It Will Never Work in Theory

http://neverworkintheory.org

November 2, 2021

References

[**Balali2018**] Sogol Balali, Igor Steinmacher, Umayal Annamalai, Anita Sarma, and Marco Aurelio Gerosa. Newcomers' barriers... is that all? an analysis of mentors' and newcomers' barriers in OSS projects. *Computer Supported Cooperative Work*, 27(3-6):679–714, 4 2018, DOI 10.1007/s10606-018-9310-8.

Abstract: Newcomers' seamless onboarding is important for open collaboration communities, particularly those that leverage outsiders' contributions to remain sustainable. Nevertheless, previous work shows that OSS newcomers often face several barriers to contribute, which lead them to lose motivation and even give up on contributing. A well-known way to help newcomers overcome initial contribution barriers is mentoring. This strategy has proven effective in offline and online communities, and to some extent has been employed in OSS projects. Studying mentors' perspectives on the barriers that newcomers face play a vital role in improving onboarding processes; yet, OSS mentors face their own barriers, which hinder the effectiveness of the strategy. Since little is known about the barriers mentors face, in this paper, we investigate the barriers that affect mentors and their newcomer mentees. We interviewed mentors from OSS projects and qualitatively analyzed their answers. We found 44 barriers: 19 that affect mentors; and 34 that affect newcomers (9 affect both newcomers and mentors). Interestingly, most of the barriers we identified (66%) have a social nature. Additionally, we identified 10 strategies that mentors indicated to potentially alleviate some of the barriers. Since gender-related challenges emerged in our analysis, we conducted nine follow-up structured interviews to further explore this perspective. The contributions of this paper include: identifying the barriers mentors face; bringing the unique perspective of mentors on barriers faced by newcomers; unveiling strategies that can be used by mentors to support newcomers; and investigating gender-specific challenges in OSS mentorship. Mentors, newcomers, online communities, and educators can leverage this knowledge to foster new contributors to OSS projects.

[**Barke2019**] Helena Barke and Lutz Prechelt. Role clarity deficiencies can wreck agile teams. *PeerJ Computer Science*, 5:e241, 12 2019, DOI

10.7717/peerj-cs.241.
Abstract: Background One of the twelve agile principles is to build projects around motivated individuals and trust them to get the job done. Such agile teams must self-organize, but this involves conflict, making self-organization difficult. One area of difficulty is agreeing on everybody's role. Background What dynamics arise in a self-organizing team from the negotiation of everybody's role? Method We conceptualize observations from five agile teams (work observations, interviews) by Charmazian Grounded Theory Methodology. Results We define role as something transient and implicit, not fixed and named. The roles are characterized by the responsibilities and expectations of each team member. Every team member must understand and accept their own roles (Local role clarity) and everbody else's roles (Team-wide role clarity). Role clarity allows a team to work smoothly and effectively and to develop its members' skills fast. Lack of role clarity creates friction that not only hampers the day-to-day work, but also appears to lead to high employee turnover. Agile coaches are critical to create and maintain role clarity. Conclusions Agile teams should pay close attention to the levels of Local role clarity of each member and Team-wide role clarity overall, because role clarity deficits are highly detrimental.

[**Bi2021**] Tingting Bi, Wei Ding, Peng Liang, and Antony Tang. Architecture information communication in two OSS projects: The why, who, when, and what. *Journal of Systems and Software*, 181:111035, 11 2021, DOI 10.1016/j.jss.2021.111035.
Abstract: Architecture information is vital for Open Source Software (OSS) development, and mailing list is one of the widely used channels for developers to share and communicate architecture information. This work investigates the nature of architecture information communication (i.e., why, who, when, and what) by OSS developers via developer mailing lists. We employed a multiple case study approach to extract and analyze the architecture information communication from the developer mailing lists of two OSS projects, ArgoUML and Hibernate, during their development lifecycle of over 18 years. Our main findings are: (a) architecture negotiation and interpretation are the two main reasons (i.e., why) of architecture communication; (b) the amount of architecture information communicated in developer mailing lists decreases after the first stable release (i.e., when); (c) architecture communications centered around a few core developers (i.e., who); (d) and the most frequently communicated architecture elements (i.e., what) are Architecture Rationale and Architecture Model. There are a few similarities of architecture communication between the two OSS projects. Such similarities point to how OSS developers naturally gravitate towards the four aspects of architecture communication in OSS development.

[**Bogart2021**] Chris Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. When and how to make breaking changes. *ACM Transactions on Software Engineering and Methodology*, 30(4):1–56, 7 2021, DOI 10.1145/3447245.

Abstract: Open source software projects often rely on package management systems that help projects discover, incorporate, and maintain dependencies on other packages, maintained by other people. Such systems save a great deal of effort over ad hoc ways of advertising, packaging, and transmitting useful libraries, but coordination among project teams is still needed when one package makes a breaking change affecting other packages. Ecosystems differ in their approaches to breaking changes, and there is no general theory to explain the relationships between features, behavioral norms, ecosystem outcomes, and motivating values. We address this through two empirical studies. In an interview case study, we contrast Eclipse, NPM, and CRAN, demonstrating that these different norms for coordination of breaking changes shift the costs of using and maintaining the software among stakeholders, appropriate to each ecosystem's mission. In a second study, we combine a survey, repository mining, and document analysis to broaden and systematize these observations across 18 ecosystems. We find that all ecosystems share values such as stability and compatibility, but differ in other values. Ecosystems' practices often support their espoused values, but in surprisingly diverse ways. The data provides counterevidence against easy generalizations about why ecosystem communities do what they do.

[**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.

[**Butler2019**] Simon Butler, Jonas Gamalielsson, Bjorn Lundell, Christoffer Brax, Johan Sjoberg, Anders Mattsson, Tomas Gustavsson, Jonas Feist, and Erik Lonroth. On company contributions to community open source software projects. *IEEE Transactions on Software Engineering*, pages 1–1,

2019, DOI 10.1109/tse.2019.2919305.
Abstract: The majority of contributions to community open source software (OSS) projects are made by practitioners acting on behalf of companies and other organisations. Previous research has addressed the motivations of both individuals and companies to engage with OSS projects. However, limited research has been undertaken that examines and explains the practical mechanisms or work practices used by companies and their developers to pursue their commercial and technical objectives when engaging with OSS projects. This research investigates the variety of work practices used in public communication channels by company contributors to engage with and contribute to eight community OSS projects. Through interviews with contributors to the eight projects we draw on their experiences and insights to explore the motivations to use particular methods of contribution. We find that companies utilise work practices for contributing to community projects which are congruent with the circumstances and their capabilities that support their short- and long-term needs. We also find that companies contribute to community OSS projects in ways that may not always be apparent from public sources, such as employing core project developers, making donations, and joining project steering committees in order to advance strategic interests. The factors influencing contributor work practices can be complex and are often dynamic arising from considerations such as company and project structure, as well as technical concerns and commercial strategies. The business context in which software created by the OSS project is deployed is also found to influence contributor work practices.

