It Will Never Work in Theory

http://neverworkintheory.org

October 5, 2021

References

[**Abad2018**] Zahra Shakeri Hossein Abad, Oliver Karras, Kurt Schneider, Ken Barker, and Mike Bauer. Task interruption in software development projects. In *Proc. International Conference on Evaluation and Assessment in Software Engineering (EASE)*. ACM, 6 2018, DOI `10.1145/3210459.3210471`.

Abstract: Multitasking has always been an inherent part of software development and is known as the primary source of interruptions due to task switching in software development teams. Developing software involves a mix of analytical and creative work, and requires a significant load on brain functions, such as working memory and decision making. Thus, task switching in the context of software development imposes a cognitive load that causes software developers to lose focus and concentration while working thereby taking a toll on productivity. To investigate the disruptiveness of task switching and interruptions in software development projects, and to understand the reasons for and perceptions of the disruptiveness of task switching we used a mixed-methods approach including a longitudinal data analysis on 4,910 recorded tasks of 17 professional software developers, and a survey of 132 software developers. We found that, compared to task-specific factors (e.g. priority, level, and temporal stage), contextual factors such as interruption type (e.g. self/external), time of day, and task type and context are a more potent determinant of task switching disruptiveness in software development tasks. Furthermore, while most survey respondents believe external interruptions are more disruptive than self-interruptions, the results of our retrospective analysis reveals otherwise. We found that self-interruptions (i.e. voluntary task switchings) are more disruptive than external interruptions and have a negative effect on the performance of the interrupted tasks. Finally, we use the results of both studies to provide a set of comparative vulnerability and interaction patterns which can be used as a mean to guide decision-making and forecasting the consequences of task switching in software development teams.

[**Abdalkareem2017**] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. Why do developers use trivial packages? an empirical case study on NPM. In *Proc. European Software Engineering*

Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE). ACM, 8 2017, DOI 10.1145/3106237.3106267.

Abstract: Code reuse is traditionally seen as good practice. Recent trends have pushed the concept of code reuse to an extreme, by using packages that implement simple and trivial tasks, which we call 'trivial packages'. A recent incident where a trivial package led to the breakdown of some of the most popular web applications such as Facebook and Netflix made it imperative to question the growing use of trivial packages. Therefore, in this paper, we mine more than 230,000 npm packages and 38,000 JavaScript applications in order to study the prevalence of trivial packages. We found that trivial packages are common and are increasing in popularity, making up 16.8% of the studied npm packages. We performed a survey with 88 Node.js developers who use trivial packages to understand the reasons and drawbacks of their use. Our survey revealed that trivial packages are used because they are perceived to be well implemented and tested pieces of code. However, developers are concerned about maintaining and the risks of breakages due to the extra dependencies trivial packages introduce. To objectively verify the survey results, we empirically validate the most cited reason and drawback and find that, contrary to developers' beliefs, only 45.2% of trivial packages even have tests. However, trivial packages appear to be 'deployment tested' and to have similar test, usage and community interest as non-trivial packages. On the other hand, we found that 11.5% of the studied trivial packages have more than 20 dependencies. Hence, developers should be careful about which trivial packages they decide to use.

[**AbuHassan2020**] Amjad AbuHassan, Mohammad Alshayeb, and Lahouari Ghouti. Software smell detection techniques: A systematic literature review. *Journal of Software: Evolution and Process*, 33(3), 10 2020, DOI 10.1002/smr.2320.

Abstract: Software smells indicate design or code issues that might degrade the evolution and maintenance of software systems. Detecting and identifying these issues are challenging tasks. This paper explores, identifies, and analyzes the existing software smell detection techniques at design and code levels. We carried out a systematic literature review (SLR) to identify and collect 145 primary studies related to smell detection in software design and code. Based on these studies, we address several questions related to the analysis of the existing smell detection techniques in terms of abstraction level (design or code), targeted smells, used metrics, implementation, and validation. Our analysis identified several detection techniques categories. We observed that 57% of the studies did not use any performance measures, 41% of them omitted details on the targeted programing language, and the detection techniques were not validated in 14% of these studies. With respect to the abstraction level, only 18% of the studies addressed bad smell detection at the design level. This low coverage urges for more focus on bad smell detection at the design level to handle them at early stages. Finally, our SLR brings to the attention of the research community several opportunities for

future research.

[**Aghajani2019**] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Marquez, Mario Linares-Vasquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. Software documentation issues unveiled. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2019, DOI `10.1109/icse.2019.00122`.
Abstract: (Good) Software documentation provides developers and users with a description of what a software system does, how it operates, and how it should be used. For example, technical documentation (e.g., an API reference guide) aids developers during evolution/maintenance activities, while a user manual explains how users are to interact with a system. Despite its intrinsic value, the creation and the maintenance of documentation is often neglected, negatively impacting its quality and usefulness, ultimately leading to a generally unfavourable take on documentation. Previous studies investigating documentation issues have been based on surveying developers, which naturally leads to a somewhat biased view of problems affecting documentation. We present a large scale empirical study, where we mined, analyzed, and categorized 878 documentation-related artifacts stemming from four different sources, namely mailing lists, Stack Overflow discussions, issue repositories, and pull requests. The result is a detailed taxonomy of documentation issues from which we infer a series of actionable proposals both for researchers and practitioners.

[**Ajami2018**] Shulamyt Ajami, Yonatan Woodbridge, and Dror G. Feitelson. Syntax, predicates, idioms—what really affects code complexity? *Empirical Software Engineering*, 24(1):287–328, 6 2018, DOI `10.1007/s10664-018-9628-3`.
Abstract: Program comprehension concerns the ability to understand code written by others. But not all code is the same. We use an experimental platform fashioned as an online game-like environment to measure how quickly and accurately 220 professional programmers can interpret code snippets with similar functionality but different structures; snippets that take longer to understand or produce more errors are considered harder. The results indicate, inter alia, that for loops are significantly harder than ifs, that some but not all negations make a predicate harder, and that loops counting down are slightly harder than loops counting up. This demonstrates how the effect of syntactic structures, different ways to express predicates, and the use of known idioms can be measured empirically, and that syntactic structures are not necessarily the most important factor. We also found that the metrics of time to understanding and errors made are not necessarily equivalent. Thus loops counting down took slightly longer, but loops with unusual bounds caused many more errors. By amassing many more empirical results like these it may be possible to derive better code complexity metrics than we have today, and also to better appreciate their limitations.

[**Akerblom2016**] Beatrice Åkerblom and Tobias Wrigstad. Measuring polymorphism in Python programs. *ACM SIGPLAN Notices*, 51(2):114–128, 5 2016, DOI `10.1145/2936313.2816717`.

Abstract: Following the increased popularity of dynamic languages and their increased use in critical software, there have been many proposals to retrofit static type system to these languages to improve possibilities to catch bugs and improve performance. A key question for any type system is whether the types should be structural, for more expressiveness, or nominal, to carry more meaning for the programmer. For retrofitted type systems, it seems the current trend is using structural types. This paper attempts to answer the question to what extent this extra expressiveness is needed, and how the possible polymorphism in dynamic code is used in practise. We study polymorphism in 36 real-world open source Python programs and approximate to what extent nominal and structural types could be used to type these programs. The study is based on collecting traces from multiple runs of the programs and analysing the polymorphic degrees of targets at more than 7 million call-sites. Our results show that while polymorphism is used in all programs, the programs are to a great extent monomorphic. The polymorphism found is evenly distributed across libraries and program-specific code and occur both during program start-up and normal execution. Most programs contain a few 'megamorphic' call-sites where receiver types vary widely. The non-monomorphic parts of the programs can to some extent be typed with nominal or structural types, but none of the approaches can type entire programs.

[**AlSubaihin2021**] Afnan A. Al-Subaihin, Federica Sarro, Sue Black, Licia Capra, and Mark Harman. App store effects on software engineering practices. *IEEE Transactions on Software Engineering*, 47(2):300–319, 2 2021, DOI `10.1109/tse.2019.2891715`.

Abstract: In this paper, we study the app store as a phenomenon from the developers' perspective to investigate the extent to which app stores affect software engineering tasks. Through developer interviews and questionnaires, we uncover findings that highlight and quantify the effects of three high-level app store themes: bridging the gap between developers and users, increasing market transparency and affecting mobile release management. Our findings have implications for testing, requirements engineering and mining software repositories research fields. These findings can help guide future research in supporting mobile app developers through a deeper understanding of the app store-developer interaction.

[**AlencarDaCosta2017**] Daniel Alencar da Costa, Shane McIntosh, Christoph Treude, Uirá Kulesza, and Ahmed E. Hassan. The impact of rapid release cycles on the integration delay of fixed issues. *Empirical Software Engineering*, 23(2):835–904, 11 2017, DOI `10.1007/s10664-017-9548-7`.

Abstract: The release frequency of software projects has increased in recent years. Adopters of so-called rapid releases—short release cycles, often on the order of weeks, days, or even hours—claim that they can deliver fixed

issues (i.e., implemented bug fixes and new features) to users more quickly. However, there is little empirical evidence to support these claims. In fact, our prior work shows that code integration phases may introduce delays for rapidly releasing projects—98% of the fixed issues in the rapidly releasing Firefox project had their integration delayed by at least one release. To better understand the impact that rapid release cycles have on the integration delay of fixed issues, we perform a comparative study of traditional and rapid release cycles. Our comparative study has two parts: (i) a quantitative empirical analysis of 72,114 issue reports from the Firefox project, and a (ii) qualitative study involving 37 participants, who are contributors of the Firefox, Eclipse, and ArgoUML projects. Our study is divided into quantitative and qualitative analyses. Quantitative analyses reveal that, surprisingly, fixed issues take a median of 54% (57 days) longer to be integrated in rapid Firefox releases than the traditional ones. To investigate the factors that are related to integration delay in traditional and rapid release cycles, we train regression models that model whether a fixed issue will have its integration delayed or not. Our explanatory models achieve good discrimination (ROC areas of 0.80–0.84) and calibration scores (Brier scores of 0.05–0.16) for rapid and traditional releases. Our explanatory models indicate that (i) traditional releases prioritize the integration of backlog issues, while (ii) rapid releases prioritize issues that were fixed in the current release cycle. Complementary qualitative analyses reveal that participants' perception about integration delay is tightly related to activities that involve decision making, risk management, and team collaboration. Moreover, the allure of shipping fixed issues faster is a main motivator for adopting rapid release cycles among participants (although this motivation is not supported by our quantitative analysis). Furthermore, to explain why traditional releases deliver fixed issues more quickly, our participants point out the rush for integration in traditional releases and the increased time that is invested on polishing issues in rapid releases. Our results suggest that rapid release cycles may not be a silver bullet for the rapid delivery of new content to users. Instead, our results suggest that the benefits of rapid releases are increased software stability and user feedback.

[**Ali2020**] Rao Hamza Ali, Chelsea Parlett-Pelleriti, and Erik Linstead. Cheating death: a statistical survival analysis of publicly available Python projects. In *Proc. International Conference on Mining Software Repositories (MSR)*. ACM, 6 2020, DOI `10.1145/3379597.3387511`.
Abstract: We apply survival analysis methods to a dataset of publicly-available software projects in order to examine the attributes that might lead to their inactivity over time. We ran a Kaplan-Meier analysis and fit a Cox Proportional-Hazards model to a subset of Software Heritage Graph Dataset, consisting of 3052 popular Python projects hosted on GitLab/GitHub, Debian, and PyPI, over a period of 165 months. We show that projects with repositories on multiple hosting services, a timeline of publishing major releases, and a good network of developers, remain healthy over time and

should be worthy of the effort put in by developers and contributors.

[**Alkhabaz2021**] Ridha Alkhabaz, Seth Poulsen, Mei Chen, and Abdussalam Alawini. Insights from student solutions to MongoDB homework problems. In *Proc. Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, 6 2021, DOI `10.1145/3430665.3456308`.
Abstract: We analyze submissions for homework assignments of 527 students in an upper-level database course offered at the University of Illinois at Urbana-Champaign. The ability to query databases is becoming a crucial skill for technology professionals and academics. Although we observe a large demand for teaching database skills, there is little research on database education. Also, despite the industry's continued demand for NoSQL databases, we have virtually no research on the matter of how students learn NoSQL databases, such as MongoDB. In this paper, we offer an in-depth analysis of errors committed by students working on MongoDB homework assignments over the course of two semesters. We show that as students use more advanced MongoDB operators, they make more Reference errors. Additionally, when students face a new functionality of MongoDB operators, such as `$group` operator, they usually take time to understand it but do not make the same errors again in later problems. Finally, our analysis suggests that students struggle with advanced concepts for a comparable amount of time. Our results suggest that instructors should allocate more time and effort for the discussed topics in our paper.

