the code. Existing approaches to this problem are based solely on authorship of code. In this paper, we present data from two professional software development teams to show that both authorship and interaction information about how a developer interacts with the code are important in characterizing a developer's knowledge of code. We introduce the degree-of-knowledge model that computes automatically a real value for each source code element based on both authorship and interaction information. We show that the degree-of-knowledge model can provide better results than an existing expertise finding approach and also report on case studies of the use of the model to support knowledge transfer and to identify changes of interest.

[**Fritzsch2021**] Jonas Fritzsch, Marvin Wyrich, Justus Bogner, and Stefan Wagner. Résumé-driven development: A definition and empirical characterization. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2021, DOI 10.1109/icse-seis52602.2021.00011.
Abstract: Technologies play an important role in the hiring process for software professionals. Within this process, several studies revealed misconceptions and bad practices which lead to suboptimal recruitment experiences. In the same context, grey literature anecdotally coined the term Résumé-Driven Development (RDD), a phenomenon describing the overemphasis of trending technologies in both job offerings and resumes as an interaction between employers and applicants. While RDD has been sporadically mentioned in books and online discussions, there are so far no scientific studies on the topic, despite its potential negative consequences. We therefore empirically investigated this phenomenon by surveying 591 software professionals in both hiring (130) and technical (558) roles and identified RDD facets in substantial parts of our sample: 60% of our hiring professionals agreed that trends influence their job offerings, while 82% of our software professionals believed that using trending technologies in their daily work makes them

more attractive for prospective employers. Grounded in the survey results, we conceptualize a theory to frame and explain Résumé-Driven Development. Finally, we discuss influencing factors and consequences and propose a definition of the term. Our contribution provides a foundation for future research and raises awareness for a potentially systemic trend that may broadly affect the software industry.

[**Gerosa2021**] Marco Gerosa, Igor Wiese, Bianca Trinkenreich, Georg Link, Gregorio Robles, Christoph Treude, Igor Steinmacher, and Anita Sarma. The shifting sands of motivation: Revisiting what drives contributors in open source. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2021, DOI 10.1109/icse43902.2021.00098.
Abstract: Open Source Software (OSS) has changed drastically over the last decade, with OSS projects now producing a large ecosystem of popular products, involving industry participation, and providing professional career opportunities. But our field's understanding of what motivates people to contribute to OSS is still fundamentally grounded in studies from the early 2000s. With the changed landscape of OSS, it is very likely that motivations to join OSS have also evolved. Through a survey of 242 OSS contributors, we investigate shifts in motivation from three perspectives: (1) the impact of the new OSS landscape, (2) the impact of individuals' personal growth as they become part of OSS communities, and (3) the impact of differences in individuals' demographics. Our results show that some motivations related to social aspects and reputation increased in frequency and that some intrinsic and internalized motivations, such as learning and intellectual stimulation, are still highly relevant. We also found that contributing to OSS often transforms extrinsic motivations to intrinsic, and that while experienced contributors often shift toward altruism, novices often shift toward career, fun, kinship, and learning. OSS projects can leverage our results to revisit current strategies to attract and retain contributors, and researchers and tool builders can better support the design of new studies and tools to engage and support OSS development.

[**Golubev2021**] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. One thousand and one stories: a large-scale survey of software refactoring. In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 8 2021, DOI 10.1145/3468264.3473924.
Abstract: Despite the availability of refactoring as a feature in popular IDEs, recent studies revealed that developers are reluctant to use them, and still prefer the manual refactoring of their code. At JetBrains, our goal is to fully support refactoring features in IntelliJ-based IDEs and improve their adoption in practice. Therefore, we start by raising the following main questions. How exactly do people refactor code? What refactorings are the most popular? Why do some developers tend not to use convenient IDE refactoring tools? In this paper, we investigate the raised questions through the

design and implementation of a survey targeting 1,183 users of IntelliJ-based IDEs. Our quantitative and qualitative analysis of the survey results shows that almost two-thirds of developers spend more than one hour in a single session refactoring their code; that refactoring types vary greatly in popularity; and that a lot of developers would like to know more about IDE refactoring features but lack the means to do so. These results serve us internally to support the next generation of refactoring features, as well as can help our research community to establish new directions in the refactoring usability research.

[**Hora2021a**] Andre Hora. Googling for software development: What developers search for and what they find. In *Proc. International Conference on Mining Software Repositories (MSR)*. IEEE, 5 2021, DOI 10.1109/msr52588.2021.00044.

Abstract: Developers often search for software resources on the web. In practice, instead of going directly to websites (e.g., Stack Overflow), they rely on search engines (e.g., Google). Despite this being a common activity, we are not yet aware of what developers search from the perspective of popular software development websites and what search results are returned. With this knowledge, we can understand real-world queries, developers' needs, and the query impact on the search results. In this paper, we provide an empirical study to understand what developers search on the web and what they find. We assess 1.3M queries to popular programming websites and we perform thousands of queries on Google to explore search results. We find that (i) developers' queries typically start with keywords (e.g., Python, Android, etc.), are short (3 words), tend to omit functional words, and are similar among each other; (ii) minor changes to queries do not largely affect the Google search results, however, some cosmetic changes may have a non-negligible impact; and (iii) search results are dominated by Stack Overflow, but YouTube is also a relevant source nowadays. We conclude by presenting detailed implications for researchers and developers.

[**Huang2020**] Yu Huang, Kevin Leach, Zohreh Sharafi, Nicholas McKay, Tyler Santander, and Westley Weimer. Biases and differences in code review using medical imaging and eye-tracking: genders, humans, and machines. In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 11 2020, DOI 10.1145/3368089.3409681.