[**Cates2021**] Roee Cates, Nadav Yunik, and Dror G. Feitelson. Does code structure affect comprehension? on using and naming intermediate variables. In *Proc. International Conference on Program Comprehension (ICPC)*. IEEE, 5 2021, DOI 10.1109/icpc52881.2021.00020.
Abstract: Intermediate variables can be used to break complex expressions into more manageable smaller expressions, which may be easier to understand. But it is unclear when and whether this actually helps. We conducted an experiment in which subjects read 6 mathematical functions and were supposed to give them meaningful names. 113 subjects participated, of which 58% had 3 or more years of programming work experience. Each function had 3 versions: using a compound expression, using intermediate variables with meaningless names, or using intermediate variables with meaningful names. The results were that in only one case there was a significant difference between the two extreme versions, in favor of the one with intermediate variables with meaningful names. This case was the function that was the hardest to understand to begin with. In two additional cases using intermediate variables with meaningless names appears to have caused a slight decrease in understanding. In all other cases the code structure did not make much of a difference. As it is hard to anticipate what others will find difficult to understand, the conclusion is that using intermediate variables is generally desirable. However, this recommendation hinges on giving them good

names.

[**Cogo2021**] Filipe R. Cogo, Gustavo A. Oliva, Cor-Paul Bezemer, and Ahmed E. Hassan. An empirical study of same-day releases of popular packages in the npm ecosystem. *Empirical Software Engineering*, 26(5), 7 2021, DOI 10.1007/s10664-021-09980-6.

Abstract: Within a software ecosystem, client packages can reuse provider packages as third-party libraries. The reuse relation between client and provider packages is called a dependency. When a client package depends on the code of a provider package, every change that is introduced in a release of the provider has the potential to impact the client package. Since a large number of dependencies exist within a software ecosystem, releases of a popular provider package can impact a large number of clients. Occasionally, multiple releases of a popular package need to be published on the same day, leading to a scenario in which the time available to revise, test, build, and document the release is restricted compared to releases published within a regular schedule. In this paper, our objective is to study the same-day releases that are published by popular packages in the npm ecosystem. We design an exploratory study to characterize the type of changes that are introduced in same-day releases, the prevalence of same-day releases in the npm ecosystem, and the adoption of same-day releases by client packages. A preliminary manual analysis of the existing release notes suggests that same-day releases introduce non-trivial changes (e.g., bug fixes). We then focus on three RQs. First, we study how often same-day releases are published. We found that the median proportion of regularly scheduled releases that are interrupted by a same-day release (per popular package) is 22%, suggesting the importance of having timely and systematic procedures to cope with same-day releases. Second, we study the performed code changes in same-day releases. We observe that 32% of the same-day releases have larger changes compared with their prior release, thus showing that some same-day releases can undergo significant maintenance activity despite their time-constrained nature. In our third RQ, we study how client packages react to same-day releases of their providers. We observe the vast majority of client packages that adopt the release preceding the same-day release would also adopt the latter without having to change their versioning statement (implicit updates). We also note that explicit adoptions of same-day releases (i.e., adoptions that require a change to the versioning statement of the provider in question) is significantly faster than the explicit adoption of regular releases. Based on our findings, we argue that (i) third-party tools that support the automation of dependency management (e.g., Dependabot) should consider explicitly flagging same-day releases, (ii) popular packages should strive for optimized release pipelines that can properly handle same-day releases, and (iii) future research should design scalable, ecosystem-ready tools that support provider packages in assessing the impact of their code changes on client packages.

[**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.

[**Decan2021**] Alexandre Decan and Tom Mens. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*, 47(6):1226–1240, 6 2021, DOI 10.1109/tse.2019.2918315.
Abstract: The semantic versioning (semver) policy is commonly accepted by open source package management systems to inform whether new releases of software packages introduce possibly backward incompatible changes. Maintainers depending on such packages can use this information to avoid or reduce the risk of breaking changes in their own packages by specifying version constraints on their dependencies. Depending on the amount of control a package maintainer desires to have over her package dependencies, these constraints can range from very permissive to very restrictive. This article empirically compares semver compliance of four software packaging ecosystems (Cargo, npm, Packagist and Rubygems), and studies how this compliance evolves over time. We explore to what extent ecosystem-specific characteristics or policies influence the degree of compliance. We also propose an evaluation based on the "wisdom of the crowds" principle to help package maintainers decide which type of version constraints they should impose on their dependencies.

[**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.

[**Farzat2021**] Fabio de A. Farzat, Marcio de O. Barros, and Guilherme H. Travassos. Evolving JavaScript code to reduce load time. *IEEE Transactions on Software Engineering*, 47(8):1544–1558, 8 2021, DOI 10.1109/tse.2019.2928293.
Abstract: JavaScript is one of the most used programming languages for front-end development of Web applications. The increase in complexity of front-end features brings concerns about performance, especially the load and execution time of JavaScript code. In this paper, we propose an evolutionary program improvement technique to reduce the size of JavaScript programs and, therefore, the time required to load and execute them in Web applications. To guide the development of this technique, we performed an

experimental study to characterize the patches applied to JavaScript programs to reduce their size while keeping the functionality required to pass all test cases in their test suites. We applied this technique to 19 JavaScript programs varying from 92 to 15,602 LOC and observed reductions from 0.2 to 73.8 percent of the original code, as well as a relationship between the quality of a program's test suite and the ability to reduce the size of its source code.