[**Almeida2017**] Daniel A. Almeida, Gail C. Murphy, Greg Wilson, and Mike Hoye. do software developers understand open source licenses? In *Proc. International Conference on Program Comprehension (ICPC)*. IEEE, 5 2017, DOI `10.1109/icpc.2017.7`.
Abstract: Software provided under open source licenses is widely used, from forming high-profile stand-alone applications (e.g., Mozilla Firefox) to being embedded in commercial offerings (e.g., network routers). Despite the high frequency of use of open source licenses, there has been little work about whether software developers understand the open source licenses they use. To our knowledge, only one survey has been conducted, which focused on which licenses developers choose and when they encounter problems with licensing open source software. To help fill the gap of whether or not developers understand the open source licenses they use, we conducted a survey that posed development scenarios involving three popular open source licenses (GNU GPL 3.0, GNU LGPL 3.0 and MPL 2.0) both alone and in combination. The 375 respondents to the survey, who were largely developers, gave answers consistent with those of a legal expert's opinion in 62% of 42 cases. Although developers clearly understood cases involving one license, they struggled when multiple licenses were involved. An analysis of the quantitative and qualitative results of the study indicate a need for tool support to help guide developers in understanding this critical information attached to software components.

[**Altadmri2015**] Amjad Altadmri and Neil C.C. Brown. 37 million compilations: investigating novice programming mistakes in large-scale student data. In *Proc. Technical Symposium on Computer Science Education (SIGCSE)*. ACM, 2 2015, DOI `10.1145/2676723.2677258`.

Abstract: Previous investigations of student errors have typically focused on samples of hundreds of students at individual institutions. This work uses a year's worth of compilation events from over 250,000 students all over the world, taken from the large Blackbox data set. We analyze the frequency, time-to-fix, and spread of errors among users, showing how these factors inter-relate, in addition to their development over the course of the year. These results can inform the design of courses, textbooks and also tools to target the most frequent (or hardest to fix) errors.

[**Ameller2012**] David Ameller, Claudia Ayala, Jordi Cabot, and Xavier Franch. How do software architects consider non-functional requirements: an exploratory study. In *Proc. International Requirements Engineering Conference (RE)*. IEEE, 9 2012, DOI `10.1109/re.2012.6345838`.

Abstract: Dealing with non-functional requirements (NFRs) has posed a challenge onto software engineers for many years. Over the years, many methods and techniques have been proposed to improve their elicitation, documentation, and validation. Knowing more about the state of the practice on these topics may benefit both practitioners' and researchers' daily work. A few empirical studies have been conducted in the past, but none under the perspective of software architects, in spite of the great influence that NFRs have on daily architects' practices. This paper presents some of the findings of an empirical study based on 13 interviews with software architects. It addresses questions such as: who decides the NFRs, what types of NFRs matter to architects, how are NFRs documented, and how are NFRs validated. The results are contextualized with existing previous work.

[**Ames2018**] Morgan G. Ames. Hackers, computers, and cooperation: A critical history of logo and constructionist learning. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):1–19, 11 2018, DOI `10.1145/3274287`.

Abstract: This paper examines the history of the learning theory "constructionism" and its most well-known implementation, Logo, to examine beliefs involving both "C's" in CSCW: computers and cooperation. Tracing the tumultuous history of one of the first examples of computer-supported cooperative learning (CSCL) allows us to question some present-day assumptions regarding the universal appeal of learning to program computers that undergirds popular CSCL initiatives today, including the Scratch programming environment and the "FabLab" makerspace movement. Furthermore, teasing out the individualistic and anti-authority threads in this project and its links to present day narratives of technology development exposes the deeply atomized and even oppositional notions of collaboration in these projects and others under the auspices of CSCW today that draw on early notions of 'hacker culture.' These notions tend to favor a limited view of

work, learning, and practice-an invisible constraint that continues to inform how we build and evaluate CSCW technologies.

[**Anda2009**] B.C.D. Anda, D.I.K. Sjøberg, and Audris Mockus. Variability and reproducibility in software engineering: a study of four companies that developed the same system. *IEEE Transactions on Software Engineering*, 35(3):407–429, 5 2009, DOI `10.1109/tse.2008.89`.
Abstract: The scientific study of a phenomenon requires it to be reproducible. Mature engineering industries are recognized by projects and products that are, to some extent, reproducible. Yet, reproducibility in software engineering (SE) has not been investigated thoroughly, despite the fact that lack of reproducibility has both practical and scientific consequences. We report a longitudinal multiple-case study of variations and reproducibility in software development, from bidding to deployment, on the basis of the same requirement specification. In a call for tender to 81 companies, 35 responded. Four of them developed the system independently. The firm price, planned schedule, and planned development process, had, respectively, "low," "low," and "medium" reproducibilities. The contractor's costs, actual lead time, and schedule overrun of the projects had, respectively, "medium," "high," and "low" reproducibilities. The quality dimensions of the delivered products, reliability, usability, and maintainability had, respectively, "low," "high," and "low" reproducibilities. Moreover, variability for predictable reasons is also included in the notion of reproducibility. We found that the observed outcome of the four development projects matched our expectations, which were formulated partially on the basis of SE folklore. Nevertheless, achieving more reproducibility in SE remains a great challenge for SE research, education, and industry.

[**Apel2011**] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. Semistructured merge: rethinking merge in revision control systems. In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM Press, 2011, DOI `10.1145/2025113.2025141`.
Abstract: An ongoing problem in revision control systems is how to resolve conflicts in a merge of independently developed revisions. Unstructured revision control systems are purely text-based and solve conflicts based on textual similarity. Structured revision control systems are tailored to specific languages and use language-specific knowledge for conflict resolution. We propose semistructured revision control systems that inherit the strengths of both: the generality of unstructured systems and the expressiveness of structured systems. The idea is to provide structural information of the underlying software artifacts — declaratively, in the form of annotated grammars. This way, a wide variety of languages can be supported and the information provided can assist in the automatic resolution of two classes of conflicts: ordering conflicts and semantic conflicts. The former can be resolved independently of the language and the latter using specific conflict handlers. We

have been developing a tool that supports semistructured merge and conducted an empirical study on 24 software projects developed in Java, C#, and Python comprising 180 merge scenarios. We found that semistructured merge reduces the number of conflicts in 60% of the sample merge scenarios by, on average, 34%, compared to unstructured merge. We found also that renaming is challenging in that it can increase the number of conflicts during semistructured merge, and that a combination of unstructured and semistructured merge is a pragmatic way to go.

[**Aranda2009**] Jorge Aranda and Gina Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 2009, DOI `10.1109/icse.2009.5070530`.
Abstract: Every bug has a story behind it. The people that discover and resolve it need to coordinate, to get information from documents, tools, or other people, and to navigate through issues of accountability, ownership, and organizational structure. This paper reports on a field study of coordination activities around bug fixing that used a combination of case study research and a survey of software professionals. Results show that the histories of even simple bugs are strongly dependent on social, organizational, and technical knowledge that cannot be solely extracted through automation of electronic repositories, and that such automation provides incomplete and often erroneous accounts of coordination. The paper uses rich bug histories and survey results to identify common bug fixing coordination patterns and to provide implications for tool designers and researchers of coordination in software development.

[**Aurora2019**] Valerie Aurora and Mary Gardiner. *How to Respond to Code of Conduct Reports*. Frame Shift Consulting LLC, version 1.1 edition, 2019.
Abstract: A detailed, experience-based guide to handling what is often the most difficult situation in any project.

[**Bafatakis2019**] Nikolaos Bafatakis, Niels Boecker, Wenjie Boon, Martin Cabello Salazar, Jens Krinke, Gazi Oznacar, and Robert White. Python coding style compliance on stack overflow. In *Proc. International Conference on Mining Software Repositories (MSR)*. IEEE, 5 2019, DOI `10.1109/msr.2019.00042`.
Abstract: Software developers all over the world use Stack Overflow (SO) to interact and exchange code snippets. Research also uses SO to harvest code snippets for use with recommendation systems. However, previous work has shown that code on SO may have quality issues, such as security or license problems. We analyse Python code on SO to determine its coding style compliance. From 1,962,535 code snippets tagged with 'python', we extracted 407,097 snippets of at least 6 statements of Python code. Surprisingly, 93.87% of the extracted snippets contain style violations, with an average of 0.7 violations per statement and a huge number of snippets with a considerably higher ratio. Researchers and developers should, therefore,

be aware that code snippets on SO may not representative of good coding style. Furthermore, while user reputation seems to be unrelated to coding style compliance, for posts with vote scores in the range between -10 and 20, we found a strong correlation ($r = -0.87$, $p < 10^-7$) between the vote score a post received and the average number of violations per statement for snippets in such posts.

[**Balachandran2013**] Vipin Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2013, DOI `10.1109/icse.2013.6606642`.

Abstract: Peer code review is a cost-effective software defect detection technique. Tool assisted code review is a form of peer code review, which can improve both quality and quantity of reviews. However, there is a significant amount of human effort involved even in tool based code reviews. Using static analysis tools, it is possible to reduce the human effort by automating the checks for coding standard violations and common defect patterns. Towards this goal, we propose a tool called Review Bot for the integration of automatic static analysis with the code review process. Review Bot uses output of multiple static analysis tools to publish reviews automatically. Through a user study, we show that integrating static analysis tools with code review process can improve the quality of code review. The developer feedback for a subset of comments from automatic reviews shows that the developers agree to fix 93% of all the automatically generated comments. There is only 14.71% of all the accepted comments which need improvements in terms of priority, comment message, etc. Another problem with tool assisted code review is the assignment of appropriate reviewers. Review Bot solves this problem by generating reviewer recommendations based on change history of source code lines. Our experimental results show that the recommendation accuracy is in the range of 60%-92%, which is significantly better than a comparable method based on file change history.

[**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.

[**Baltes2020**] Sebastian Baltes, George Park, and Alexander Serebrenik. Is 40 the new 60? how popular media portrays the employability of older software developers. *IEEE Software*, 37(6):26–31, 11 2020, DOI `10.1109/ms.2020.3014178`.
Abstract: We studied the public discourse around age and software development, focusing on the United States. This work was designed to build awareness among decision makers in software projects to help them anticipate and mitigate challenges that their older employees may face.

[**Bao2021**] Lingfeng Bao, Xin Xia, David Lo, and Gail C. Murphy. A large scale study of long-time contributor prediction for GitHub projects. *IEEE Transactions on Software Engineering*, 47(6):1277–1298, 6 2021, DOI `10.1109/tse.2019.2918536`.
Abstract: The continuous contributions made by long time contributors (LTCs) are a key factor enabling open source software (OSS) projects to be successful and survival. We study Github as it has a large number of OSS projects and millions of contributors, which enables the study of the transition from newcomers to LTCs. In this paper, we investigate whether we can effectively predict newcomers in OSS projects to be LTCs based on their activity data that is collected from Github. We collect Github data from GHTorrent, a mirror of Github data. We select the most popular 917 projects, which contain 75,046 contributors. We determine a developer as a LTC of a project if the time interval between his/her first and last commit in the project is larger than a certain time T. In our experiment, we use three different settings on the time interval: 1, 2, and 3 years. There are 9,238, 3,968, and 1,577 contributors who become LTCs of a project in three settings of time interval, respectively. To build a prediction model, we extract many features from the activities of developers on Github, which group into five dimensions: developer profile, repository profile, developer monthly activity, repository monthly activity, and collaboration network. We apply several classifiers including naive Bayes, SVM, decision tree, kNN and

random forest. We find that random forest classifier achieves the best performance with AUCs of more than 0.75 in all three settings of time interval for LTCs. We also investigate the most important features that differentiate newcomers who become LTCs from newcomers who stay in the projects for a short time. We find that the number of followers is the most important feature in all three settings of the time interval studied. We also find that the programming language and the average number of commits contributed by other developers when a newcomer joins a project also belong to the top 10 most important features in all three settings of time interval for LTCs. Finally, we provide several implications for action based on our analysis results to help OSS projects retain newcomers.