Abstract: Code review is a critical step in modern software quality assurance, yet it is vulnerable to human biases. Previous studies have clarified the extent of the problem, particularly regarding biases against the authors of code,but no consensus understanding has emerged. Advances in medical imaging are increasingly applied to software engineering, supporting grounded neurobiological explorations of computing activities, including the review, reading, and writing of source code. In this paper, we present the results of a controlled experiment using both medical imaging and also eye tracking to investigate the neurological correlates of biases and differences

between genders of humans and machines (e.g., automated program repair tools) in code review. We find that men and women conduct code reviews differently, in ways that are measurable and supported by behavioral, eye-tracking and medical imaging data. We also find biases in how humans review code as a function of its apparent author, when controlling for code quality. In addition to advancing our fundamental understanding of how cognitive biases relate to the code review process, the results may inform subsequent training and tool design to reduce bias.

[**Huijgens2020**] Hennie Huijgens, Ayushi Rastogi, Ernst Mulders, Georgios Gousios, and Arie van Deursen. Questions for data scientists in software engineering: a replication. In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 11 2020, DOI 10.1145/3368089.3409717.
Abstract: In 2014, a Microsoft study investigated the sort of questions that data science applied to software engineering should answer. This resulted in 145 questions that developers considered relevant for data scientists to answer, thus providing a research agenda to the community. Fast forward to five years, no further studies investigated whether the questions from the software engineers at Microsoft hold for other software companies, including software-intensive companies with different primary focus (to which we refer as software-defined enterprises). Furthermore, it is not evident that the problems identified five years ago are still applicable, given the technological advances in software engineering. This paper presents a study at ING, a software-defined enterprise in banking in which over 15,000 IT staff provides in-house software solutions. This paper presents a comprehensive guide of questions for data scientists selected from the previous study at Microsoft along with our current work at ING. We replicated the original Microsoft study at ING, looking for questions that impact both software companies and software-defined enterprises and continue to impact software engineering. We also add new questions that emerged from differences in the context of the two companies and the five years gap in between. Our results show that software engineering questions for data scientists in the software-defined enterprise are largely similar to the software company, albeit with exceptions. We hope that the software engineering research community builds on the new list of questions to create a useful body of knowledge.

[**Imam2021**] Ahmed Imam and Tapajit Dey. Tracking hackathon code creation and reuse. In *Proc. International Conference on Mining Software Repositories (MSR)*. IEEE, 5 2021, DOI 10.1109/msr52588.2021.00085.
Abstract: Background: Hackathons have become popular events for teams to collaborate on projects and develop software prototypes. Most existing research focuses on activities during an event with limited attention to the evolution of the code brought to or created during a hackathon. Aim: We aim to understand the evolution of hackathon-related code, specifically, how much hackathon teams rely on pre-existing code or how much new code they develop during a hackathon. Moreover, we aim to understand if and where that

code gets reused. Method: We collected information about 22,183 hackathon projects from Devpost—a hackathon database—and obtained related code (blobs), authors, and project characteristics from the World of Code. We investigated if code blobs in hackathon projects were created before, during, or after an event by identifying the original blob creation date and author, and also checked if the original author was a hackathon project member. We tracked code reuse by first identifying all commits containing blobs created during an event before determining all projects that contain those commits. Result: While only approximately 9.14% of the code blobs are created during hackathons, this amount is still significant considering time and member constraints of such events. Approximately a third of these code blobs get reused in other projects. Conclusion: Our study demonstrates to what extent pre-existing code is used and new code is created during a hackathon and how much of it is reused elsewhere afterwards. Our findings help to better understand code reuse as a phenomenon and the role of hackathons in this context and can serve as a starting point for further studies in this area.

[**Jin2021**] Xianhao Jin and Francisco Servant. What helped, and what did not? an evaluation of the strategies to improve continuous integration. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2021, DOI `10.1109/icse43902.2021.00031`.
Abstract: Continuous integration (CI) is a widely used practice in modern software engineering. Unfortunately, it is also an expensive practice - Google and Mozilla estimate their CI systems in millions of dollars. There are a number of techniques and tools designed to or having the potential to save the cost of CI or expand its benefit - reducing time to feedback. However, their benefits in some dimensions may also result in drawbacks in others. They may also be beneficial in other scenarios where they are not designed to help. In this paper, we perform the first exhaustive comparison of techniques to improve CI, evaluating 14 variants of 10 techniques using selection and prioritization strategies on build and test granularity. We evaluate their strengths and weaknesses with 10 different cost and time-tofeedback saving metrics on 100 real-world projects. We analyze the results of all techniques to understand the design decisions that helped different dimensions of benefit. We also synthesized those results to lay out a series of recommendations for the development of future research techniques to advance this area.

[**Johnson2019**] John Johnson, Sergio Lubo, Nishitha Yedla, Jairo Aponte, and Bonita Sharif. An empirical study assessing source code readability in comprehension. In *Proc. International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 9 2019, DOI `10.1109/icsme.2019.00085`.
Abstract: Software developers spend a significant amount of time reading source code. If code is not written with readability in mind, it impacts the time required to maintain it. In order to alleviate the time taken to read and understand code, it is important to consider how readable the code is. The general consensus is that source code should be written to minimize the time it takes for others to read and understand it. In this paper, we conduct

a controlled experiment to assess two code readability rules: nesting and looping. We test 32 Java methods in four categories: ones that follow/do not follow the readability rule and that are correct/incorrect. The study was conducted online with 275 participants. The results indicate that minimizing nesting decreases the time a developer spends reading and understanding source code, increases confidence about the developer's understanding of the code, and also suggests that it improves their ability to find bugs. The results also show that avoiding the do-while statement had no significant impact on level of understanding, time spent reading and understanding, confidence in understanding, or ease of finding bugs. It was also found that the better knowledge of English a participant had, the more their readability and comprehension confidence ratings were affected by the minimize nesting rule. We discuss the implications of these findings for code readability and comprehension.