[**Foundjem2021**] Armstrong Foundjem and Bram Adams. Release synchronization in software ecosystems. *Empirical Software Engineering*, 26(3), 3 2021, DOI 10.1007/s10664-020-09929-1.
Abstract: Software ecosystems bring value by integrating software projects related to a given domain, such as Linux distributions integrating upstream open-source projects or the Android ecosystem for mobile Apps. Since each project within an ecosystem may potentially have its release cycle and roadmap, this creates an enormous burden for users who must expend the effort to identify and install compatible project releases from the ecosystem manually. Thus, many ecosystems, such as the Linux distributions, take it upon them to release a polished, well-integrated product to the end-user. However, the body of knowledge lacks empirical evidence about the coordination and synchronization efforts needed at the ecosystem level to ensure such federated releases. This paper empirically studies the strategies used to synchronize releases of ecosystem projects in the context of the OpenStack ecosystem, in which a central release team manages the six-month release cycle of the overall OpenStack ecosystem product. We use qualitative analysis on the release team's IRC-meeting logs that comprise two OpenStack releases (one-year long). Thus, we identified, cataloged, and documented ten major release synchronization activities, which we further validated through interviews with eight active OpenStack senior practitioners (members of either the release team or project teams). Our results suggest that even though an ecosystem's power lies in the interaction of inter-dependent projects, release synchronization remains a challenge for both the release team and the project teams. Moreover, we found evidence (and reasons) of multiple release strategies co-existing within a complex ecosystem.

[**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.

[**Glanz2020**] Leonid Glanz, Patrick Müller, Lars Baumgärtner, Michael Reif, Sven Amann, Pauline Anthonysamy, and Mira Mezini. Hidden in plain sight: Obfuscated strings threatening your privacy. In *Proc. Asia Conference on Computer and Communications Security (ACCCS)*. ACM, 10 2020, DOI `10.1145/3320269.3384745`.
Abstract: String obfuscation is an established technique used by proprietary, closed-source applications to protect intellectual property. Furthermore, it is also frequently used to hide spyware or malware in applications.

In both cases, the techniques range from bit-manipulation over XOR operations to AES encryption. However, string obfuscation techniques/tools suffer from one shared weakness: They generally have to embed the necessary logic to deobfuscate strings into the app code. In this paper, we show that most of the string obfuscation techniques found in malicious and benign applications for Android can easily be broken in an automated fashion. We developed StringHound, an open-source tool that uses novel techniques that identify obfuscated strings and reconstruct the originals using slicing. We evaluated StringHound on both benign and malicious Android apps. In summary, we deobfuscate almost 30 times more obfuscated strings than other string deobfuscation tools. Additionally, we analyzed 100,000 Google Play Store apps and found multiple obfuscated strings that hide vulnerable cryptographic usages, insecure internet accesses, API keys, hard-coded passwords, and exploitation of privileges without the awareness of the developer. Furthermore, our analysis reveals that not only malware uses string obfuscation but also benign apps make extensive use of string obfuscation.

[**Gujral2021**] Harshit Gujral, Sangeeta Lal, and Heng Li. An exploratory semantic analysis of logging questions. *Journal of Software: Evolution and Process*, 33(7), 6 2021, DOI `10.1002/smr.2361`.

Abstract: Logging is an integral part of software development. Software practitioners often face issues in software logging, and they post these issues on Q&A websites to take suggestions from the experts. In this study, we perform a three-level empirical analysis of logging questions posted on six popular technical Q&A websites, namely, Stack Overflow (SO), Serverfault (SF), Superuser (SU), Database Administrators (DB), Software Engineering (SE), and Android Enthusiasts (AE). The findings show that logging issues are prevalent across various domains, for example, database, networks, and mobile computing, and software practitioners from different domains face different logging issues. The semantic analysis of logging questions using Latent Dirichlet Allocation (LDA) reveals trends of several existing and new logging topics, such as logging conversion pattern, Android device logging, and database logging. In addition, we observe specific logging topics for each website: DB (log shipping and log file growing/shrinking), SU (event log and syslog configuration), SF (log analysis and syslog configuration), AE (app install and usage tracking), SE (client server logging and exception logging), and SO (log file creation/deletion, Android emulator logging, and logger class of Log4j). We obtain an increasing trend of logging topics on the SO, SU, and DB websites whereas a decreasing trend of logging topics on the SF website.

[**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.

[**Hoyos2021**] Juan Hoyos, Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Albeiro Espinosa Bedoya. On the removal of feature toggles: A study of python projects and practitioners motivations. *Empirical Software Engineering*, 26(2), 2 2021, DOI 10.1007/s10664-020-09902-y.
Abstract: Feature Toggling is a technique to control the execution of features in a software project. For example, practitioners using feature toggles can experiment with new features in a production environment by exposing them to a subset of users. Some of these toggles require additional maintainability efforts and are expected to be removed, whereas others are meant to remain for a long time. However, to date, very little is known about the removal of feature toggles, which is why we focus on this topic in our paper. We conduct an empirical study that focuses on the removal of feature toggles. We use source code analysis techniques to analyze 12 Python open source projects and surveyed 61 software practitioners to provide deeper insights on the topic. Our study shows that 75% of the toggle components in the studied Python projects are removed within 49 weeks after introduction. However, eventually practitioners remove feature toggles to follow the life cycle of a feature when it becomes stable in production. We also find that not all long-term feature toggles are designed to live that long and not all feature toggles are removed from the source code, opening the possibilities to unwanted risks. Our study broadens the understanding of feature toggles by identifying reasons for their survival in practice and aims to help practitioners make better decisions regarding the way they manage and remove feature toggles.

[**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.

[**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.

[**Johnson2021**] Brittany Johnson, Thomas Zimmermann, and Christian Bird. The effect of work environments on productivity and satisfaction of software engineers. *IEEE Transactions on Software Engineering*, 47(4):736–757, 4 2021, DOI 10.1109/tse.2019.2903053.

Abstract: The physical work environment of software engineers can have various effects on their satisfaction and the ability to get the work done. To better understand the factors of the environment that affect productivity and satisfaction of software engineers, we explored different work environments at Microsoft. We used a mixed-methods, multiple stage research design with a total of 1,159 participants: two surveys with 297 and 843 responses respectively and interviews with 19 employees. We found several factors that were considered as important for work environments: personalization, social norms and signals, room composition and atmosphere, work-related environment affordances, work area and furniture, and productivity strategies. We built statistical models for satisfaction with the work environment and perceived productivity of software engineers and compared them to models for employees in the Program Management, IT Operations, Marketing, and Business Program & Operations disciplines. In the satisfaction models, the ability to work privately with no interruptions and the ability to communicate with the team and leads were important factors among all disciplines. In the productivity models, the overall satisfaction with the work environment and the ability to work privately with no interruptions were important factors among all disciplines. For software engineers, another important factor for perceived productivity was the ability to communicate with the team and leads. We found that private offices were linked to higher perceived productivity across all disciplines.