[**Barbosa2014**] Eiji Adachi Barbosa, Alessandro Garcia, and Simone Diniz Junqueira Barbosa. Categorizing faults in exception handling: a study of open source projects. In *Proc. Brazilian Symposium on Software Engineering (BSSE)*. IEEE, 9 2014, DOI `10.1109/sbes.2014.19`.
Abstract: Even though exception handling mechanisms have been proposed as a means to improve software robustness, empirical evidence suggests that exception handling code is still poorly implemented in industrial systems. Moreover, it is often claimed that the poor quality of exception handling code can be a source of faults in a software system. However, there is still a gap in the literature in terms of better understanding exceptional faults, i.e., faults whose causes regard to exception handling. In particular, there is still little empirical knowledge about what are the specific causes of exceptional faults in software systems. In this paper we start to fill this gap by presenting a categorization of the causes of exceptional faults observed in two mainstream open source projects. We observed ten different categories of exceptional faults, most of which were never reported before in the literature. Our results pinpoint that current verification and validation mechanisms for exception handling code are still not properly addressing these categories of exceptional faults.

[**Barik2017**] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. Do developers read compiler error messages? In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2017, DOI `10.1109/icse.2017.59`.
Abstract: In integrated development environments, developers receive compiler error messages through a variety of textual and visual mechanisms, such as popups and wavy red underlines. Although error messages are the primary means of communicating defects to developers, researchers have a limited understanding on how developers actually use these messages to resolve defects. To understand how developers use error messages, we conducted an eye tracking study with 56 participants from undergraduate and graduate software engineering courses at our university. The participants attempted to resolve common, yet problematic defects in a Java code base within the Eclipse development environment. We found that: 1) participants read error messages and the difficulty of reading these messages is comparable to the

difficulty of reading source code, 2) difficulty reading error messages significantly predicts participants' task performance, and 3) participants allocate a substantial portion of their total task to reading error messages (13%-25%). The results of our study offer empirical justification for the need to improve compiler error messages for developers.

[**Barik2018**] Titus Barik, Denae Ford, Emerson Murphy-Hill, and Chris Parnin. How should compilers explain problems to developers? In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 10 2018, DOI `10.1145/3236024.3236040`.
Abstract: Compilers primarily give feedback about problems to developers through the use of error messages. Unfortunately, developers routinely find these messages to be confusing and unhelpful. In this paper, we postulate that because error messages present poor explanations, theories of explanation—such as Toulmin's model of argument—can be applied to improve their quality. To understand how compilers should present explanations to developers, we conducted a comparative evaluation with 68 professional software developers and an empirical study of compiler error messages found in Stack Overflow questions across seven different programming languages. Our findings suggest that, given a pair of error messages, developers significantly prefer the error message that employs proper argument structure over a deficient argument structure when neither offers a resolution—but will accept a deficient argument structure if it provides a resolution to the problem. Human-authored explanations on Stack Overflow converge to one of the three argument structures: those that provide a resolution to the error, simple arguments, and extended arguments that provide additional evidence for the problem. Finally, we contribute three practical design principles to inform the design and evaluation of compiler error messages.

[**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 (Teamwide 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.

[**Barnett2011**] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: the Spec# experience. *Communications of the ACM*, 54(6):81–91, 6 2011, DOI `10.1145/1953122.1953145`.
Abstract: Can a programming language really help programmers write better programs?

[**Barr2012**] Earl T. Barr, Christian Bird, Peter C. Rigby, Abram Hindle, Daniel M. German, and Premkumar Devanbu. Cohesive and isolated development with branches. In *Proc. International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 316–331. Springer Berlin Heidelberg, 2012, DOI `10.1007/978-3-642-28872-2_22`.
Abstract: The adoption of distributed version control (DVC), such as Git and Mercurial, in open-source software (OSS) projects has been explosive. Why is this and how are projects using DVC? This new generation of version control supports two important new features: distributed repositories and histories that preserve branches and merges. Through interviews with lead developers in OSS projects and a quantitative analysis of mined data from the histories of sixty project, we find that the vast majority of the projects now using DVC continue to use a centralized model of code sharing, while using branching much more extensively than before their transition to DVC. We then examine the Linux history in depth in an effort to understand and evaluate how branches are used and what benefits they provide. We find that they enable natural collaborative processes: DVC branching allows developers to collaborate on tasks in highly cohesive branches, while enjoying reduced interference from developers working on other tasks, even if those tasks are strongly coupled to theirs.

[**Barzilay2011**] Ohad Barzilay. Example embedding. In *Proc. Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (ONWARD)*. ACM Press, 2011, DOI `10.1145/2089131.2089135`.
Abstract: Using code examples in professional software development is like teenage sex. Those who say they do it all the time are probably lying. Although it is natural, those who do it feel guilty. Finally, once they start doing it, they are often not too concerned with safety, they discover that it is going to take a while to get really good at it, and they realize they will have to come up with a bunch of new ways of doing it before they really figure it all out.

[**Beck2011**] Fabian Beck and Stephan Diehl. On the congruence of modularity and code coupling. In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering*

(ESEC/FSE). ACM Press, 2011, DOI 10.1145/2025113.2025162.

Abstract: Software systems are modularized to make their inherent complexity manageable. While there exists a set of well-known principles that may guide software engineers to design the modules of a software system, we do not know which principles are followed in practice. In a study based on 16 open source projects, we look at different kinds of coupling concepts between source code entities, including structural dependencies, fan-out similarity, evolutionary coupling, code ownership, code clones, and semantic similarity. The congruence between these coupling concepts and the modularization of the system hints at the modularity principles used in practice. Furthermore, the results provide insights on how to support developers to modularize software systems.

[**Becker2019**] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. Compiler error messages considered unhelpful. In *Proc. Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, 12 2019, DOI 10.1145/3344429.3372508.

Abstract: Diagnostic messages generated by compilers and interpreters such as syntax error messages have been researched for over half of a century. Unfortunately, these messages which include error, warning, and run-time messages, present substantial difficulty and could be more effective, particularly for novices. Recent years have seen an increased number of papers in the area including studies on the effectiveness of these messages, improving or enhancing them, and their usefulness as a part of programming process data that can be used to predict student performance, track student progress, and tailor learning plans. Despite this increased interest, the long history of literature is quite scattered and has not been brought together in any digestible form. In order to help the computing education community (and related communities) to further advance work on programming error messages, we present a comprehensive, historical and state-of-the-art report on research in the area. In addition, we synthesise and present the existing evidence for these messages including the difficulties they present and their effectiveness. We finally present a set of guidelines, curated from the literature, classified on the type of evidence supporting each one (historical, anecdotal, and empirical). This work can serve as a starting point for those who wish to conduct research on compiler error messages, runtime errors, and warnings. We also make the bibtex file of our 300+ reference corpus publicly available. Collectively this report and the bibliography will be useful to those who wish to design better messages or those that aim to measure their effectiveness, more effectively.

[**Begel2014**] Andrew Begel and Thomas Zimmermann. Analyze this! 145 questions for data scientists in software engineering. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 5 2014, DOI 10.1145/2568225.2568233.

Abstract: In this paper, we present the results from two surveys related to data science applied to software engineering. The first survey solicited questions that software engineers would like data scientists to investigate about software, about software processes and practices, and about software engineers. Our analyses resulted in a list of 145 questions grouped into 12 categories. The second survey asked a different pool of software engineers to rate these 145 questions and identify the most important ones to work on first. Respondents favored questions that focus on how customers typically use their applications. We also saw opposition to questions that assess the performance of individual employees or compare them with one another. Our categorization and catalog of 145 questions can help researchers, practitioners, and educators to more easily focus their efforts on topics that are important to the software industry.

[**Behroozi2019**] Mahnaz Behroozi, Chris Parnin, and Titus Barik. Hiring is broken: What do developers say about technical interviews? In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 10 2019, DOI `10.1109/vlhcc.2019.8818836`.
Abstract: Technical interviews—a problem-solving form of interview in which candidates write code—are commonplace in the software industry, and are used by several well-known companies including Facebook, Google, and Microsoft. These interviews are intended to objectively assess candidates and determine fit within the company. But what do developers say about them?To understand developer perceptions about technical interviews, we conducted a qualitative study using the online social news website, Hacker News—a venue for software practitioners. Hacker News posters report several concerns and negative perceptions about interviews, including their lack of real-world relevance, bias towards younger developers, and demanding time commitment. Posters report that these interviews cause unnecessary anxiety and frustration, requiring them to learn arbitrary, implicit, and obscure norms. The findings from our study inform inclusive hiring guidelines for technical interviews, such as collaborative problem-solving sessions.

[**Behroozi2020**] Mahnaz Behroozi, Shivani Shirolkar, Titus Barik, and Chris Parnin. Debugging hiring: What went right and what went wrong in the technical interview process. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 2020, DOI `10.1145/3377815.3381372`.
Abstract: The typical hiring pipeline for software engineering occurs over several stages—from phone screening and technical on-site interviews, to offer and negotiation. When these hiring pipelines are "leaky," otherwise qualified candidates are lost at some stage of the pipeline. These leaky pipelines impact companies in several ways, including hindering a company's ability to recruit competitive candidates and build diverse software teams.To understand where candidates become disengaged in the hiring pipeline—and what companies can do to prevent it—we conducted a qualitative study on over 10,000 reviews on 19 companies from Glassdoor, a website where candidates can leave reviews about their hiring process experiences. We identified several

poor practices which prematurely sabotage the hiring process—for example, not adequately communicating hiring criteria, conducting interviews with inexperienced interviewers, and ghosting candidates. Our findings provide a set of guidelines to help companies improve their hiring pipeline practices— such as being deliberate about phrasing and language during initial contact with the candidate, providing candidates with constructive feedback after their interviews, and bringing salary transparency and long-term career discussions into offers and negotiations. Operationalizing these guidelines helps make the hiring pipeline more transparent, fair, and inclusive.

[**Beller2015**] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their IDEs. In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 8 2015, DOI `10.1145/2786805.2786843`.

Abstract: The research community in Software Engineering and Software Testing in particular builds many of its contributions on a set of mutually shared expectations. Despite the fact that they form the basis of many publications as well as open-source and commercial testing applications, these common expectations and beliefs are rarely ever questioned. For example, Frederic Brooks' statement that testing takes half of the development time seems to have manifested itself within the community since he first made it in the "Mythical Man Month" in 1975. With this paper, we report on the surprising results of a large-scale field study with 416 software engineers whose development activity we closely monitored over the course of five months, resulting in over 13 years of recorded work time in their integrated development environments (IDEs). Our findings question several commonly shared assumptions and beliefs about testing and might be contributing factors to the observed bug proneness of software in practice: the majority of developers in our study does not test; developers rarely run their tests in the IDE; Test-Driven Development (TDD) is not widely practiced; and, last but not least, software developers only spend a quarter of their work time engineering tests, whereas they think they test half of their time.

[**Beller2019**] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. Developer testing in the IDE: Patterns, beliefs, and behavior. *IEEE Transactions on Software Engineering*, 45(3):261–284, 3 2019, DOI `10.1109/tse.2017.2776152`.

Abstract: Software testing is one of the key activities to achieve software quality in practice. Despite its importance, however, we have a remarkable lack of knowledge on how developers test in real-world projects. In this paper, we report on a large-scale field study with 2,443 software engineers whose development activities we closely monitored over 2.5 years in four integrated development environments (IDEs). Our findings, which largely generalized across the studied IDEs and programming languages Java and C#, question several commonly shared assumptions and beliefs about developer testing: half of the developers in our study do not test; developers rarely run their

tests in the IDE; most programming sessions end without any test execution; only once they start testing, do they do it extensively; a quarter of test cases is responsible for three quarters of all test failures; 12 percent of tests show flaky behavior; Test-Driven Development (TDD) is not widely practiced; and software developers only spend a quarter of their time engineering tests, whereas they think they test half of their time. We summarize these practices of loosely guiding one's development efforts with the help of testing in an initial summary on Test-Guided Development (TGD), a behavior we argue to be closer to the development reality of most developers than TDD.