[**Kamienski2021**] Arthur V. Kamienski, Luisa Palechor, Cor-Paul Bezemer, and Abram Hindle. PySStuBs: Characterizing single-statement bugs in popular open-source python projects. In *Proc. International Conference on Mining Software Repositories (MSR)*. IEEE, 5 2021, DOI 10.1109/msr52588.2021.00066.

Abstract: Single-statement bugs (SStuBs) can have a severe impact on developer productivity. Despite usually being simple and not offering much of a challenge to fix, these bugs may still disturb a developer's workflow and waste precious development time. However, few studies have paid attention to these simple bugs, focusing instead on bugs of any size and complexity. In this study, we explore the occurrence of SStuBs in some of the most popular open-source Python projects on GitHub, while also characterizing their patterns and distribution. We further compare these bugs to SStuBs found in a previous study on Java Maven projects. We find that these Python projects have different SStuB patterns than the ones in Java Maven projects and identify 7 new SStuB patterns. Our results may help uncover the importance of understanding these bugs for the Python programming language, and how developers can handle them more effectively.

[**Krueger2020**] Ryan Krueger, Yu Huang, Xinyu Liu, Tyler Santander, Westley Weimer, and Kevin Leach. Neurological divide: an fMRI study of prose and code writing. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 6 2020, DOI 10.1145/3377811.3380348.

Abstract: Software engineering involves writing new code or editing existing code. Recent efforts have investigated the neural processes associated with reading and comprehending code—however, we lack a thorough understanding of the human cognitive processes underlying code writing. While prose reading and writing have been studied thoroughly, that same scrutiny has not been applied to code writing. In this paper, we leverage functional brain imaging to investigate neural representations of code writing in comparison to prose writing. We present the first human study in which participants wrote code and prose while undergoing a functional magnetic res-

onance imaging (fMRI) brain scan, making use of a full-sized fMRI-safe QWERTY keyboard. We find that code writing and prose writing are significantly dissimilar neural tasks. While prose writing entails significant left hemisphere activity associated with language, code writing involves more activations of the right hemisphere, including regions associated with attention control, working memory, planning and spatial cognition. These findings are unlike existing work in which code and prose comprehension were studied. By contrast, we present the first evidence suggesting that code and prose writing are quite dissimilar at the neural level.

[**Lamba2020**] Hemank Lamba, Asher Trockman, Daniel Armanios, Christian Kästner, Heather Miller, and Bogdan Vasilescu. Heard it through the gitvine: an empirical study of tool diffusion across the npm ecosystem. In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 11 2020, DOI 10.1145/3368089.3409705.
Abstract: Automation tools like continuous integration services, code coverage reporters, style checkers, dependency managers, etc. are all known to provide significant improvements in developer productivity and software quality. Some of these tools are widespread, others are not. How do these automation "best practices" spread? And how might we facilitate the diffusion process for those that have seen slower adoption? In this paper, we rely on a recent innovation in transparency on code hosting platforms like GitHub—the use of repository badges—to track how automation tools spread in open-source ecosystems through different social and technical mechanisms over time. Using a large longitudinal data set, multivariate network science techniques, and survival analysis, we study which socio-technical factors can best explain the observed diffusion process of a number of popular automation tools. Our results show that factors such as social exposure, competition, and observability affect the adoption of tools significantly, and they provide a roadmap for software engineers and researchers seeking to propagate best practices and tools.

[**Lampel2021**] Johannes Lampel, Sascha Just, Sven Apel, and Andreas Zeller. When life gives you oranges: detecting and diagnosing intermittent job failures at mozilla. In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 8 2021, DOI 10.1145/3468264.3473931.
Abstract: Continuous delivery of cloud systems requires constant running of jobs (build processes, tests, etc.). One issue that plagues this continuous integration (CI) process are intermittent failures - non-deterministic, false alarms that do not result from a bug in the software or job specification, but rather from issues in the underlying infrastructure. At Mozilla, such intermittent failures are called oranges as a reference to the color of the build status indicator. As such intermittent failures disrupt CI and lead to failures, they erode the developers' trust in the jobs. We present a novel approach that automatically classifies failing jobs to determine whether job execution

failures arise from an actual software bug or were caused by flakiness in the job (e.g., test) or the underlying infrastructure. For this purpose, we train classification models using job telemetry data to diagnose failure patterns involving features such as runtime, cpu load, operating system version, or specific platform with high precision. In an evaluation on a set of Mozilla CI jobs, our approach achieves precision scores of 73%, on average, across all data sets with some test suites achieving precision scores good enough for fully automated classification (i.e., precision scores of up to 100%), and recall scores of 82% on average (up to 94%).

[**Latendresse2021**] Jasmine Latendresse, Rabe Abdalkareem, Diego Elias Costa, and Emad Shihab. How effective is continuous integration in indicating single-statement bugs? In *Proc. International Conference on Mining Software Repositories (MSR)*. IEEE, 5 2021, DOI 10.1109/msr52588.2021.00062.