[**Jolak2020**] Rodi Jolak, Maxime Savary-Leblanc, Manuela Dalibor, Andreas Wortmann, Regina Hebig, Juraj Vincur, Ivan Polasek, Xavier Le Pallec, Sébastien Gérard, and Michel R. V. Chaudron. Software engineering whispers: The effect of textual vs. graphical software design descriptions on software design communication. *Empirical Software Engineering*, 25(6):4427–4471, 9 2020, DOI `10.1007/s10664-020-09835-6`.

Abstract: Software engineering is a social and collaborative activity. Communicating and sharing knowledge between software developers requires much effort. Hence, the quality of communication plays an important role in influencing project success. To better understand the effect of communication on project success, more in-depth empirical studies investigating this phenomenon are needed. We investigate the effect of using a graphical versus textual design description on co-located software design communication. Therefore, we conducted a family of experiments involving a mix of 240 software engineering students from four universities. We examined how different design representations (i.e., graphical vs. textual) affect the ability to Explain, Understand, Recall, and Actively Communicate knowledge. We found that the graphical design description is better than the textual in promoting Active Discussion between developers and improving the Recall of design details. Furthermore, compared to its unaltered version, a well-organized and motivated textual design description—that is used for the same amount of time—enhances the recall of design details and increases the amount of active discussions at the cost of reducing the perceived quality of explaining.

[**Kim2021**] Dong Jae Kim, Tse-Hsun Chen, and Jinqiu Yang. The secret life of test smells—an empirical study on test smell evolution and maintenance. *Empirical Software Engineering*, 26(5), 7 2021, DOI 10.1007/s10664-021-09969-1.

Abstract: In recent years, researchers and practitioners have been studying the impact of test smells in test maintenance. However, there is still limited empirical evidence on why developers remove test smells in software maintenance and the mechanism employed for addressing test smells. In this paper, we conduct an empirical study on 12 real-world open-source systems to study the evolution and maintenance of test smells and how test smells are related to software quality. Results show that: 1) Although the number of test smell instances increases, test smell density decreases as systems evolve. 2) However, our qualitative analysis on those removed test smells reveals that most test smell removal (83%) is a by-product of feature maintenance activities. 45% of the removed test smells relocate to other test cases due to refactoring, while developers deliberately address the only 17% of test smells, consisting of largely Exception Catch/Throw and Sleepy Test. 3) Our statistical model shows that test smell metrics can provide additional explanatory power on post-release defects over traditional baseline metrics (an average of 8.25% increase in AUC). However, most types of test smells have a minimal effect on post-release defects. Our study provides insight into developers' perception of test smells and current practices. Future studies on test smells may consider focusing on the specific types of test smells that may have a higher correlation with defect-proneness when helping developers with test code maintenance.

[**Kochhar2019**] Pavneet Singh Kochhar, Eirini Kalliamvakou, Nachiappan Nagappan, Thomas Zimmermann, and Christian Bird. Moving from closed to open source: Observations from six transitioned projects to GitHub. *IEEE Transactions on Software Engineering*, pages 1–1, 2019, DOI 10.1109/tse.2019.2937025.

Abstract: Open source software systems have gained a lot of attention in the past few years. With the emergence of open source platforms like GitHub, developers can contribute, store, and manage their projects with ease. Large organizations like Microsoft, Google, and Facebook are open sourcing their in-house technologies in an effort to more broadly involve the community in the development of software systems. Although closed source and open source systems have been studied extensively, there has been little research on the transition from closed source to open source systems. Through this study we aim to: a) provide guidance and insights for other teams planning to open source their projects and b) to help them avoid pitfalls during the transition process. We studied six different Microsoft systems, which were recently open-sourced i.e., CoreFX, CoreCLR, Roslyn, Entity Framework, MVC, and Orleans. This paper presents the transition from the viewpoints of both Microsoft and the open source community based on interviews with eleven Microsoft developer, five Microsoft senior managers involved in the

decision to open source, and eleven open-source developers. From Microsoft's perspective we discuss the reasons for the transition, experiences of developers involved, and the transition's outcomes and challenges. Our results show that building a vibrant community, prompt answers, developing an open source culture, security regulations and business opportunities are the factors which persuade companies to open source their products. We also discuss the transition outcomes on processes such as code reviews, version control systems, continuous integration as well as developers' perception of these changes. From the open source community's perspective, we illustrate the response to the open-sourcing initiative through contributions and interactions with the internal developers and provide guidelines for other projects planning to go open source.

[**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 resonance 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.

[**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.

[**Lee2020a**] Daniel Lee, Dayi Lin, Cor-Paul Bezemer, and Ahmed E. Hassan. Building the perfect game – an empirical study of game modifications. *Empirical Software Engineering*, 25(4):2485–2518, 3 2020, DOI `10.1007/s10664-019-09783-w`.

Abstract: Prior work has shown that gamer loyalty is important for the sales of a developer's future games. Therefore, it is important for game developers to increase the longevity of their games. However, game developers cannot always meet the growing and changing needs of the gaming community, due to the often already overloaded schedules of developers. So-called modders can potentially assist game developers with addressing

gamers' needs. Modders are enthusiasts who provide modifications or completely new content for a game. By supporting modders, game developers can meet the rapidly growing and varying needs of their gamer base. Modders have the potential to play a role in extending the life expectancy of a game, thereby saving game developers time and money, and leading to a better overall gaming experience for their gamer base. In this paper, we empirically study the metadata of 9,521 mods that were extracted from the Nexus Mods distribution platform. The Nexus Mods distribution platform is one of the largest mod distribution platforms for PC games at the time of our study. The goal of our paper is to provide useful insights about mods on the Nexus Mods distribution platform from a quantitative perspective, and to provide researchers a solid foundation to further explore game mods. To better understand the potential of mods to extend the longevity of a game we study their characteristics, and we study their release schedules and post-release support (in terms of bug reports) as a proxy for the willingness of the modding community to contribute to a game. We find that providing official support for mods can be beneficial for the perceived quality of the mods of a game: games for which a modding tool is provided by the original game developer have a higher median endorsement ratio than mods for games that do not have such a tool. In addition, mod users are willing to submit bug reports for a mod. However, they often fail to do this in a systematic manner using the bug reporting tool of the Nexus Mods platform, resulting in low-quality bug reports which are difficult to resolve. Our findings give the first insights into the characteristics, release schedule and post-release support of game mods. Our findings show that some games have a very active modding community, which contributes to those games through mods. Based on our findings, we recommend that game developers who desire an active modding community for their own games provide the modding community with an officially-supported modding tool. In addition, we recommend that mod distribution platforms, such as Nexus Mods, improve their bug reporting system to receive higher quality bug reports.