[**BenAri2011**] Mordechai Ben-Ari, Roman Bednarik, Ronit Ben-Bassat Levy, Gil Ebel, Andrés Moreno, Niko Myller, and Erkki Sutinen. A decade of research and development on program animation: the Jeliot experience. *Journal of Visual Languages & Computing*, 22(5):375–384, 10 2011, DOI `10.1016/j.jvlc.2011.04.004`.
Abstract: Jeliot is a program animation system for teaching and learning elementary programming that has been developed over the past decade, building on the Eliot animation system developed several years before. Extensive pedagogical research has been done on various aspects of the use of Jeliot including improvements in learning, effects on attention, and acceptance by teachers. This paper surveys this research and development, and summarizes the experience and the lessons learned.

[**Beniamini2017**] Gal Beniamini, Sarah Gingichashvili, Alon Klein Orbach, and Dror G. Feitelson. Meaningful identifier names: the case of single-letter variables. In *Proc. International Conference on Program Comprehension (ICPC)*. IEEE, 5 2017, DOI `10.1109/icpc.2017.18`.
Abstract: It is widely accepted that variable names in computer programs should be meaningful, and that this aids program comprehension. "Meaningful" is commonly interpreted as favoring long descriptive names. However, there is at least some use of short and even single-letter names: using i in loops is very common, and we show (by extracting variable names from 1000 popular GitHub projects in 5 languages) that some other letters are also widely used. In addition, controlled experiments with different versions of the same functions (specifically, different variable names) failed to show significant differences in ability to modify the code. Finally, an online survey showed that certain letters are strongly associated with certain types and meanings. This implies that a single letter can in fact convey meaning. The conclusion from all this is that single letter variables can indeed be used beneficially in certain cases, leading to more concise code.

[**Bettenburg2008**] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proc. International Symposium on Foundations of Software Engineering/International Symposium on the Foundations of Software Engineering (SIGSOFT/FSE)*. ACM Press, 2008, DOI `10.1145/1453101.1453146`.
Abstract: In software development, bug reports provide crucial informa-

tion to developers. However, these reports widely differ in their quality. We conducted a survey among developers and users of Apache, Eclipse, and Mozilla to find out what makes a good bug report. The analysis of the 466 responses revealed an information mismatch between what developers need and what users supply. Most developers consider steps to reproduce, stack traces, and test cases as helpful, which are, at the same time, most difficult to provide for users. Such insight is helpful for designing new bug tracking tools that guide users at collecting and providing more helpful information. Our Cuezilla prototype is such a tool and measures the quality of new bug reports; it also recommends which elements should be added to improve the quality. We trained Cuezilla on a sample of 289 bug reports, rated by developers as part of the survey. The participants of our survey also provided 175 comments on hurdles in reporting and resolving bugs. Based on these comments, we discuss several recommendations for better bug tracking systems, which should focus on engaging bug reporters, better tool support, and improved handling of bug duplicates.

[**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.

[**Bird2011**] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proc. International Symposium on Foundations of Software Engineering/International Symposium on the Foundations of Software Engineering (SIGSOFT/FSE)*. ACM Press, 2011, DOI `10.1145/2025113.2025119`.
Abstract: Ownership is a key aspect of large-scale software development.

We examine the relationship between different ownership measures and software failures in two large software projects: Windows Vista and Windows 7. We find that in all cases, measures of ownership such as the number of low-expertise developers, and the proportion of ownership for the top owner have a relationship with both pre-release faults and post-release failures. We also empirically identify reasons that low-expertise developers make changes to components and show that the removal of low-expertise contributions dramatically decreases the performance of contribution based defect prediction. Finally we provide recommendations for source code change policies and utilization of resources such as code inspections based on our results.

[**Blackwell2019**] Alan F. Blackwell, Marian Petre, and Luke Church. Fifty years of the psychology of programming. *International Journal of Human-Computer Studies*, 131:52–63, 11 2019, DOI `10.1016/j.ijhcs.2019.06.009`.
Abstract: Abstract This paper reflects on the evolution (past, present and future) of the 'psychology of programming' over the 50 year period of this anniversary issue. The International Journal of Human-Computer Studies (IJHCS) has been a key venue for much seminal work in this field, including its first foundations, and we review the changing research concerns seen in publications over these five decades. We relate this thematic evolution to research taking place over the same period within more specialist communities, especially the Psychology of Programming Interest Group (PPIG), the Empirical Studies of Programming series (ESP), and the ongoing community in Visual Languages and Human-Centric Computing (VL/HCC). Many other communities have interacted with psychology of programming, both influenced by research published within the specialist groups, and in turn influencing research priorities. We end with an overview of the core theories that have been developed over this period, as an introductory resource for new researchers, and also with the authors' own analysis of key priorities for future research.

[**Bluedorn1999**] Allen C. Bluedorn, Daniel B. Turban, and Mary Sue Love. The effects of stand-up and sit-down meeting formats on meeting outcomes. *Journal of Applied Psychology*, 84(2):277–285, 1999, DOI `10.1037/0021-9010.84.2.277`.
Abstract: The effects of meeting format (standing or sitting) on meeting length and the quality of group decision making were investigated by comparing meeting outcomes for 56 five-member groups that conducted meetings in a standing format with 55 five-member groups that conducted meetings in a seated format. Sit-down meetings were 34% longer than stand-up meetings, but they produced no better decisions than stand-up meetings. Significant differences were also obtained for satisfaction with the meeting and task information use during the meeting but not for synergy or commitment to the group's decision. The findings were generally congruent with meeting-management recommendations in the time-management literature, although

the lack of a significant difference for decision quality was contrary to theoretical expectations. This contrary finding may have been due to differences between the temporal context in which this study was conducted and those in which other time constraint research has been conducted, thereby revealing a potentially important contingency-temporal context.

[**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.

[**Bogomolov2021**] Egor Bogomolov, Vladimir Kovalenko, Yurii Rebryk, Alberto Bacchelli, and Timofey Bryksin. Authorship attribution of source code: a language-agnostic approach and applicability in software engineering. In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 8 2021, DOI `10.1145/3468264.3468606`.

Abstract: Authorship attribution (i.e., determining who is the author of a piece of source code) is an established research topic. State-of-the-art results for the authorship attribution problem look promising for the software engineering field, where they could be applied to detect plagiarized code and prevent legal issues. With this article, we first introduce a new language-agnostic approach to authorship attribution of source code. Then, we discuss limitations of existing synthetic datasets for authorship attribution, and propose a data collection approach that delivers datasets that better reflect aspects important for potential practical use in software engineering. Finally, we demonstrate that high accuracy of authorship attribution models on existing datasets drastically drops when they are evaluated on more realistic

data. We outline next steps for the design and evaluation of authorship attribution models that could bring the research efforts closer to practical use for software engineering.

[**Borle2017**] Neil C. Borle, Meysam Feghhi, Eleni Stroulia, Russell Greiner, and Abram Hindle. Analyzing the effects of test driven development in GitHub. *Empirical Software Engineering*, 23(4):1931–1958, 11 2017, DOI `10.1007/s10664-017-9576-3`.
Abstract: Testing is an integral part of the software development lifecycle, approached with varying degrees of rigor by different process models. Agile process models recommend Test Driven Development (TDD) as a key practice for reducing costs and improving code quality. The objective of this work is to perform a cost-benefit analysis of this practice. To that end, we have conducted a comparative analysis of GitHub repositories that adopts TDD to a lesser or greater extent, in order to determine how TDD affects software development productivity and software quality. We classified GitHub repositories archived in 2015 in terms of how rigorously they practiced TDD, thus creating a TDD spectrum. We then matched and compared various subsets of these repositories on this TDD spectrum with control sets of equal size. The control sets were samples from all GitHub repositories that matched certain characteristics, and that contained at least one test file. We compared how the TDD sets differed from the control sets on the following characteristics: number of test files, average commit velocity, number of bug-referencing commits, number of issues recorded, usage of continuous integration, number of pull requests, and distribution of commits per author. We found that Java TDD projects were relatively rare. In addition, there were very few significant differences in any of the metrics we used to compare TDD-like and non-TDD projects; therefore, our results do not reveal any observable benefits from using TDD.

[**Brown2018**] Neil C. C. Brown, Amjad Altadmri, Sue Sentance, and Michael Kölling. Blackbox, five years on: an evaluation of a large-scale programming data collection project. In *Proc. Conference on International Computing Education Research (ICER)*. ACM, 8 2018, DOI `10.1145/3230977.3230991`.
Abstract: The Blackbox project has been collecting programming activity data from users of BlueJ (a novice-targeted Java development environment) for nearly five years. The resulting dataset of more than two terabytes of data has been made available to interested researchers from the outset. In this paper, we assess the impact of the Blackbox project: we perform a mapping study to assess eighteen publications which have made use of the Blackbox data, and we report on the advantages and difficulties experienced by researchers working with this data, collected via a survey. We find that Blackbox has enabled pieces of research which otherwise would not have been possible, but there remain technical challenges in the analysis. Some of these—but not all—relate to the scale of the data. We provide suggestions for the future use of Blackbox, and reflections on the role of such data collection projects in programming research.

[**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.

[**Brun2011**] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *Proc. International Symposium on Foundations of Software Engineering/International Symposium on the Foundations of Software Engineering (SIGSOFT/FSE)*. ACM Press, 2011, DOI `10.1145/2025113.2025139`.

Abstract: Collaborative development can be hampered when conflicts arise because developers have inconsistent copies of a shared project. We present an approach to help developers identify and resolve conflicts early, before those conflicts become severe and before relevant changes fade away in the developers' memories. This paper presents three results. First, a study of open-source systems establishes that conflicts are frequent, persistent, and appear not only as overlapping textual edits but also as subsequent build and test failures. The study spans nine open-source systems totaling 3.4 million lines of code; our conflict data is derived from 550,000 development versions of the systems. Second, using previously-unexploited information, we precisely diagnose important classes of conflicts using the novel technique of speculative analysis over version control operations. Third, we describe the design of Crystal, a publicly-available tool that uses speculative analysis to make concrete advice unobtrusively available to developers, helping them identify, manage, and prevent conflicts.

[**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.

[**Cabral2007**] Bruno Cabral and Paulo Marques. Exception handling: A field study in java and .NET. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, pages 151–175. Springer Berlin Heidelberg, 2007, DOI `10.1007/978-3-540-73589-2_8`.

Abstract: Most modern programming languages rely on exceptions for dealing with abnormal situations. Although exception handling was a significant improvement over other mechanisms like checking return codes, it is far from perfect. In fact, it can be argued that this mechanism is seriously limited, if not, flawed. This paper aims to contribute to the discussion by providing quantitative measures on how programmers are currently using exception handling. We examined 32 different applications, both for Java and .NET. The major conclusion for this work is that exceptions are not being correctly used as an error recovery mechanism. Exception handlers are not specialized enough for allowing recovery and, typically, programmers just do one of the following actions: logging, user notification and application termination. To our knowledge, this is the most comprehensive study done on exception handling to date, providing a quantitative measure useful for guiding the development of new error handling mechanisms.

[**Campos2017**] Eduardo Cunha Campos and Marcelo de Almeida Maia. Common bug-fix patterns: a large-scale observational study. In *Proc. Inter-*

national Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, 11 2017, DOI `10.1109/esem.2017.55`.

Abstract: [Background]: There are more bugs in real-world programs than human programmers can realistically address. Several approaches have been proposed to aid debugging. A recent research direction that has been increasingly gaining interest to address the reduction of costs associated with defect repair is automatic program repair. Recent work has shown that some kind of bugs are more suitable for automatic repair techniques. [Aim]: The detection and characterization of common bug-fix patterns in software repositories play an important role in advancing the field of automatic program repair. In this paper, we aim to characterize the occurrence of known bug-fix patterns in Java repositories at an unprecedented large scale. [Method]: The study was conducted for Java GitHub projects organized in two distinct data sets: the first one (i.e., Boa data set) contains more than 4 million bug-fix commits from 101,471 projects and the second one (i.e., Defects4J data set) contains 369 real bug fixes from five open-source projects. We used a domain-specific programming language called Boa in the first data set and conducted a manual analysis on the second data set in order to confront the results. [Results]: We characterized the prevalence of the five most common bug-fix patterns (identified in the work of Pan et al.) in those bug fixes. The combined results showed direct evidence that developers often forget to add IF preconditions in the code. Moreover, 76% of bug-fix commits associated with the IF-APC bug-fix pattern are isolated from the other four bug-fix patterns analyzed. [Conclusion]: Targeting on bugs that miss preconditions is a feasible alternative in automatic repair techniques that would produce a relevant payback.