Abstract: Continuous Integration (CI) is the process of automatically compiling, building, and testing code changes in the hope of catching bugs as they are introduced into the code base. With bug fixing being a core and increasingly costly task in software development, the community has adopted CI to mitigate this issue and improve the quality of their software products. Bug fixing is a core task in software development and becomes increasingly costly over time. However, little is known about how effective CI is at detecting simple, single-statement bugs.In this paper, we analyze the effectiveness of CI in 14 popular open source Java-based projects to warn about 318 single-statement bugs (SStuBs). We analyze the build status at the commits that introduce SStuBs and before the SStuBs were fixed. We then investigate how often CI indicates the presence of these bugs, through test failure. Our results show that only 2% of the commits that introduced SStuBs have builds with failed tests and 7.5% of builds before the fix reported test failures. Upon close manual inspection, we found that none of the failed builds actually captured SStuBs, indicating that CI is not the right medium to capture the SStuBs we studied. Our results suggest that developers should not rely on CI to catch SStuBs or increase their CI pipeline coverage to detect single-statement bugs.

[**Louis2020**] Annie Louis, Santanu Kumar Dash, Earl T. Barr, Michael D. Ernst, and Charles Sutton. Where should i comment my code?: a dataset and model for predicting locations that need comments. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 6 2020, DOI 10.1145/3377816.3381736.

Abstract: Programmers should write code comments, but not on every line of code. We have created a machine learning model that suggests locations where a programmer should write a code comment. We trained it on existing commented code to learn locations that are chosen by developers. Once trained, the model can predict locations in new code. Our models achieved precision of 74% and recall of 13% in identifying comment-worthy locations.

This first success opens the door to future work, both in the new where-to-comment problem and in guiding comment generation. Our code and data is available at http://groups.inf.ed.ac.uk/cup/comment-locator/.

[**Melo2019**] Hugo Melo, Roberta Coelho, and Christoph Treude. Unveiling exception handling guidelines adopted by java developers. In *Proc. International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2 2019, DOI 10.1109/saner.2019.8668001.

Abstract: Despite being an old language feature, Java exception handling code is one of the least understood parts of many systems. Several studies have analyzed the characteristics of exception handling code, trying to identify common practices or even link such practices to software bugs. Few works, however, have investigated exception handling issues from the point of view of developers. None of the works have focused on discovering exception handling guidelines adopted by current systems—which are likely to be a driver of common practices. In this work, we conducted a qualitative study based on semi-structured interviews and a survey whose goal was to investigate the guidelines that are (or should be) followed by developers in their projects. Initially, we conducted semi-structured interviews with seven experienced developers, which were used to inform the design of a survey targeting a broader group of Java developers (i.e., a group of active Java developers from top-starred projects on GitHub). We emailed 863 developers and received 98 valid answers. The study shows that exception handling guidelines usually exist (70%) and are usually implicit and undocumented (54%). Our study identifies 48 exception handling guidelines related to seven different categories. We also investigated how such guidelines are disseminated to the project team and how compliance between code and guidelines is verified; we could observe that according to more than half of respondents the guidelines are both disseminated and verified through code inspection or code review. Our findings provide software development teams with a means to improve exception handling guidelines based on insights from the state of practice of 87 software projects.

[**Patra2021**] Jibesh Patra and Michael Pradel. Semantic bug seeding: a learning-based approach for creating realistic bugs. In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 8 2021, DOI 10.1145/3468264.3468623.

Abstract: When working on techniques to address the wide-spread problem of software bugs, one often faces the need for a large number of realistic bugs in real-world programs. Such bugs can either help evaluate an approach, e.g., in form of a bug benchmark or a suite of program mutations, or even help build the technique, e.g., in learning-based bug detection. Because gathering a large number of real bugs is difficult, a common approach is to rely on automatically seeded bugs. Prior work seeds bugs based on syntactic transformation patterns, which often results in unrealistic bugs and typically cannot introduce new, application-specific code tokens. This paper presents

SemSeed, a technique for automatically seeding bugs in a semantics-aware way. The key idea is to imitate how a given real-world bug would look like in other programs by semantically adapting the bug pattern to the local context. To reason about the semantics of pieces of code, our approach builds on learned token embeddings that encode the semantic similarities of identifiers and literals. Our evaluation with real-world JavaScript software shows that the approach effectively reproduces real bugs and clearly outperforms a semantics-unaware approach. The seeded bugs are useful as training data for learning-based bug detection, where they significantly improve the bug detection ability. Moreover, we show that SemSeed-created bugs complement existing mutation testing operators, and that our approach is efficient enough to seed hundreds of thousands of bugs within an hour.

[**Peitek2021**] Norman Peitek, Sven Apel, Chris Parnin, Andre Brechmann, and Janet Siegmund. Program comprehension and code complexity metrics: An fMRI study. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2021, DOI 10.1109/icse43902.2021.00056.
Abstract: Background: Researchers and practitioners have been using code complexity metrics for decades to predict how developers comprehend a program. While it is plausible and tempting to use code metrics for this purpose, their validity is debated, since they rely on simple code properties and rarely consider particularities of human cognition. Aims: We investigate whether and how code complexity metrics reflect difficulty of program comprehension. Method: We have conducted a functional magnetic resonance imaging (fMRI) study with 19 participants observing program comprehension of short code snippets at varying complexity levels. We dissected four classes of code complexity metrics and their relationship to neuronal, behavioral, and subjective correlates of program comprehension, overall analyzing more than 41 metrics. Results: While our data corroborate that complexity metrics can-to a limited degree-explain programmers' cognition in program comprehension, fMRI allowed us to gain insights into why some code properties are difficult to process. In particular, a code's textual size drives programmers' attention, and vocabulary size burdens programmers' working memory. Conclusion: Our results provide neuro-scientific evidence supporting warnings of prior research questioning the validity of code complexity metrics and pin down factors relevant to program comprehension. Future Work: We outline several follow-up experiments investigating fine-grained effects of code complexity and describe possible refinements to code complexity metrics.