[**Lee2020b**] Daniel Lee, Gopi Krishnan Rajbahadur, Dayi Lin, Mohammed Sayagh, Cor-Paul Bezemer, and Ahmed E. Hassan. An empirical study of the characteristics of popular minecraft mods. *Empirical Software Engineering*, 25(5):3396–3429, 8 2020, DOI `10.1007/s10664-020-09840-9`.
Abstract: It is becoming increasingly difficult for game developers to manage the cost of developing a game, while meeting the high expectations of gamers. One way to balance the increasing gamer expectation and development stress is to build an active modding community around the game. There exist several examples of games with an extremely active and successful modding community, with the Minecraft game being one of the most notable ones. This paper reports on an empirical study of 1,114 popular and 1,114 unpopular Minecraft mods from the CurseForge mod distribution platform, one of the largest distribution platforms for Minecraft mods. We analyzed the relationship between 33 features across 5 dimensions of mod

characteristics and the popularity of mods (i.e., mod category, mod documentation, environmental context of the mod, remuneration for the mod, and community contribution for the mod), to understand the characteristics of popular Minecraft mods. We firstly verify that the studied dimensions have significant explanatory power in distinguishing the popularity of the studied mods. Then we evaluated the contribution of each of the 33 features across the 5 dimensions. We observed that popular mods tend to have a high quality description and promote community contribution.

[**Lima2021a**] Luan P. Lima, Lincoln S. Rocha, Carla I. M. Bezerra, and Matheus Paixao. Assessing exception handling testing practices in open-source libraries. *Empirical Software Engineering*, 26(5), 6 2021, DOI 10.1007/s10664-021-09983-3.

Abstract: Modern programming languages (e.g., Java and C#) provide features to separate error-handling code from regular code, seeking to enhance software comprehensibility and maintainability. Nevertheless, the way exception handling (EH) code is structured in such languages may lead to multiple, different, and complex control flows, which may affect the software testability. Previous studies have reported that EH code is typically neglected, not well tested, and its misuse can lead to reliability degradation and catastrophic failures. However, little is known about the relationship between testing practices and EH testing effectiveness. In this exploratory study, we (i) measured the adequacy degree of EH testing concerning code coverage (instruction, branch, and method) criteria; and (ii) evaluated the effectiveness of the EH testing by measuring its capability to detect artificially injected faults (i.e., mutants) using 7 EH mutation operators. Our study was performed using test suites of 27 long-lived Java libraries from open-source ecosystems. Our results show that instructions and branches within catch blocks and throw instructions are less covered, with statistical significance, than the overall instructions and branches. Nevertheless, most of the studied libraries presented test suites capable of detecting more than 70% of the injected faults. From a total of 12, 331 mutants created in this study, the test suites were able to detect 68% of them.

[**Liu2021**] Kui Liu, Dongsun Kim, Tegawende F. Bissyande, Shin Yoo, and Yves Le Traon. Mining fix patterns for FindBugs violations. *IEEE Transactions on Software Engineering*, 47(1):165–188, 1 2021, DOI 10.1109/tse.2018.2884955.

Abstract: Several static analysis tools, such as Splint or FindBugs, have been proposed to the software development community to help detect security vulnerabilities or bad programming practices. However, the adoption of these tools is hindered by their high false positive rates. If the false positive rate is too high, developers may get acclimated to violation reports from these tools, causing concrete and severe bugs being overlooked. Fortunately, some violations are actually addressed and resolved by developers. We claim that those violations that are recurrently fixed are likely to be

true positives, and an automated approach can learn to repair similar unseen violations. However, there is lack of a systematic way to investigate the distributions on existing violations and fixed ones in the wild, that can provide insights into prioritizing violations for developers, and an effective way to mine code and fix patterns which can help developers easily understand the reasons of leading violations and how to fix them. In this paper, we first collect and track a large number of fixed and unfixed violations across revisions of software. The empirical analyses reveal that there are discrepancies in the distributions of violations that are detected and those that are fixed, in terms of occurrences, spread and categories, which can provide insights into prioritizing violations. To automatically identify patterns in violations and their fixes, we propose an approach that utilizes convolutional neural networks to learn features and clustering to regroup similar instances. We then evaluate the usefulness of the identified fix patterns by applying them to unfixed violations. The results show that developers will accept and merge a majority (69/116) of fixes generated from the inferred fix patterns. It is also noteworthy that the yielded patterns are applicable to four real bugs in the Defects4J major benchmark for software testing and automated repair.

[**Macho2021**] Christian Macho, Stefanie Beyer, Shane McIntosh, and Martin Pinzger. The nature of build changes: An empirical study of maven-based build systems. *Empirical Software Engineering*, 26(3), 3 2021, DOI 10.1007/s10664-020-09926-4.

Abstract: Build systems are an essential part of modern software projects. As software projects change continuously, it is crucial to understand how the build system changes because neglecting its maintenance can, at best, lead to expensive build breakage, or at worst, introduce user-reported defects due to incorrectly compiled, linked, packaged, or deployed official releases. Recent studies have investigated the (co-)evolution of build configurations and reasons for build breakage; however, the prior analysis focused on a coarse-grained outcome (i.e., either build changing or not). In this paper, we present BUILDDIFF, an approach to extract detailed build changes from MAVEN build files and classify them into 143 change types. In a manual evaluation of 400 build-changing commits, we show that BUILDDIFF can extract and classify build changes with average precision, recall, and f1-scores of 0.97, 0.98, and 0.97, respectively. We then present two studies using the build changes extracted from 144 open source Java projects to study the frequency and time of build changes. The results show that the top-10 most frequent change types account for 51% of the build changes. Among them, changes to version numbers and changes to dependencies of the projects occur most frequently. We also observe frequently co-occurring changes, such as changes to the source code management definitions, and corresponding changes to the dependency management system and the dependency declaration. Furthermore, our results show that build changes frequently occur around release days. In particular, critical changes, such as updates to plugin configuration parts and dependency insertions, are performed before a release day. The

contributions of this paper lay in the foundation for future research, such as for analyzing the (co-)evolution of build files with other artifacts, improving effort estimation approaches by incorporating necessary modifications to the build system specification, or automatic repair approaches for configuration code. Furthermore, our detailed change information enables improvements of refactoring approaches for build configurations and improvements of prediction models to identify error-prone build files.