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

Abstract: As social as software engineers are, there is a known and established gender imbalance in our community structures, regardless of their open-or closed-source nature. To shed light on the actual benefits of achieving such balance, this empirical study looks into the relations between such balance and the occurrence of community smells, that is, sub-optimal circumstances and patterns across the software organizational structure. Examples of community smells are Organizational Silo effects (overly disconnected subgroups) or Lone Wolves (defiant community members). Results indicate that the presence of women generally reduces the amount of community smells. We conclude that women are instrumental to reducing community smells in software development teams.

[**Chatterjee2021**] Preetha Chatterjee, Kostadin Damevski, Nicholas A. Kraft, and Lori Pollock. Automatically identifying the quality of developer chats

for post hoc use. *ACM Transactions on Software Engineering and Methodology*, 30(4):1–28, 7 2021, DOI `10.1145/3450503`.

Abstract: Software engineers are crowdsourcing answers to their everyday challenges on Q&A forums (e.g., Stack Overflow) and more recently in public chat communities such as Slack, IRC, and Gitter. Many software-related chat conversations contain valuable expert knowledge that is useful for both mining to improve programming support tools and for readers who did not participate in the original chat conversations. However, most chat platforms and communities do not contain built-in quality indicators (e.g., accepted answers, vote counts). Therefore, it is difficult to identify conversations that contain useful information for mining or reading, i.e., conversations of post hoc quality. In this article, we investigate automatically detecting developer conversations of post hoc quality from public chat channels. We first describe an analysis of 400 developer conversations that indicate potential characteristics of post hoc quality, followed by a machine learning-based approach for automatically identifying conversations of post hoc quality. Our evaluation of 2,000 annotated Slack conversations in four programming communities (python, clojure, elm, and racket) indicates that our approach can achieve precision of 0.82, recall of 0.90, F-measure of 0.86, and MCC of 0.57. To our knowledge, this is the first automated technique for detecting developer conversations of post hoc quality.

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

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

[**Chen2016**] Tse-Hsun Chen, Weiyi Shang, Jinqiu Yang, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora. An empirical study on the practice of maintaining object-relational mapping code in Java systems. In *Proc. International Conference on Mining Software Repositories (MSR)*. ACM, 5 2016, DOI `10.1145/2901739.2901758`.

Abstract: Databases have become one of the most important components in modern software systems. For example, web services, cloud computing systems, and online transaction processing systems all rely heavily on databases. To abstract the complexity of accessing a database, developers make use of Object-Relational Mapping (ORM) frameworks. ORM frameworks provide an abstraction layer between the application logic and the underlying database. Such abstraction layer automatically maps objects in Object-Oriented Languages to database records, which significantly reduces the amount of boilerplate code that needs to be written. Despite the advantages of using ORM frameworks, we observe several difficulties in maintaining ORM code (i.e., code that makes use of ORM frameworks) when cooperating with our industrial partner. After conducting studies on other open source systems, we find that such difficulties are common in other Java systems. Our study finds that i) ORM cannot completely encapsulate database accesses in objects or abstract the underlying database technology, thus may cause ORM code changes more scattered; ii) ORM code changes are more frequent than regular code, but there is a lack of tools that help developers verify ORM code at compilation time; iii) we find that changes to ORM code are more commonly due to performance or security reasons; however, traditional static code analyzers need to be extended to capture the peculiarities of ORM code in order to detect such problems. Our study highlights the hidden maintenance costs of using ORM frameworks, and provides some initial insights about potential approaches to help maintain ORM code. Future studies should carefully examine ORM code, especially given the rising use of ORM in modern software systems.

[**Cherubini2007**] Mauro Cherubini, Gina Venolia, Rob DeLine, and Amy J. Ko. Let's go to the whiteboard: how and why software developers use drawings. In *Proc. Conference on Human Factors in Computing Systems (HFCS)*. ACM, 4 2007, DOI `10.1145/1240624.1240714`.
Abstract: Software developers are rooted in the written form of their code, yet they often draw diagrams representing their code. Unfortunately, we still know little about how and why they create these diagrams, and so there is little research to inform the design of visual tools to support developers' work. This paper presents findings from semi-structured interviews that have been validated with a structured survey. Results show that most of the diagrams had a transient nature because of the high cost of changing whiteboard sketches to electronic renderings. Diagrams that documented design decisions were often externalized in these temporary drawings and then subsequently lost. Current visualization tools and the software development practices that we observed do not solve these issues, but these results suggest several directions for future research.

[**Chong2007**] Jan Chong and Tom Hurlbutt. The social dynamics of pair programming. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2007, DOI `10.1109/icse.2007.87`.
Abstract: This paper presents data from a four month ethnographic study

of professional pair programmers from two software development teams. Contrary to the current conception of pair programmers, the pairs in this study did not hew to the separate roles of "driver" and "navigator". Instead, the observed programmers moved together through different phases of the task, considering and discussing issues at the same strategic "range" or level of abstraction and in largely the same role. This form of interaction was reinforced by frequent switches in keyboard control during pairing and the use of dual keyboards. The distribution of expertise among the members of a pair had a strong influence on the tenor of pair programming interaction. Keyboard control had a consistent secondary effect on decisionmaking within the pair. These findings have implications for software development managers and practitioners as well as for the design of software development tools.

[**Cinneide2012**] Mel Ó Cinnéide, Laurence Tratt, Mark Harman, Steve Counsell, and Iman Hemati Moghadam. Experimental assessment of software metrics using automated refactoring. In *Proc. International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM Press, 2012, DOI `10.1145/2372251.2372260`.
Abstract: A large number of software metrics have been proposed in the literature, but there is little understanding of how these metrics relate to one another. We propose a novel experimental technique, based on search-based refactoring, to assess software metrics and to explore relationships between them. Our goal is not to improve the program being refactored, but to assess the software metrics that guide the automated refactoring through repeated refactoring experiments. We apply our approach to five popular cohesion metrics using eight real-world Java systems, involving 300,000 lines of code and over 3,000 refactorings. Our results demonstrate that cohesion metrics disagree with each other in 55% of cases, and show how our approach can be used to reveal novel and surprising insights into the software metrics under investigation.

[**Coelho2016**] Roberta Coelho, Lucas Almeida, Georgios Gousios, Arie van Deursen, and Christoph Treude. Exception handling bug hazards in Android. *Empirical Software Engineering*, 22(3):1264–1304, 8 2016, DOI `10.1007/s10664-016-9443-7`.
Abstract: Adequate handling of exceptions has proven difficult for many software engineers. Mobile app developers in particular, have to cope with compatibility, middleware, memory constraints, and battery restrictions. The goal of this paper is to obtain a thorough understanding of common exception handling bug hazards that app developers face. To that end, we first provide a detailed empirical study of over 6,000 Java exception stack traces we extracted from over 600 open source Android projects. Key insights from this study include common causes for system crashes, and common chains of wrappings between checked and unchecked exceptions. Furthermore, we provide a survey with 71 developers involved in at least one of the projects analyzed. The results corroborate the stack trace findings, and

indicate that developers are unaware of frequently occurring undocumented exception handling behavior. Overall, the findings of our study call for tool support to help developers understand their own and third party exception handling and wrapping logic.

[**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.

[**Costa2019**] Diego Elias Damasceno Costa, Cor-Paul Bezemer, Philip Leitner, and Artur Andrzejak. What's wrong with my benchmark results? studying bad practices in JMH benchmarks. *IEEE Transactions on Software Engineering*, pages 1–1, 2019, DOI `10.1109/tse.2019.2925345`.

Abstract: Microbenchmarking frameworks, such as Java's Microbenchmark Harness (JMH), allow developers to write fine-grained performance test suites at the method or statement level. However, due to the complexities of the Java Virtual Machine, developers often struggle with writing expressive JMH benchmarks which accurately represent the performance of such methods or statements. In this paper, we empirically study bad practices of JMH benchmarks. We present a tool that leverages static analysis to identify 5 bad JMH practices. Our empirical study of 123 open source Java-based systems shows that each of these 5 bad practices are prevalent in open source software. Further, we conduct several experiments to quantify the impact of each bad practice in multiple case studies, and find that bad practices often significantly impact the benchmark results. To validate our experimental results, we constructed seven patches that fix the identified bad practices for six of the studied open source projects, of which six were merged into the main branch of the project. In this paper, we show that developers struggle with accurate Java microbenchmarking, and provide several recommendations to developers of microbenchmarking frameworks on how to improve future versions of their framework.

[**CruzLemus2009**] José A. Cruz-Lemus, Marcela Genero, M. Esperanza Manso, Sandro Morasca, and Mario Piattini. Assessing the understandability of UML statechart diagrams with composite states—a family of empirical studies. *Empirical Software Engineering*, 14(6):685–719, 2 2009, DOI `10.1007/s10664-009-9106-z`.

Abstract: The main goal of this work is to present a family of empirical studies that we have carried out to investigate whether the use of composite states may improve the understandability of UML statechart diagrams derived from class diagrams. Our hypotheses derive from conventional wisdom, which says that hierarchical modeling mechanisms are helpful in mastering the complexity of a software system. In our research, we have carried out three empirical studies, consisting of five experiments in total. The studies differed somewhat as regards the size of the UML statechart models, though their size and the complexity of the models were chosen so that they could be analyzed by the subjects within a limited time period. The studies also differed with respect to the type of subjects (students vs. professionals), the familiarity of the subjects with the domains of the diagrams, and other factors. To integrate the results obtained from each of the five experiments, we performed a meta-analysis study which allowed us to take into account the differences between studies and to obtain the overall effect that the use of composite states has on the understandability of UML statechart diagrams throughout all the experiments. The results obtained are not completely

conclusive. They cast doubts on the usefulness of composite states for a better understanding and memorizing of UML statechart diagrams. Composite states seem only to be helpful for acquiring knowledge from the diagrams. At any rate, it should be noted that these results are affected by the previous experience of the subjects on modeling, as well as by the size and complexity of the UML statechart diagrams we used, so care should be taken when generalizing our results.

[**Dabbish2012**] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in GitHub: transparency and collaboration in an open software repository. In *Proc. Conference on Computer Supported Cooperative Work (CSCW)*. ACM Press, 2012, DOI `10.1145/2145204.2145396`.
Abstract: Social applications on the web let users track and follow the activities of a large number of others regardless of location or affiliation. There is a potential for this transparency to radically improve collaboration and learning in complex knowledge-based activities. Based on a series of in-depth interviews with central and peripheral GitHub users, we examined the value of transparency for large-scale distributed collaborations and communities of practice. We find that people make a surprisingly rich set of social inferences from the networked activity information in GitHub, such as inferring someone else's technical goals and vision when they edit code, or guessing which of several similar projects has the best chance of thriving in the long term. Users combine these inferences into effective strategies for coordinating work, advancing technical skills and managing their reputation.

[**Dagenais2010**] Barthélémy Dagenais and Martin P. Robillard. Creating and evolving developer documentation. In *Proc. International Symposium on the Foundations of Software Engineering (FSE)*. ACM Press, 2010, DOI `10.1145/1882291.1882312`.
Abstract: Developer documentation helps developers learn frameworks and libraries. To better understand how documentation in open source projects is created and maintained, we performed a qualitative study in which we interviewed core contributors who wrote developer documentation and developers who read documentation. In addition, we studied the evolution of 19 documents by analyzing more than 1500 document revisions. We identified the decisions that contributors make, the factors influencing these decisions and the consequences for the project. Among many findings, we observed how working on the documentation could improve the code quality and how constant interaction with the projects' community positively impacted the documentation.