[**Qiu2019**] Huilian Sophie Qiu, Alexander Nolte, Anita Brown, Alexander Serebrenik, and Bogdan Vasilescu. Going farther together: The impact of social capital on sustained participation in open source. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2019, DOI 10.1109/icse.2019.00078.
Abstract: Sustained participation by contributors in opensource software is critical to the survival of open-source projects and can provide career advancement benefits to individual contributors. However, not all contributors

reap the benefits of open-source participation fully, with prior work showing that women are particularly underrepresented and at higher risk of disengagement. While many barriers to participation in open-source have been documented in the literature, relatively little is known about how the social networks that open-source contributors form impact their chances of long-term engagement. In this paper we report on a mixed-methods empirical study of the role of social capital (i.e., the resources people can gain from their social connections) for sustained participation by women and men in open-source GitHub projects. After combining survival analysis on a large, longitudinal data set with insights derived from a user survey, we confirm that while social capital is beneficial for prolonged engagement for both genders, women are at disadvantage in teams lacking diversity in expertise.

[**Rahman2020b**] Mohammad Masudur Rahman, Foutse Khomh, and Marco Castelluccio. Why are some bugs non-reproducible? an empirical investigation using data fusion. In *Proc. International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 9 2020, DOI 10.1109/icsme46990.2020.00063.

Abstract: Software developers attempt to reproduce software bugs to understand their erroneous behaviours and to fix them. Unfortunately, they often fail to reproduce (or fix) them, which leads to faulty, unreliable software systems. However, to date, only a little research has been done to better understand what makes the software bugs non-reproducible. In this paper, we conduct a multimodal study to better understand the non-reproducibility of software bugs. First, we perform an empirical study using 576 non-reproducible bug reports from two popular software systems (Firefox, Eclipse) and identify 11 key factors that might lead a reported bug to non-reproducibility. Second, we conduct a user study involving 13 professional developers where we investigate how the developers cope with non-reproducible bugs. We found that they either close these bugs or solicit for further information, which involves long deliberations and counter-productive manual searches. Third, we offer several actionable insights on how to avoid non-reproducibility (e.g., false-positive bug report detector) and improve reproducibility of the reported bugs (e.g., sandbox for bug reproduction) by combining our analyses from multiple studies (e.g., empirical study, developer study).

[**Reyes2018**] Rolando P. Reyes, Oscar Dieste, Efraín R. Fonseca, and Natalia Juristo. Statistical errors in software engineering experiments. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 5 2018, DOI 10.1145/3180155.3180161.

Abstract: Background: Statistical concepts and techniques are often applied incorrectly, even in mature disciplines such as medicine or psychology. Surprisingly, there are very few works that study statistical problems in software engineering (SE). Aim: Assess the existence of statistical errors in SE experiments. Method: Compile the most common statistical errors in experimental disciplines. Survey experiments published in ICSE to assess

whether errors occur in high quality SE publications. Results: The same errors as identified in others disciplines were found in ICSE experiments, where 30 of the reviewed papers included several error types such as: a) missing statistical hypotheses, b) missing sample size calculation, c) failure to assess statistical test assumptions, and d) uncorrected multiple testing. This rather large error rate is greater for research papers where experiments are confined to the validation section. The origin of the errors can be traced back to: a) researchers not having sufficient statistical training, and b) a profusion of exploratory research. Conclusions: This paper provides preliminary evidence that SE research suffers from the same statistical problems as other experimental disciplines. However, the SE community appears to be unaware of any shortcomings in its experiments, whereas other disciplines work hard to avoid these threats. Further research is necessary to find the underlying causes and set up corrective measures, but there are some potentially effective actions and are a priori easy to implement: a) improve the statistical training of SE researchers, and b) enforce quality assessment and reporting guidelines in SE publications.

[**Rodeghero2021**] Paige Rodeghero, Thomas Zimmermann, Brian Houck, and Denae Ford. Please turn your cameras on: Remote onboarding of software developers during a pandemic. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2021, DOI 10.1109/icse-seip52600.2021.00013.

Abstract: The COVID-19 pandemic has impacted the way that software development teams onboard new hires. Previously, most software developers worked in physical offices and new hires onboarded to their teams in the physical office, following a standard onboarding process. However, when companies transitioned employees to work from home due to the pandemic, there was little to no time to develop new onboarding procedures. In this paper, we present a survey of 267 new hires at Microsoft that onboarded to software development teams during the pandemic. We explored their remote onboarding process, including the challenges that the new hires encountered and their social connectedness with their teams. We found that most developers onboarded remotely and never had an opportunity to meet their teammates in person. This leads to one of the biggest challenges faced by these new hires, building a strong social connection with their team. We use these results to provide recommendations for onboarding remote hires.

[**Romano2021**] Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. An empirical analysis of UI-based flaky tests. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2021, DOI 10.1109/icse43902.2021.00141.