[**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.

[**Mo2021**] Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng. Architecture anti-patterns: Automatically detectable violations of design principles. *IEEE Transactions on Software Engineering*, 47(5):1008–1028, 5 2021, DOI 10.1109/tse.2019.2910856.

Abstract: In large-scale software systems, error-prone or change-prone files rarely stand alone. They are typically architecturally connected and their connections usually exhibit architecture problems causing the propagation of error-proneness or change-proneness. In this paper, we propose and empirically validate a suite of architecture anti-patterns that occur in all large-scale software systems and are involved in high maintenance costs. We define

these architecture anti-patterns based on fundamental design principles and Baldwin and Clark's design rule theory. We can automatically detect these anti-patterns by analyzing a project's structural relationships and revision history. Through our analyses of 19 large-scale software projects, we demonstrate that these architecture anti-patterns have significant impact on files' bug-proneness and change-proneness. In particular, we show that 1) files involved in these architecture anti-patterns are more error-prone and change-prone; 2) the more anti-patterns a file is involved in, the more error-prone and change-prone it is; and 3) while all of our defined architecture anti-patterns contribute to file's error-proneness and change-proneness, Unstable Interface and Crossing contribute the most by far.

[**Moraes2021**] João Pedro Moraes, Ivanilton Polato, Igor Wiese, Filipe Saraiva, and Gustavo Pinto. From one to hundreds: multi-licensing in the JavaScript ecosystem. *Empirical Software Engineering*, 26(3), 3 2021, DOI 10.1007/s10664-020-09936-2.

Abstract: Open source licenses create a legal framework that plays a crucial role in the widespread adoption of open source projects. Without a license, any source code available on the internet could not be openly (re)distributed. Although recent studies provide evidence that most popular open source projects have a license, developers might lack confidence or expertise when they need to combine software licenses, leading to a mistaken project license unification.This license usage is challenged by the high degree of reuse that occurs in the heart of modern software development practices, in which third-party libraries and frameworks are easily and quickly integrated into a software codebase.This scenario creates what we call "multi-licensed" projects, which happens when one project has components that are licensed under more than one license. Although these components exist at the file-level, they naturally impact licensing decisions at the project-level. In this paper, we conducted a mix-method study to shed some light on these questions. We started by parsing 1,426,263 (source code and non-source code) files available on 1,552 JavaScript projects, looking for license information. Among these projects, we observed that 947 projects (61%) employ more than one license. On average, there are 4.7 licenses per studied project (max: 256). Among the reasons for multi-licensing is to incorporate the source code of third-party libraries into the project's codebase. When doing so, we observed that 373 of the multi-licensed projects introduced at least one license incompatibility issue. We also surveyed with 83 maintainers of these projects aimed to cross-validate our findings. We observed that 63% of the surveyed maintainers are not aware of the multi-licensing implications. For those that are aware, they adopt multiple licenses mostly to conform with third-party libraries' licenses.

[**MoreiraSoares2020**] Daricélio Moreira Soares, Manoel Limeira Lima Júnior, Leonardo Murta, and Alexandre Plastino. What factors influence the lifetime of pull requests? *Software: Practice and Experience*, 51(6):1173–1193, 12 2020, DOI 10.1002/spe.2946.

Abstract: When external contributors want to collaborate with an open-source project, they fork the repository, make changes, and send a pull request to the core team. However, the lifetime of a pull request, defined by the time interval between its opening and its closing, has a high variation, potentially affecting the contributor engagement. In this context, understanding the root causes of pull request lifetime is important to both the external contributors and the core team. The former can adopt strategies that increase the chances of fast review, while the latter can establish priorities in the reviewing process, alleviating the pending tasks and improving the software quality. In this work, we mined association rules from 97,463 pull requests from 30 projects in order to find characteristics that have affected the pull requests lifetime. In addition, we present a qualitative analysis, helping to understand the patterns discovered from the association rules. The results indicate that: (i) contributions with shorter lifetimes tend to be accepted; (ii) structural characteristics, such as number of commits, changed files, and lines of code, have influence, in an isolated or combined way, on the pull request lifetime; (iii) the files changed and the directories to which they belong can be robust predictors for pull request lifetime; (iv) the profile of external contributors and their social relationships have influence on lifetime; and (v) the number of comments in a pull request, as well as the developer responsible for the review, are important predictors for its lifetime.

[**NguyenDuc2021**] Anh Nguyen-Duc, Kai-Kristian Kemell, and Pekka Abrahamsson. The entrepreneurial logic of startup software development: A study of 40 software startups. *Empirical Software Engineering*, 26(5), 7 2021, DOI 10.1007/s10664-021-09987-z.

Abstract: Context: Software startups are an essential source of innovation and software-intensive products. The need to understand product development in startups and to provide relevant support are highlighted in software research. While state-of-the-art literature reveals how startups develop their software, the reasons why they adopt these activities are underexplored. Objective: This study investigates the tactics behind software engineering (SE) activities by analyzing key engineering events during startup journeys. We explore how entrepreneurial mindsets may be associated with SE knowledge areas and with each startup case. Method: Our theoretical foundation is based on causation and effectuation models. We conducted semi-structured interviews with 40 software startups. We used two-round open coding and thematic analysis to describe and identify entrepreneurial software development patterns. Additionally, we calculated an effectuation index for each startup case. Results: We identified 621 events merged into 32 codes of entrepreneurial logic in SE from the sample. We found a systemic occurrence of the logic in all areas of SE activities. Minimum Viable Product (MVP), Technical Debt (TD), and Customer Involvement (CI) tend to be associated with effectual logic, while testing activities at different levels are associated with causal logic. The effectuation index revealed that startups are either effectuation-driven or mixed-logics-driven. Conclusions: Software startups

fall into two types that differentiate between how traditional SE approaches may apply to them. Effectuation seems the most relevant and essential model for explaining and developing suitable SE practices for software startups.