[**Dang2012**] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. ReBucket: a method for clustering duplicate crash reports based on call stack similarity. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 6 2012, DOI `10.1109/icse.2012.6227111`.
Abstract: Software often crashes. Once a crash happens, a crash report could be sent to software developers for investigation upon user permis-

sion. To facilitate efficient handling of crashes, crash reports received by Microsoft's Windows Error Reporting (WER) system are organized into a set of "buckets". Each bucket contains duplicate crash reports that are deemed as manifestations of the same bug. The bucket information is important for prioritizing efforts to resolve crashing bugs. To improve the accuracy of bucketing, we propose ReBucket, a method for clustering crash reports based on call stack matching. ReBucket measures the similarities of call stacks in crash reports and then assigns the reports to appropriate buckets based on the similarity values. We evaluate ReBucket using crash data collected from five widely-used Microsoft products. The results show that ReBucket achieves better overall performance than the existing methods. On average, the F-measure obtained by ReBucket is about 0.88.

[**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.

[**Davis2019**] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. Why aren't regular expressions a lingua franca? An empirical study on the re-use and portability of regular expressions. In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 8 2019, DOI `10.1145/3338906.3338909`.
Abstract: This paper explores the extent to which regular expressions (regexes) are portable across programming languages. Many languages offer similar regex syntaxes, and it would be natural to assume that regexes can be ported across language boundaries. But can regexes be copy/pasted across

language boundaries while retaining their semantic and performance characteristics? In our survey of 158 professional software developers, most indicated that they re-use regexes across language boundaries and about half reported that they believe regexes are a universal language. We experimentally evaluated the riskiness of this practice using a novel regex corpus—537,806 regexes from 193,524 projects written in JavaScript, Java, PHP, Python, Ruby, Go, Perl, and Rust. Using our polyglot regex corpus, we explored the hitherto-unstudied regex portability problems: logic errors due to semantic differences, and security vulnerabilities due to performance differences. We report that developers' belief in a regex lingua franca is understandable but unfounded. Though most regexes compile across language boundaries, 15% exhibit semantic differences across languages and 10% exhibit performance differences across languages. We explained these differences using regex documentation, and further illuminate our findings by investigating regex engine implementations. Along the way we found bugs in the regex engines of JavaScript-V8, Python, Ruby, and Rust, and potential semantic and performance regex bugs in thousands of modules.

[**DeLucia2009**] Andrea De Lucia, Carmine Gravino, Rocco Oliveto, and Genoveffa Tortora. An experimental comparison of ER and UML class diagrams for data modelling. *Empirical Software Engineering*, 15(5):455–492, 12 2009, DOI `10.1007/s10664-009-9127-7`.
Abstract: We present the results of three sets of controlled experiments aimed at analysing whether UML class diagrams are more comprehensible than ER diagrams during data models maintenance. In particular, we considered the support given by the two notations in the comprehension and interpretation of data models, comprehension of the change to perform to meet a change request, and detection of defects contained in a data model. The experiments involved university students with different levels of ability and experience. The results demonstrate that using UML class diagrams subjects achieved better comprehension levels. With regard to the support given by the two notations during maintenance activities the results demonstrate that the two notations give the same support, while in general UML class diagrams provide a better support with respect to ER diagrams during verification activities.

[**DeOliveiraNeto2019**] Francisco Gomes de Oliveira Neto, Richard Torkar, Robert Feldt, Lucas Gren, Carlo A. Furia, and Ziwei Huang. Evolution of statistical analysis in empirical software engineering research: Current state and steps forward. *Journal of Systems and Software*, 156:246–267, 10 2019, DOI `10.1016/j.jss.2019.07.002`.
Abstract: Software engineering research is evolving and papers are increasingly based on empirical data from a multitude of sources, using statistical tests to determine if and to what degree empirical evidence supports their hypotheses. To investigate the practices and trends of statistical analysis in empirical software engineering (ESE), this paper presents a review of a large pool of papers from top-ranked software engineering journals. First, we

manually reviewed 161 papers and in the second phase of our method, we conducted a more extensive semi-automatic classification of papers spanning the years 2001–2015 and 5,196 papers. Results from both review steps was used to: i) identify and analyze the predominant practices in ESE (e.g., using t-test or ANOVA), as well as relevant trends in usage of specific statistical methods (e.g., nonparametric tests and effect size measures) and, ii) develop a conceptual model for a statistical analysis workflow with suggestions on how to apply different statistical methods as well as guidelines to avoid pitfalls. Lastly, we confirm existing claims that current ESE practices lack a standard to report practical significance of results. We illustrate how practical significance can be discussed in terms of both the statistical analysis and in the practitioner's context.

[**DePadua2018**] Guilherme B. de Pádua and Weiyi Shang. Studying the relationship between exception handling practices and post-release defects. In *Proc. International Conference on Mining Software Repositories (MSR)*. ACM, 5 2018, DOI 10.1145/3196398.3196435.
Abstract: Modern programming languages, such as Java and C#, typically provide features that handle exceptions. These features separate error-handling code from regular source code and aim to assist in the practice of software comprehension and maintenance. Nevertheless, their misuse can still cause reliability degradation or even catastrophic software failures. Prior studies on exception handling revealed the suboptimal practices of the exception handling flows and the prevalence of their anti-patterns. However, little is known about the relationship between exception handling practices and software quality. In this work, we investigate the relationship between software quality (measured by the probability of having post-release defects) and: (i) exception flow characteristics and (ii) 17 exception handling anti-patterns. We perform a case study on three Java and C# open-source projects. By building statistical models of the probability of post-release defects using traditional software metrics and metrics that are associated with exception handling practice, we study whether exception flow characteristics and exception handling anti-patterns have a statistically significant relationship with post-release defects. We find that exception flow characteristics in Java projects have a significant relationship with post-release defects. In addition, although the majority of the exception handing anti-patterns are not significant in the models, there exist anti-patterns that can provide significant explanatory power to the probability of post-release defects. Therefore, development teams should consider allocating more resources to improving their exception handling practices and avoid the anti-patterns that are found to have a relationship with post-release defects. Our findings also highlight the need for techniques that assist in handling exceptions in the software development practice.

[**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.

[**Devanbu2016**] Prem Devanbu, Thomas Zimmermann, and Christian Bird. Belief & evidence in empirical software engineering. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 5 2016, DOI `10.1145/2884781.2884812`.
Abstract: Empirical software engineering has produced a steady stream of evidence-based results concerning the factors that affect important outcomes such as cost, quality, and interval. However, programmers often also have strongly-held a priori opinions about these issues. These opinions are important, since developers are highly trained professionals whose beliefs would doubtless affect their practice. As in evidence-based medicine, disseminating empirical findings to developers is a key step in ensuring that the findings impact practice. In this paper, we describe a case study, on the prior beliefs of developers at Microsoft, and the relationship of these beliefs to actual empirical data on the projects in which these developers work. Our findings are that a) programmers do indeed have very strong beliefs on certain topics b) their beliefs are primarily formed based on personal experience, rather than on findings in empirical research and c) beliefs can vary with each project, but do not necessarily correspond with actual evidence in that project. Our findings suggest that more effort should be taken to disseminate empirical findings to developers and that more in-depth study the interplay of belief and evidence in software practice is needed.

[**Dias2021**] Edson Dias, Paulo Meirelles, Fernando Castor, Igor Steinmacher, Igor Wiese, and Gustavo Pinto. What makes a great maintainer of open source projects? In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2021, DOI `10.1109/icse43902.2021.00093`.
Abstract: Although Open Source Software (OSS) maintainers devote a significant proportion of their work to coding tasks, great maintainers must excel in many other activities beyond coding. Maintainers should care about fostering a community, helping new members to find their place, while also saying "no" to patches that although are well-coded and well-tested, do not

contribute to the goal of the project. To perform all these activities masterfully, maintainers should exercise attributes that software engineers (working on closed source projects) do not always need to master. This paper aims to uncover, relate, and prioritize the unique attributes that great OSS maintainers might have. To achieve this goal, we conducted 33 semi-structured interviews with well-experienced maintainers that are the gatekeepers of notable projects such as the Linux Kernel, the Debian operating system, and the GitLab coding platform. After we analyzed the interviews and curated a list of attributes, we created a conceptual framework to explain how these attributes are connected. We then conducted a rating survey with 90 OSS contributors. We noted that "technical excellence" and "communication" are the most recurring attributes. When grouped, these attributes fit into four broad categories: management, social, technical, and personality. While we noted that "sustain a long term vision of the project" and being "extremely careful" seem to form the basis of our framework, we noted through our survey that the communication attribute was perceived as the most essential one.

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

[**Durieux2020**] Thomas Durieux, Claire Le Goues, Michael Hilton, and Rui Abreu. Empirical study of restarted and flaky builds on travis CI. In *Proc. International Conference on Mining Software Repositories (MSR)*. ACM, 6 2020, DOI `10.1145/3379597.3387460`.
Abstract: Continuous Integration (CI) is a development practice where developers frequently integrate code into a common codebase. After the code is integrated, the CI server runs a test suite and other tools to produce a set of reports (e.g., the output of linters and tests). If the result of a CI test run is unexpected, developers have the option to manually restart the build, re-running the same test suite on the same code; this can reveal build flakiness, if the restarted build outcome differs from the original build. In this study, we analyze restarted builds, flaky builds, and their impact on the development workflow. We observe that developers restart at least 1.72%

of builds, amounting to 56,522 restarted builds in our Travis CI dataset. We observe that more mature and more complex projects are more likely to include restarted builds. The restarted builds are mostly builds that are initially failing due to a test, network problem, or a Travis CI limitations such as execution timeout. Finally, we observe that restarted builds have an impact on development workflow. Indeed, in 54.42% of the restarted builds, the developers analyze and restart a build within an hour of the initial build execution. This suggests that developers wait for CI results, interrupting their workflow to address the issue. Restarted builds also slow down the merging of pull requests by a factor of three, bringing median merging time from 16h to 48h.

[**Dyba2006**] Tore Dybå, Vigdis By Kampenes, and Dag I.K. Sjøberg. A systematic review of statistical power in software engineering experiments. *Information and Software Technology*, 48(8):745–755, 8 2006, DOI `10.1016/j.infsof.2005.08.009`.
Abstract: Statistical power is an inherent part of empirical studies that employ significance testing and is essential for the planning of studies, for the interpretation of study results, and for the validity of study conclusions. This paper reports a quantitative assessment of the statistical power of empirical software engineering research based on the 103 papers on controlled experiments (of a total of 5,453 papers) published in nine major software engineering journals and three conference proceedings in the decade 1993–2002. The results show that the statistical power of software engineering experiments falls substantially below accepted norms as well as the levels found in the related discipline of information systems research. Given this study's findings, additional attention must be directed to the adequacy of sample sizes and research designs to ensure acceptable levels of statistical power. Furthermore, the current reporting of significance tests should be enhanced by also reporting effect sizes and confidence intervals.

[**Dzidek2008**] W.J. Dzidek, E. Arisholm, and L.C. Briand. A realistic empirical evaluation of the costs and benefits of UML in software maintenance. *IEEE Transactions on Software Engineering*, 34(3):407–432, 5 2008, DOI `10.1109/tse.2008.15`.
Abstract: The Unified Modeling Language (UML) is the de facto standard for object-oriented software analysis and design modeling. However, few empirical studies exist that investigate the costs and evaluate the benefits of using UML in realistic contexts. Such studies are needed so that the software industry can make informed decisions regarding the extent to which they should adopt UML in their development practices. This is the first controlled experiment that investigates the costs of maintaining and the benefits of using UML documentation during the maintenance and evolution of a real, non-trivial system, using professional developers as subjects, working with a state-of-the-art UML tool during an extended period of time. The subjects in the control group had no UML documentation. In this experiment, the subjects in the UML group had on average a practically and statistically

significant 54% increase in the functional correctness of changes (p=0.03), and an insignificant 7% overall improvement in design quality (p=0.22) - though a much larger improvement was observed on the first change task (56%) - at the expense of an insignificant 14% increase in development time caused by the overhead of updating the UML documentation (p=0.35).

[**Eichberg2015**] Michael Eichberg, Ben Hermann, Mira Mezini, and Leonid Glanz. Hidden truths in dead software paths. In *Proc. International Symposium on the Foundations of Software Engineering (FSE)*. ACM, 8 2015, DOI `10.1145/2786805.2786865`.