Abstract: Flaky tests have gained attention from the research community in recent years and with good reason. These tests lead to wasted time and resources, and they reduce the reliability of the test suites and build systems they affect. However, most of the existing work on flaky tests focus exclusively on traditional unit tests. This work ignores UI tests that have larger

input spaces and more diverse running conditions than traditional unit tests. In addition, UI tests tend to be more complex and resource-heavy, making them unsuited for detection techniques involving rerunning test suites multiple times. In this paper, we perform a study on flaky UI tests. We analyze 235 flaky UI test samples found in 62 projects from both web and Android environments. We identify the common underlying root causes of flakiness in the UI tests, the strategies used to manifest the flaky behavior, and the fixing strategies used to remedy flaky UI tests. The findings made in this work can provide a foundation for the development of detection and prevention techniques for flakiness arising in UI tests.

[**Sarker2019**] Farhana Sarker, Bogdan Vasilescu, Kelly Blincoe, and Vladimir Filkov. Socio-technical work-rate increase associates with changes in work patterns in online projects. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2019, DOI 10.1109/icse.2019.00099.
Abstract: Software developers work on a variety of tasks ranging from the technical, e.g., writing code, to the social, e.g., participating in issue resolution discussions. The amount of work developers perform per week (their work-rate) also varies and depends on project needs and developer schedules. Prior work has shown that while moderate levels of increased technical work and multitasking lead to higher productivity, beyond a certain threshold, they can lead to lowered performance. Here, we study how increases in the short-term work-rate along both the technical and social dimensions are associated with changes in developers' work patterns, in particular communication sentiment, technical productivity, and social productivity. We surveyed active and prolific developers on GitHub to understand the causes and impacts of increased work-rates. Guided by the responses, we developed regression models to study how communication and committing patterns change with increased work-rates and fit those models to large-scale data gathered from traces left by thousands of GitHub developers. From our survey and models, we find that most developers do experience work-rate-increase-related changes in behavior. Most notably, our models show that there is a sizable effect when developers comment much more than their average: the negative sentiment in their comments increases, suggesting an increased level of stress. Our models also show that committing patterns do not change with increased commenting, and vice versa, suggesting that technical and social activities tend not to be multitasked.

[**Shao2020**] Shudi Shao, Zhengyi Qiu, Xiao Yu, Wei Yang, Guoliang Jin, Tao Xie, and Xintao Wu. Database-access performance antipatterns in database-backed web applications. In *Proc. International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 9 2020, DOI 10.1109/icsme46990.2020.00016.
Abstract: Database-backed web applications are prone to performance bugs related to database accesses. While much work has been conducted on database-access antipatterns with some recent work focusing on performance impact, there still lacks a comprehensive view of database-access per-

formance antipatterns in database-backed web applications. To date, no existing work systematically reports known antipatterns in the literature, and no existing work has studied database-access performance bugs in major types of web applications that access databases differently.To address this issue, we first summarize all known database-access performance antipatterns found through our literature survey, and we report all of them in this paper. We further collect database-access performance bugs from web applications that access databases through language-provided SQL interfaces, which have been largely ignored by recent work, to check how extensively the known antipatterns can cover these bugs. For bugs not covered by the known antipatterns, we extract new database-access performance antipatterns based on real-world performance bugs from such web applications. Our study in total reports 24 known and 10 new database-access performance antipatterns. Our results can guide future work to develop effective tool support for different types of web applications.

[**Sharma2021**] Pankajeshwara Nand Sharma, Bastin Tony Roy Savarimuthu, and Nigel Stanger. Extracting rationale for open source software development decisions—a study of python email archives. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2021, DOI 10.1109/icse43902.2021.00095.

Abstract: A sound Decision-Making (DM) process is key to the successful governance of software projects. In many Open Source Software Development (OSSD) communities, DM processes lie buried amongst vast amounts of publicly available data. Hidden within this data lie the rationale for decisions that led to the evolution and maintenance of software products. While there have been some efforts to extract DM processes from publicly available data, the rationale behind 'how' the decisions are made have seldom been explored. Extracting the rationale for these decisions can facilitate transparency (by making them known), and also promote accountability on the part of decision-makers. This work bridges this gap by means of a large-scale study that unearths the rationale behind decisions from Python development email archives comprising about 1.5 million emails. This paper makes two main contributions. First, it makes a knowledge contribution by unearthing and presenting the rationale behind decisions made. Second, it makes a methodological contribution by presenting a heuristics-based rationale extraction system called Rationale Miner that employs multiple heuristics, and follows a data-driven, bottom-up approach to infer the rationale behind specific decisions (e.g., whether a new module is implemented based on core developer consensus or benevolent dictator's pronouncement). Our approach can be applied to extract rationale in other OSSD communities that have similar governance structures.

[**Shrestha2020**] Nischal Shrestha, Colton Botta, Titus Barik, and Chris Parnin. Here we go again: why is it difficult for developers to learn another programming language? In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 6 2020, DOI 10.1145/3377811.3380352.

Abstract: Once a programmer knows one language, they can leverage concepts and knowledge already learned, and easily pick up another programming language. But is that always the case? To understand if programmers have difficulty learning additional programming languages, we conducted an empirical study of Stack Overflow questions across 18 different programming languages. We hypothesized that previous knowledge could potentially interfere with learning a new programming language. From our inspection of 450 Stack Overflow questions, we found 276 instances of interference that occurred due to faulty assumptions originating from knowledge about a different language. To understand why these difficulties occurred, we conducted semi-structured interviews with 16 professional programmers. The interviews revealed that programmers make failed attempts to relate a new programming language with what they already know. Our findings inform design implications for technical authors, toolsmiths, and language designers, such as designing documentation and automated tools that reduce interference, anticipating uncommon language transitions during language design, and welcoming programmers not just into a language, but its entire ecosystem.