[**Paltoglou2021**] Katerina Paltoglou, Vassilis E. Zafeiris, N.A. Diamantidis, and E.A. Giakoumakis. Automated refactoring of legacy JavaScript code to ES6 modules. *Journal of Systems and Software*, 181:111049, 11 2021, DOI `10.1016/j.jss.2021.111049`.
Abstract: The JavaScript language did not specify, until ECMAScript 6 (ES6), native features for streamlining encapsulation and modularity. Developer community filled the gap with a proliferation of design patterns and module formats, with impact on code reusability, portability and complexity of build configurations. This work studies the automated refactoring of legacy ES5 code to ES6 modules with fine-grained reuse of module contents through the named import/export language constructs. The focus is on reducing the coupling of refactored modules through destructuring exported module objects to fine-grained module features and enhancing module dependencies by leveraging the ES6 syntax. We employ static analysis to construct a model of a JavaScript project, the Module Dependence Graph (MDG), that represents modules and their dependencies. On the basis of MDG we specify the refactoring procedure for module migration to ES6. A prototype implementation has been empirically evaluated on 19 open source projects. Results highlight the relevance of the refactoring with a developer intent for fine-grained reuse. The analysis of refactored code shows an increase in the number of reusable elements per project and reduction in the coupling of refactored modules. The soundness of the refactoring is empirically validated through code inspection and execution of projects' test suites.

[**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).

[**Rahman2021**] Akond Rahman, Md Rayhanur Rahman, Chris Parnin, and Laurie Williams. Security smells in ansible and chef scripts. *ACM Transactions on Software Engineering and Methodology*, 30(1):1–31, 1 2021, DOI 10.1145/3408897.

Abstract: Context: Security smells are recurring coding patterns that are indicative of security weakness and require further inspection. As infrastructure as code (IaC) scripts, such as Ansible and Chef scripts, are used to provision cloud-based servers and systems at scale, security smells in IaC scripts could be used to enable malicious users to exploit vulnerabilities in the provisioned systems. Goal: The goal of this article is to help practitioners avoid insecure coding practices while developing infrastructure as code scripts through an empirical study of security smells in Ansible and Chef scripts. Methodology: We conduct a replication study where we apply qualitative analysis with 1,956 IaC scripts to identify security smells for IaC scripts written in two languages: Ansible and Chef. We construct a static analysis tool called Security Linter for Ansible and Chef scripts (SLAC) to automatically identify security smells in 50,323 scripts collected from 813 open source software repositories. We also submit bug reports for 1,000 randomly selected smell occurrences. Results: We identify two security smells not reported in prior work: missing default in case statement and no integrity check. By applying SLAC we identify 46,600 occurrences of security smells that include 7,849 hard-coded passwords. We observe agreement for 65 of the responded 94 bug reports, which suggests the relevance of security smells for Ansible and Chef scripts amongst practitioners. Conclusion: We observe security smells to be prevalent in Ansible and Chef scripts, similarly to that of the Puppet scripts. We recommend practitioners to rigorously inspect the presence of the identified security smells in Ansible and Chef scripts using (i) code review, and (ii) static analysis tools.

[**RakAmnouykit2020**] Ingkarat Rak-amnouykit, Daniel McCrevan, Ana Milanova, Martin Hirzel, and Julian Dolby. Python 3 types in the wild: a tale of two type systems. In *Proc. International Symposium on Dynamic Languages (ISDL)*. ACM, 11 2020, DOI 10.1145/3426422.3426981.

Abstract: Python 3 is a highly dynamic language, but it has introduced a syntax for expressing types with PEP484. This paper ex- plores how developers use these type annotations, the type system semantics provided by type checking and inference tools, and the performance of these tools. We evaluate the types and tools on a corpus of public GitHub repositories. We review MyPy and PyType, two canonical static type checking and inference tools, and their distinct approaches to type analysis. We then address three research questions: (i) How often and in what ways do developers use Python 3 types? (ii) Which type errors do developers make? (iii) How

do type errors from different tools compare? Surprisingly, when developers use static types, the code rarely type-checks with either of the tools. MyPy and PyType exhibit false positives, due to their static nature, but also flag many useful errors in our corpus. Lastly, MyPy and PyType embody two distinct type systems, flagging different errors in many cases. Understanding the usage of Python types can help guide tool-builders and researchers. Understanding the performance of popular tools can help increase the adoption of static types and tools by practitioners, ultimately leading to more correct and more robust Python code.

[**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.

[**RodriguezPerez2020**] Gema Rodríguez-Pérez, Gregorio Robles, Alexander Serebrenik, Andy Zaidman, Daniel M. Germán, and Jesus M. Gonzalez-Barahona. How bugs are born: a model to identify how bugs are introduced in software components. *Empirical Software Engineering*, 25(2):1294–1340, 2 2020, DOI `10.1007/s10664-019-09781-y`.

Abstract: When identifying the origin of software bugs, many studies assume that "a bug was introduced by the lines of code that were modified to fix it". However, this assumption does not always hold and at least in some cases, these modified lines are not responsible for introducing the bug. For example, when the bug was caused by a change in an external API. The lack of empirical evidence makes it impossible to assess how important these cases are and therefore, to which extent the assumption is valid. To advance in this direction, and better understand how bugs "are born", we propose a model for defining criteria to identify the first snapshot of an evolving software system that exhibits a bug. This model, based on the perfect test idea, decides whether a bug is observed after a change to the software. Furthermore, we studied the model's criteria by carefully analyzing how 116

bugs were introduced in two different open source software projects. The manual analysis helped classify the root cause of those bugs and created manually curated datasets with bug-introducing changes and with bugs that were not introduced by any change in the source code. Finally, we used these datasets to evaluate the performance of four existing SZZ-based algorithms for detecting bug-introducing changes. We found that SZZ-based algorithms are not very accurate, especially when multiple commits are found; the F-Score varies from 0.44 to 0.77, while the percentage of true positives does not exceed 63%. Our results show empirical evidence that the prevalent assumption, "a bug was introduced by the lines of code that were modified to fix it", is just one case of how bugs are introduced in a software system. Finding what introduced a bug is not trivial: bugs can be introduced by the developers and be in the code, or be created irrespective of the code. Thus, further research towards a better understanding of the origin of bugs in software projects could help to improve design integration tests and to design other procedures to make software development more robust.