Abstract: Approaches and techniques for statically finding a multitude of issues in source code have been developed in the past. A core property of these approaches is that they are usually targeted towards finding only a very specific kind of issue and that the effort to develop such an analysis is significant. This strictly limits the number of kinds of issues that can be detected. In this paper, we discuss a generic approach based on the detection of infeasible paths in code that can discover a wide range of code smells ranging from useless code that hinders comprehension to real bugs. Code issues are identified by calculating the difference between the control-flow graph that contains all technically possible edges and the corresponding graph recorded while performing a more precise analysis using abstract interpretation. We have evaluated the approach using the Java Development Kit as well as the Qualitas Corpus (a curated collection of over 100 Java Applications) and were able to find thousands of issues across a wide range of categories.

[**ElEmam2001**] K. El Emam, S. Benlarbi, N. Goel, and S.N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27(7):630–650, 7 2001, DOI `10.1109/32.935855`.

Abstract: Much effort has been devoted to the development and empirical validation of object-oriented metrics. The empirical validations performed thus far would suggest that a core set of validated metrics is close to being identified. However, none of these studies allow for the potentially confounding effect of class size. We demonstrate a strong size confounding effect and question the results of previous object-oriented metrics validation studies. We first investigated whether there is a confounding effect of class size in validation studies of object-oriented metrics and show that, based on previous work, there is reason to believe that such an effect exists. We then describe a detailed empirical methodology for identifying those effects. Finally, we perform a study on a large C++ telecommunications framework to examine if size is really a confounder. This study considered the Chidamber and Kemerer metrics and a subset of the Lorenz and Kidd metrics. The dependent variable was the incidence of a fault attributable to a field failure (fault-proneness of a class). Our findings indicate that, before controlling for size, the results are very similar to previous studies. The metrics that are expected to be validated are indeed associated with fault-proneness.

[**Fagerholm2017**] Fabian Fagerholm, Marco Kuhrmann, and Jürgen Münch. Guidelines for using empirical studies in software engineering education. *PeerJ Computer Science*, 3:e131, 9 2017, DOI `10.7717/peerj-cs.131`.
Abstract: Software engineering education is supposed to provide students with industry-relevant knowledge and skills. Educators must address issues beyond exercises and theories that can be directly rehearsed in small settings. A way to experience such efects and to increase the relevance of software engineering education is to apply empirical studies in teaching. In our article, we show how diferent types of empirical studies can be used for educational purposes in software engineering. We give examples illustrating how to utilize empirical studies, discuss challenges, and derive an initial guideline that supports teachers to include empirical studies in software engineering courses. This summary refers to the paper Guidelines for Using Empirical Studies in Software Engineering Education [FKM17]. This paper was published in the PeerJ Computer Science journal.

[**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.

[**Feal2020**] Álvaro Feal, Paolo Calciati, Narseo Vallina-Rodriguez, Carmela Troncoso, and Alessandra Gorla. Angel or devil? a privacy study of mobile parental control apps. *Proceedings on Privacy Enhancing Technologies*, 2020(2):314–335, 4 2020, DOI `10.2478/popets-2020-0029`.
Abstract: Android parental control applications are used by parents to monitor and limit their children's mobile behaviour (e.g., mobile apps usage, web browsing, calling, and texting). In order to offer this service, parental control apps require privileged access to sys-tem resources and access to sensitive data. This may significantly reduce the dangers associated with kids' online activities, but it raises important privacy con-cerns. These concerns have so far been overlooked by organizations providing recommendations regarding the use of parental control applications to the public. We conduct

the first in-depth study of the Android parental control app's ecosystem from a privacy and regulatory point of view. We exhaustively study 46 apps from 43 developers which have a combined 20M installs in the Google Play Store. Using a combination of static and dynamic analysis we find that: these apps are on average more permissions-hungry than the top 150 apps in the Google Play Store, and tend to request more dangerous permissions with new releases; 11% of the apps transmit personal data in the clear; 34% of the apps gather and send personal information without appropriate consent; and 72% of the apps share data with third parties (including online advertising and analytics services) without mentioning their presence in their privacy policies. In summary, parental control applications lack transparency and lack compliance with reg ulatory requirements. This holds even for those applications recommended by European and other national security centers.

[**Ferreira2021**] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. Containing malicious package updates in npm with a lightweight permission system. *CoRR*, abs/2103.05769, 2021.
Abstract: The large amount of third-party packages available in fast-moving software ecosystems, such as Node.js/npm, enables attackers to compromise applications by pushing malicious updates to their package dependencies. Studying the npm repository, we observed that many packages in the npm repository that are used in Node.js applications perform only simple computations and do not need access to filesystem or network APIs. This offers the opportunity to enforce least-privilege design per package, protecting applications and package dependencies from malicious updates. We propose a lightweight permission system that protects Node.js applications by enforcing package permissions at runtime. We discuss the design space of solutions and show that our system makes a large number of packages much harder to be exploited, almost for free.

[**Fischer2015**] Lars Fischer and Stefan Hanenberg. An empirical investigation of the effects of type systems and code completion on API usability using TypeScript and JavaScript in MS Visual Studio. In *Proc. Symposium on Dynamic Languages (DL)*. ACM, 10 2015, DOI `10.1145/2816707.2816720`.
Abstract: Recent empirical studies that compared static and dynamic type systems on API usability showed a positive impact of static type systems on developer productivity in most cases. Nevertheless, it is unclear how large this effect is in comparison to other factors. One obvious factor in programming is tooling: It is commonly accepted that modern IDEs have a large positive impact on developers, although it is not clear which parts of modern IDEs are responsible for that. One possible—and for most developers obvious candidate—is code completion. This paper describes a 2x2 randomized trial that compares JavaScript and Microsoft's statically typed alternative TypeScript with and without code completion in MS Visual Studio. While the experiment shows (in correspondence to previous experiments) a large positive effect of the statically typed language TypeScript, the code completion effect is not only marginal, but also just approaching statistical significance.

This seems to be an indicator that the effect of static type systems is larger than often assumed, at least in comparison to code completion.

[**Flint2021**] Samuel W. Flint, Jigyasa Chauhan, and Robert Dyer. Escaping the time pit: Pitfalls and guidelines for using time-based git data. In *Proc. International Conference on Mining Software Repositories (MSR)*. IEEE, 5 2021, DOI 10.1109/msr52588.2021.00022.

Abstract: Many software engineering research papers rely on time-based data (e.g., commit timestamps, issue report creation/update/close dates, release dates). Like most real-world data however, time-based data is often dirty. To date, there are no studies that quantify how frequently such data is used by the software engineering research community, or investigate sources of and quantify how often such data is dirty. Depending on the research task and method used, including such dirty data could affect the research results. This paper presents the first survey of papers that utilize time-based data, published in the Mining Software Repositories (MSR) conference series. Out of the 690 technical track and data papers published in MSR 2004–2020, we saw at least 35% of papers utilized time-based data. We then used the Boa and Software Heritage infrastructures to help identify and quantify several sources of dirty commit timestamp data. Finally we provide guidelines/best practices for researchers utilizing time-based data from Git repositories.

[**Ford2016**] Denae Ford, Justin Smith, Philip J. Guo, and Chris Parnin. Paradise unplugged: identifying barriers for female participation on Stack Overflow. In *Proc. International Symposium on the Foundations of Software Engineering (FSE)*. ACM, 11 2016, DOI 10.1145/2950290.2950331.

Abstract: It is no secret that females engage less in programming fields than males. However, in online communities, such as Stack Overflow, this gender gap is even more extreme: only 5.8% of contributors are female. In this paper, we use a mixed-methods approach to identify contribution barriers females face in online communities. Through 22 semi-structured interviews with a spectrum of female users ranging from non-contributors to a top 100 ranked user of all time, we identified 14 barriers preventing them from contributing to Stack Overflow. We then conducted a survey with 1470 female and male developers to confirm which barriers are gender related or general problems for everyone. Females ranked five barriers significantly higher than males. A few of these include doubts in the level of expertise needed to contribute, feeling overwhelmed when competing with a large number of users, and limited awareness of site features. Still, there were other barriers that equally impacted all Stack Overflow users or affected particular groups, such as industry programmers. Finally, we describe several implications that may encourage increased participation in the Stack Overflow community across genders and other demographics.

[**Ford2017**] Denae Ford, Tom Zimmermann, Christian Bird, and Nachiappan Nagappan. Characterizing software engineering work with personas based on knowledge worker actions. In *Proc. International Symposium on Empir-*

41

ical Software Engineering and Measurement (ESEM). IEEE, 11 2017, DOI `10.1109/esem.2017.54`.

Abstract: Mistaking versatility for universal skills, some companies tend to categorize all software engineers the same not knowing a difference exists. For example, a company may select one of many software engineers to complete a task, later finding that the engineer's skills and style do not match those needed to successfully complete that task. This can result in delayed task completion and demonstrates that a one-size fits all concept should not apply to how software engineers work. In order to gain a comprehensive understanding of different software engineers and their working styles we interviewed 21 participants and surveyed 868 software engineers at a large software company and asked them about their work in terms of knowledge worker actions. We identify how tasks, collaboration styles, and perspectives of autonomy can significantly effect different approaches to software engineering work. To characterize differences, we describe empirically informed personas on how they work. Our defined software engineering personas include those with focused debugging abilities, engineers with an active interest in learning, experienced advisors who serve as experts in their role, and more. Our study and results serve as a resource for building products, services, and tools around these software engineering personas.

[**Ford2019**] Denae Ford, Mahnaz Behroozi, Alexander Serebrenik, and Chris Parnin. Beyond the code itself: how programmers really look at pull requests. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2019, DOI `10.1109/icse-seis.2019.00014`.

Abstract: Developers in open source projects must make decisions on contributions from other community members, such as whether or not to accept a pull request. However, secondary factors-beyond the code itself-can influence those decisions. For example, signals from GitHub profiles, such as a number of followers, activity, names, or gender can also be considered when developers make decisions. In this paper, we examine how developers use these signals (or not) when making decisions about code contributions. To evaluate this question, we evaluate how signals related to perceived gender identity and code quality influenced decisions on accepting pull requests. Unlike previous work, we analyze this decision process with data collected from an eye-tracker. We analyzed differences in what signals developers said are important for themselves versus what signals they actually used to make decisions about others. We found that after the code snippet (x=57%), the second place programmers spent their time fixating is on supplemental technical signals (x=32%), such as previous contributions and popular repositories. Diverging from what participants reported themselves, we also found that programmers fixated on social signals more than recalled.

[**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.

[**Freeman1972**] Jo Freeman. The tyranny of structurelessness. *The Second Wave*, 2(1), 1972.
Abstract: An influential essay pointing out that every organization has a power structure; the only question is whether it's formal and accountable, or informal and unaccountable.

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

[**Fritzsch2021**] Jonas Fritzsch, Marvin Wyrich, Justus Bogner, and Stefan Wagner. Résumé-driven development: A definition and empirical characterization. In *Proc. International Conference on Software Engineering (ICSE).* IEEE, 5 2021, DOI `10.1109/icse-seis52602.2021.00011`.

Abstract: Technologies play an important role in the hiring process for software professionals. Within this process, several studies revealed misconceptions and bad practices which lead to suboptimal recruitment experiences. In the same context, grey literature anecdotally coined the term Résumé-Driven Development (RDD), a phenomenon describing the overemphasis of trending technologies in both job offerings and resumes as an interaction between employers and applicants. While RDD has been sporadically mentioned in books and online discussions, there are so far no scientific studies on the topic, despite its potential negative consequences. We therefore empirically investigated this phenomenon by surveying 591 software professionals in both hiring (130) and technical (558) roles and identified RDD facets in substantial parts of our sample: 60% of our hiring professionals agreed that trends influence their job offerings, while 82% of our software professionals believed that using trending technologies in their daily work makes them more attractive for prospective employers. Grounded in the survey results, we conceptualize a theory to frame and explain Résumé-Driven Development. Finally, we discuss influencing factors and consequences and propose a definition of the term. Our contribution provides a foundation for future research and raises awareness for a potentially systemic trend that may broadly affect the software industry.

[**Fu2016**] Wei Fu, Tim Menzies, and Xipeng Shen. Tuning for software analytics: Is it really necessary? *Information and Software Technology*, 76:135–146, 8 2016, DOI `10.1016/j.infsof.2016.04.017`.