[**Spadini2019**] Davide Spadini, Fabio Palomba, Tobias Baum, Stefan Hanenberg, Magiel Bruntink, and Alberto Bacchelli. Test-driven code review: An empirical study. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2019, DOI 10.1109/icse.2019.00110.

Abstract: Test-Driven Code Review (TDR) is a code review practice in which a reviewer inspects a patch by examining the changed test code before the changed production code. Although this practice has been mentioned positively by practitioners in informal literature and interviews, there is no systematic knowledge of its effects, prevalence, problems, and advantages. In this paper, we aim at empirically understanding whether this practice has an effect on code review effectiveness and how developers' perceive TDR. We conduct (i) a controlled experiment with 93 developers that perform more than 150 reviews, and (ii) 9 semi-structured interviews and a survey with 103 respondents to gather information on how TDR is perceived. Key results from the experiment show that developers adopting TDR find the same proportion of defects in production code, but more in test code, at the expenses of fewer maintainability issues in production code. Furthermore, we found that most developers prefer to review production code as they deem it more critical and tests should follow from it. Moreover, general poor test code quality and no tool support hinder the adoption of TDR.

[**Spadini2020**] Davide Spadini, Gül Çalikli, and Alberto Bacchelli. Primers or reminders?: the effects of existing review comments on code review. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 6 2020, DOI 10.1145/3377811.3380385.

Abstract: In contemporary code review, the comments put by reviewers on a specific code change are immediately visible to the other reviewers involved. Could this visibility prime new reviewers' attention (due to the human's proneness to availability bias), thus biasing the code review outcome?

In this study, we investigate this topic by conducting a controlled experiment with 85 developers who perform a code review and a psychological experiment. With the psychological experiment, we find that ≈70% of participants are prone to availability bias. However, when it comes to the code review, our experiment results show that participants are primed only when the existing code review comment is about a type of bug that is not normally considered; when this comment is visible, participants are more likely to find another occurrence of this type of bug. Moreover, this priming effect does not influence reviewers' likelihood of detecting other types of bugs. Our findings suggest that the current code review practice is effective because existing review comments about bugs in code changes are not negative primers, rather positive reminders for bugs that would otherwise be overlooked during code review. Data and materials: https://doi.org/10.5281/zenodo.3653856

[**Tan2020a**] Xin Tan, Minghui Zhou, and Zeyu Sun. A first look at good first issues on GitHub. In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 11 2020, DOI 10.1145/3368089.3409746.
Abstract: Keeping a good influx of newcomers is critical for open source software projects' survival, while newcomers face many barriers to contributing to a project for the first time. To support newcomers onboarding, GitHub encourages projects to apply labels such as good first issue (GFI) to tag issues suitable for newcomers. However, many newcomers still fail to contribute even after many attempts, which not only reduces the enthusiasm of newcomers to contribute but makes the efforts of project members in vain. To better support the onboarding of newcomers, this paper reports a preliminary study on this mechanism from its application status, effect, problems, and best practices. By analyzing 9,368 GFIs from 816 popular GitHub projects and conducting email surveys with newcomers and project members, we obtain the following results. We find that more and more projects are applying this mechanism in the past decade, especially the popular projects. Compared to common issues, GFIs usually need more days to be solved. While some newcomers really join the projects through GFIs, almost half of GFIs are not solved by newcomers. We also discover a series of problems covering mechanism (e.g., inappropriate GFIs), project (e.g., insufficient GFIs) and newcomer (e.g., uneven skills) that makes this mechanism ineffective. We discover the practices that may address the problems, including identifying GFIs that have informative description and available support, and require limited scope and skill, etc. Newcomer onboarding is an important but challenging question in open source projects and our work enables a better understanding of GFI mechanism and its problems, as well as highlights ways in improving them.

[**Tomassi2019**] David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-Gonzalez. BugSwarm: Mining and continuously growing a dataset of reproducible failures and fixes. In *Proc. Interna-*

tional Conference on Software Engineering (ICSE). IEEE, 5 2019, DOI 10.1109/icse.2019.00048.
Abstract: Fault-detection, localization, and repair methods are vital to software quality; but it is difficult to evaluate their generality, applicability, and current effectiveness. Large, diverse, realistic datasets of durably-reproducible faults and fixes are vital to good experimental evaluation of approaches to software quality, but they are difficult and expensive to assemble and keep current. Modern continuous-integration (CI) approaches, like TRAVIS-CI, which are widely used, fully configurable, and executed within custom-built containers, promise a path toward much larger defect datasets. If we can identify and archive failing and subsequent passing runs, the containers will provide a substantial assurance of durable future reproducibility of build and test. Several obstacles, however, must be overcome to make this a practical reality. We describe BUGSWARM, a toolset that navigates these obstacles to enable the creation of a scalable, diverse, realistic, continuously growing set of durably reproducible failing and passing versions of real-world, open-source systems. The BUGSWARM toolkit has already gathered 3,091 fail-pass pairs, in Java and Python, all packaged within fully reproducible containers. Furthermore, the toolkit can be run periodically to detect fail-pass activities, thus growing the dataset continually.