[**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.

[**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 performance antipatterns in database-backed web applications. To date, no ex-

isting 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.

[**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.

[**SotoValero2021**] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. A comprehensive study of bloated dependencies in the maven ecosystem. *Empirical Software Engineering*, 26(3), 3 2021, DOI 10.1007/s10664-020-09914-8.

Abstract: Build automation tools and package managers have a profound influence on software development. They facilitate the reuse of third-party libraries, support a clear separation between the application's code and its external dependencies, and automate several software development tasks. However, the wide adoption of these tools introduces new challenges related to dependency management. In this paper, we propose an original study

of one such challenge: the emergence of bloated dependencies. Bloated dependencies are libraries that the build tool packages with the application's compiled code but that are actually not necessary to build and run the application. This phenomenon artificially grows the size of the built binary and increases maintenance effort. We propose a tool, called DepClean, to analyze the presence of bloated dependencies in Maven artifacts. We analyze 9,639 Java artifacts hosted on Maven Central, which include a total of 723,444 dependency relationships. Our key result is that 75.1% of the analyzed dependency relationships are bloated. In other words, it is feasible to reduce the number of dependencies of Maven artifacts up to 1/4 of its current count. We also perform a qualitative study with 30 notable open-source projects. Our results indicate that developers pay attention to their dependencies and are willing to remove bloated dependencies: 18/21 answered pull requests were accepted and merged by developers, removing 131 dependencies in total.

[**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.

[**Tomasdottir2020**] Kristín Fjóla Tómasdóttir, Maurício Aniche, and Arie van Deursen. The adoption of JavaScript linters in practice: A case study on ESLint. *IEEE Transactions on Software Engineering*, 46(8):863–891, 8 2020, DOI 10.1109/tse.2018.2871058.

Abstract: A linter is a static analysis tool that warns software developers about possible code errors or violations to coding standards. By using such a tool, errors can be surfaced early in the development process when they are cheaper to fix. For a linter to be successful, it is important to understand the needs and challenges of developers when using a linter. In this paper, we examine developers' perceptions on JavaScript linters. We study why and how developers use linters along with the challenges they face while using such tools. For this purpose we perform a case study on ESLint, the most popular JavaScript linter. We collect data with three different methods where we interviewed 15 developers from well-known open source projects, analyzed over 9,500 ESLint configuration files, and surveyed 337 developers from the JavaScript community. Our results provide practitioners with reasons for using linters in their JavaScript projects as well as several configuration strategies and their advantages. We also provide a list of linter rules that are often enabled and disabled, which can be interpreted as the most important rules to reason about when configuring linters. Finally, we propose several feature suggestions for tool makers and future work for researchers.

[**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.

[**Venigalla2021**] Akhila Sri Manasa Venigalla and Sridhar Chimalakonda. On the comprehension of application programming interface usability in game engines. *Software: Practice and Experience*, 51(8):1728–1744, 5 2021, DOI 10.1002/spe.2985.

Abstract: Extensive development of games for various purposes including education and entertainment has resulted in increased development of game engines. Game engines are being used on a large scale as they support and simplify game development to a greater extent. Game developers using game engines are often compelled to use various application programming interfaces (APIs) of game engines in the process of game development. Thus, both quality and ease of development of games are greatly influenced by APIs defined in game engines. Hence, understanding API usability in game engines could greatly help in choosing better game engines among the ones that are available for game development and also could help developers in designing better game engines. In this article, we thus aim to evaluate API usability of 95 publicly available game engine repositories on GitHub, written primarily in C++ programming language. We test API usability of these game engines against the eight structural API usability metrics—AMNOI, AMNCI, AMGI, APXI, APLCI, AESI, ATSI, and ADI. We see this research as a first step toward the direction of improving usability of APIs in game engines. We present the results of the study, which indicate that about 25% of the game engines considered have minimal API usability, with respect to the considered metrics. It was observed that none of the considered repositories have ideal (all metric scores equal to 1) API usability, indicating the need for developers to consider API usability metrics while designing game engines.

[**Weintrop2017**] David Weintrop and Uri Wilensky. Comparing block-based and text-based programming in high school computer science classrooms. *ACM Transactions on Computing Education*, 18(1):1–25, 12 2017, DOI 10.1145/3089799.

Abstract: The number of students taking high school computer science classes is growing. Increasingly, these students are learning with graphical, block-based programming environments either in place of or prior to traditional text-based programming languages. Despite their growing use in formal settings, relatively little empirical work has been done to understand the impacts of using block-based programming environments in high school classrooms. In this article, we present the results of a 5-week, quasi-experimental study comparing isomorphic block-based and text-based programming environments in an introductory high school programming class. The findings from this study show students in both conditions improved their scores between pre- and postassessments; however, students in the blocks condition showed greater learning gains and a higher level of interest in future computing courses. Students in the text condition viewed their programming experience as more similar to what professional programmers do and as more effective at improving their programming ability. No difference was found between students in the two conditions with respect to confidence or enjoyment. The implications of these findings with respect to pedagogy and design are discussed, along with directions for future work.

[**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.

[**Zhang2021b**] Haoxiang Zhang, Shaowei Wang, Tse-Hsun Chen, Ying Zou, and Ahmed E. Hassan. An empirical study of obsolete answers on stack over-

flow. *IEEE Transactions on Software Engineering*, 47(4):850–862, 4 2021, DOI 10.1109/tse.2019.2906315.
Abstract: Stack Overflow accumulates an enormous amount of software engineering knowledge. However, as time passes, certain knowledge in answers may become obsolete. Such obsolete answers, if not identified or documented clearly, may mislead answer seekers and cause unexpected problems (e.g., using an out-dated security protocol). In this paper, we investigate how the knowledge in answers becomes obsolete and identify the characteristics of such obsolete answers. We find that: 1) More than half of the obsolete answers (58.4 percent) were probably already obsolete when they were first posted. 2) When an obsolete answer is observed, only a small proportion (20.5 percent) of such answers are ever updated. 3) Answers to questions in certain tags (e.g., node.js, ajax, android, and objective-c) are more likely to become obsolete. Our findings suggest that Stack Overflow should develop mechanisms to encourage the whole community to maintain answers (to avoid obsolete answers) and answer seekers are encouraged to carefully go through all information (e.g., comments) in answer threads.