[**Uesbeck2020**] P. Merlin Uesbeck, Cole S. Peterson, Bonita Sharif, and Andreas Stefik. A randomized controlled trial on the effects of embedded computer language switching. In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 11 2020, DOI 10.1145/3368089.3409701.
Abstract: Polyglot programming, the use of multiple programming languages during the development process, is common practice in modern software development. This study investigates this practice through a randomized controlled trial conducted under the context of database programming. Participants in the study were given coding tasks written in Java and one of three SQL-like embedded languages. One was plain SQL in strings, one was in Java only, and the third was a hybrid embedded language that was closer to the host language. We recorded 109 valid data points. Results showed significant differences in how developers of different experience levels code using polyglot techniques. Notably, less experienced programmers wrote correct programs faster in the hybrid condition (frequent, but less severe, switches), while more experienced developers that already knew both languages performed better in traditional SQL (less frequent but more complete switches). The results indicate that the productivity impact of polyglot programming is complex and experience level dependent.

[**Weir2021**] Charles Weir, Ingolf Becker, and Lynne Blair. A passion for security: Intervening to help software developers. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2021, DOI 10.1109/icse-seip52600.2021.00011.

Abstract: While the techniques to achieve secure, privacy-preserving software are now well understood, evidence shows that many software development teams do not use them: they lack the 'security maturity' to assess security needs and decide on appropriate tools and processes; and they lack the ability to negotiate with product management for the required resources. This paper describes a measuring approach to assess twelve aspects of this security maturity; its use to assess the impact of a lightweight package of workshops designed to increase security maturity; and a novel approach within that package to support developers in resource negotiation. Based on trials in eight organizations, involving over 80 developers, this paper demonstrates that (1) development teams can notably improve their security maturity even in the absence of security specialists; and (2) suitably guided, developers can find effective ways to promote security to product management. Empowering developers to make their own decisions and promote security in this way offers a powerful grassroots approach to improving the security of software worldwide.

[**Yasmin2020**] Jerin Yasmin, Yuan Tian, and Jinqiu Yang. A first look at the deprecation of RESTful APIs: An empirical study. In *Proc. International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 9 2020, DOI 10.1109/icsme46990.2020.00024.

Abstract: REpresentational State Transfer (REST) is considered as one standard software architectural style to build web APIs that can integrate software systems over the internet. However, while connecting systems, RESTful APIs might also break the dependent applications that rely on their services when they introduce breaking changes, e.g., an older version of the API is no longer supported. To warn developers promptly and thus prevent critical impact on downstream applications, a deprecated-removed model should be followed, and deprecation-related information such as alternative approaches should also be listed. While API deprecation analysis as a theme is not new, most existing work focuses on non-web APIs, such as the ones provided by Java and Android.To investigate RESTful API deprecation, we propose a framework called RADA (RESTful API Deprecation Analyzer). RADA is capable of automatically identifying deprecated API elements and analyzing impacted operations from an OpenAPI specification, a machine-readable profile for describing RESTful web service. We apply RADA on 2,224 OpenAPI specifications of 1,368 RESTful APIs collected from APIs.guru, the largest directory of OpenAPI specifications. Based on the data mined by RADA, we perform an empirical study to investigate how the deprecated-removed protocol is followed in RESTful APIs and characterize practices in RESTful API deprecation. The results of our study reveal several severe deprecation-related problems in existing RESTful APIs. Our implementation of RADA and detailed empirical results are publicly available for future intelligent tools that could automatically identify and migrate usage of deprecated RESTful API operations in client code.

[**Young2021**] Jean-Gabriel Young, Amanda Casari, Katie McLaughlin,

Milo Z. Trujillo, Laurent Hebert-Dufresne, and James P. Bagrow. Which contributions count? analysis of attribution in open source. In *Proc. International Conference on Mining Software Repositories (MSR)*. IEEE, 5 2021, DOI 10.1109/msr52588.2021.00036.

Abstract: Open source software projects usually acknowledge contributions with text files, websites, and other idiosyncratic methods. These data sources are hard to mine, which is why contributorship is most frequently measured through changes to repositories, such as commits, pushes, or patches. Recently, some open source projects have taken to recording contributor actions with standardized systems; this opens up a unique opportunity to understand how community-generated notions of contributorship map onto codebases as the measure of contribution. Here, we characterize contributor acknowledgment models in open source by analyzing thousands of projects that use a model called All Contributors to acknowledge diverse contributions like outreach, finance, infrastructure, and community management. We analyze the life cycle of projects through this model's lens and contrast its representation of contributorship with the picture given by other methods of acknowledgment, including GitHub's top committers indicator and contributions derived from actions taken on the platform. We find that community-generated systems of contribution acknowledgment make work like idea generation or bug finding more visible, which generates a more extensive picture of collaboration. Further, we find that models requiring explicit attribution lead to more clearly defined boundaries around what is and is not a contribution.

[**Zieris2020**] Franz Zieris and Lutz Prechelt. Explaining pair programming session dynamics from knowledge gaps. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 6 2020, DOI 10.1145/3377811.3380925.

Abstract: Background: Despite a lot of research on the effectiveness of Pair Programming (PP), the question when it is useful or less useful remains unsettled. Method: We analyze recordings of many industrial PP sessions with Grounded Theory Methodology and build on prior work that identified various phenomena related to within-session knowledge build-up and transfer. We validate our findings with practitioners. Result: We identify two fundamentally different types of required knowledge and explain how different constellations of knowledge gaps in these two respects lead to different session dynamics. Gaps in project-specific systems knowledge are more hampering than gaps in general programming knowledge and are dealt with first and foremost in a PP session. Conclusion: Partner constellations with complementary knowledge make PP a particularly effective practice. In PP sessions, differences in system understanding are more important than differences in general software development knowledge.