Abstract: Context: Data miners have been widely used in software engineering to, say, generate defect predictors from static code measures. Such static code defect predictors perform well compared to manual methods, and they are easy to use and useful to use. But one of the "black arts" of data mining is setting the tunings that control the miner.Objective: We seek simple, automatic, and very effective method for finding those tunings.Method: For each experiment with different data sets (from open source JAVA systems), we ran differential evolution as an optimizer to explore the tuning space (as a first step) then tested the tunings using hold-out data.Results: Contrary to our prior expectations, we found these tunings were remarkably simple: it only required tens, not thousands, of attempts to obtain very good results. For example, when learning software defect predictors, this method can quickly find tunings that alter detection precision from 0% to 60%. Conclusion: Since (1) the improvements are so large, and (2) the tuning is so simple, we need to change standard methods in software analytics. At least for defect prediction, it is no longer enough to just run a data miner and present the result without conducting a tuning optimization study. The implication for other kinds of analytics is now an open and pressing issue.

[**Fucci2016**] Davide Fucci, Giuseppe Scanniello, Simone Romano, Martin Shepperd, Boyce Sigweni, Fernando Uyaguari, Burak Turhan, Natalia Juristo, and Markku Oivo. An external replication on the effects of test-driven development using a multi-site blind analysis approach. In *Proc. International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, 9 2016, DOI 10.1145/2961111.2962592.
Abstract: Context: Test-driven development (TDD) is an agile practice claimed to improve the quality of a software product, as well as the productivity of its developers. A previous study (i.e., baseline experiment) at the University of Oulu (Finland) compared TDD to a test-last development (TLD) approach through a randomized controlled trial. The results failed to support the claims. Goal: We want to validate the original study results by replicating it at the University of Basilicata (Italy), using a different design. Method: We replicated the baseline experiment, using a crossover design, with 21 graduate students. We kept the settings and context as close as possible to the baseline experiment. In order to limit researchers bias, we involved two other sites (UPM, Spain, and Brunel, UK) to conduct blind analysis of the data. Results: The Kruskal-Wallis tests did not show any significant difference between TDD and TLD in terms of testing effort (p-value = .27), external code quality (p-value = .82), and developers' productivity (p-value = .83). Nevertheless, our data revealed a difference based on the order in which TDD and TLD were applied, though no carry over effect. Conclusions: We verify the baseline study results, yet our results raises concerns regarding the selection of experimental objects, particularly with respect to their interaction with the order in which of treatments are applied. We recommend future studies to survey the tasks used in experiments evaluating TDD. Finally, to lower the cost of replication studies and reduce researchers' bias, we encourage other research groups to adopt similar multi-site blind analysis approach described in this paper.

[**Fucci2020**] Davide Fucci, Giuseppe Scanniello, Simone Romano, and Natalia Juristo. Need for sleep: The impact of a night of sleep deprivation on novice developers' performance. *IEEE Transactions on Software Engineering*, 46(1):1–19, 1 2020, DOI 10.1109/tse.2018.2834900.
Abstract: We present a quasi-experiment to investigate whether, and to what extent, sleep deprivation impacts the performance of novice software developers using the agile practice of test-first development (TFD). We recruited 45 undergraduates, and asked them to tackle a programming task. Among the participants, 23 agreed to stay awake the night before carrying out the task, while 22 slept normally. We analyzed the quality (i.e., the functional correctness) of the implementations delivered by the participants in both groups, their engagement in writing source code (i.e., the amount of activities performed in the IDE while tackling the programming task) and ability to apply TFD (i.e., the extent to which a participant is able to apply this practice). By comparing the two groups of participants, we found that a single night of sleep deprivation leads to a reduction of 50 percent

in the quality of the implementations. There is notable evidence that the developers' engagement and their prowess to apply TFD are negatively impacted. Our results also show that sleep-deprived developers make more fixes to syntactic mistakes in the source code. We conclude that sleep deprivation has possibly disruptive effects on software development activities. The results open opportunities for improving developers' performance by integrating the study of sleep with other psycho-physiological factors in which the software engineering research community has recently taken an interest in.

[**Furia2019**] Carlo Alberto Furia, Robert Feldt, and Richard Torkar. Bayesian data analysis in empirical software engineering research. *IEEE Transactions on Software Engineering*, pages 1–1, 2019, DOI `10.1109/tse.2019.2935974`.

Abstract: Statistics comes in two main flavors: frequentist and Bayesian. For historical and technical reasons, frequentist statistics have traditionally dominated empirical data analysis, and certainly remain prevalent in empirical software engineering. This situation is unfortunate because frequentist statistics suffer from a number of shortcomings—such as lack of flexibility and results that are unintuitive and hard to interpret—that curtail their effectiveness when dealing with the heterogeneous data that is increasingly available for empirical analysis of software engineering practice. In this paper, we pinpoint these shortcomings, and present Bayesian data analysis techniques that work better on the same data—as they can provide clearer results that are simultaneously robust and nuanced. After a short, high-level introduction to the basic tools of Bayesian statistics, our presentation targets the reanalysis of two empirical studies targeting data about the effectiveness of automatically generated tests and the performance of programming languages. By contrasting the original frequentist analysis to our new Bayesian analysis, we demonstrate concrete advantages of using Bayesian techniques, and we advocate a prominent role for them in empirical software engineering research and practice.

[**Gao2017**] Zheng Gao, Christian Bird, and Earl T. Barr. To type or not to type: quantifying detectable bugs in JavaScript. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2017, DOI `10.1109/icse.2017.75`.

Abstract: JavaScript is growing explosively and is now used in large mature projects even outside the web domain. JavaScript is also a dynamically typed language for which static type systems, notably Facebook's Flow and Microsoft's TypeScript, have been written. What benefits do these static type systems provide? Leveraging JavaScript project histories, we select a fixed bug and check out the code just prior to the fix. We manually add type annotations to the buggy code and test whether Flow and TypeScript report an error on the buggy code, thereby possibly prompting a developer to fix the bug before its public release. We then report the proportion of bugs on which these type systems reported an error. Evaluating static type systems against public bugs, which have survived testing and review,

is conservative: it understates their effectiveness at detecting bugs during private development, not to mention their other benefits such as facilitating code search/completion and serving as documentation. Despite this uneven playing field, our central finding is that both static type systems find an important percentage of public bugs: both Flow 0.30 and TypeScript 2.0 successfully detect 15%!.

[**Gao2020**] Gao Gao, Finn Voichick, Michelle Ichinco, and Caitlin Kelleher. Exploring programmers' API learning processes: Collecting web resources as external memory. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 8 2020, DOI `10.1109/vl/hcc50065.2020.9127274`.

Abstract: Modern programming frequently requires the use of APIs (Application Programming Interfaces). Yet many programmers struggle when trying to learn APIs. We ran an exploratory study in which we observed participants performing an API learning task. We analyze their processes using a proposed model of API learning, grounded in Cognitive Load Theory, Information Foraging Theory, and External Memory research. The results provide support for the model of API Learning and add new insights into the form and usage of external memory while learning APIs. Programmers quickly curated a set of API resources through Information Foraging which served as external memory and then primarily referred to these resources to meet information needs while coding.

[**Garcia2021**] Boni García, Mario Munoz-Organero, Carlos Alario-Hoyos, and Carlos Delgado Kloos. Automated driver management for selenium WebDriver. *Empirical Software Engineering*, 26(5), 7 2021, DOI `10.1007/s10664-021-09975-3`.

Abstract: Selenium WebDriver is a framework used to control web browsers automatically. It provides a cross-browser Application Programming Interface (API) for different languages (e.g., Java, Python, or JavaScript) that allows automatic navigation, user impersonation, and verification of web applications. Internally, Selenium WebDriver makes use of the native automation support of each browser. Hence, a platform-dependent binary file (the so-called driver) must be placed between the Selenium WebDriver script and the browser to support this native communication. The management (i.e., download, setup, and maintenance) of these drivers is cumbersome for practitioners. This paper provides a complete methodology to automate this management process. Particularly, we present WebDriverManager, the reference tool implementing this methodology. WebDriverManager provides different execution methods: as a Java dependency, as a Command-Line Interface (CLI) tool, as a server, as a Docker container, and as a Java agent. To provide empirical validation of the proposed approach, we surveyed the WebDriverManager users. The aim of this study is twofold. First, we assessed the extent to which WebDriverManager is adopted and used. Second, we evaluated the WebDriverManager API following Clarke's usability dimensions. A total of 148 participants worldwide completed this survey in

2020. The results show a remarkable assessment of the automation capabilities and API usability of WebDriverManager by Java users, but a scarce adoption for other languages.

[**Gauthier2013**] Francois Gauthier and Ettore Merlo. Semantic smells and errors in access control models: a case study in PHP. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2013, DOI `10.1109/icse.2013.6606670`.
Abstract: Access control models implement mechanisms to restrict access to sensitive data from unprivileged users. Access controls typically check privileges that capture the semantics of the operations they protect. Semantic smells and errors in access control models stem from privileges that are partially or totally unrelated to the action they protect. This paper presents a novel approach, partly based on static analysis and information retrieval techniques, for the automatic detection of semantic smells and errors in access control models. Investigation of the case study application revealed 31 smells and 2 errors. Errors were reported to developers who quickly confirmed their relevance and took actions to correct them. Based on the obtained results, we also propose three categories of semantic smells and errors to lay the foundations for further research on access control smells in other systems and domains.

[**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.

[**Ghiotto2020**] Gleiph Ghiotto, Leonardo Murta, Marcio Barros, and André van der Hoek. On the nature of merge conflicts: a study of 2,731 open source Java projects hosted by GitHub. *IEEE Transactions on Software Engineering*, 46(8):892–915, 8 2020, DOI `10.1109/tse.2018.2871083`.

Abstract: When multiple developers change a software system in parallel, these concurrent changes need to be merged to all appear in the software being developed. Numerous merge techniques have been proposed to support this task, but none of them can fully automate the merge process. Indeed, it has been reported that as much as 10 to 20 percent of all merge attempts result in a merge conflict, meaning that a developer has to manually complete the merge. To date, we have little insight into the nature of these merge conflicts. What do they look like, in detail? How do developers resolve them? Do any patterns exist that might suggest new merge techniques that could reduce the manual effort? This paper contributes an in-depth study of the merge conflicts found in the histories of 2,731 open source Java projects. Seeded by the manual analysis of the histories of five projects, our automated analysis of all 2,731 projects: (1) characterizes the merge conflicts in terms of number of chunks, size, and programming language constructs involved, (2) classifies the manual resolution strategies that developers use to address these merge conflicts, and (3) analyzes the relationships between various characteristics of the merge conflicts and the chosen resolution strategies. Our results give rise to three primary recommendations for future merge techniques, that—when implemented—could on one hand help in automatically resolving certain types of conflicts and on the other hand provide the developer with tool-based assistance to more easily resolve other types of conflicts that cannot be automatically resolved.

[**Giger2011**] Emanuel Giger, Martin Pinzger, and Harald Gall. Using the Gini Coefficient for bug prediction in Eclipse. In *Proc. International Workshop on Principles on Software Evolution/Workshop on Software Evolution (IWPSE-EVOL)*. ACM Press, 2011, DOI `10.1145/2024445.2024455`.

Abstract: The Gini coefficient is a prominent measure to quantify the inequality of a distribution. It is often used in the field of economy to describe how goods, e.g., wealth or farmland, are distributed among people. We use the Gini coefficient to measure code ownership by investigating how changes made to source code are distributed among the developer population. The results of our study with data from the Eclipse platform show that less bugs can be expected if a large share of all changes are accumulated, i.e., carried out, by relatively few developers.

[**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. Further-

more, 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.

[**Golubev2021**] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. One thousand and one stories: a large-scale survey of software refactoring. In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 8 2021, DOI `10.1145/3468264.3473924`.
Abstract: Despite the availability of refactoring as a feature in popular IDEs, recent studies revealed that developers are reluctant to use them, and still prefer the manual refactoring of their code. At JetBrains, our goal is to fully support refactoring features in IntelliJ-based IDEs and improve their adoption in practice. Therefore, we start by raising the following main questions. How exactly do people refactor code? What refactorings are the most popular? Why do some developers tend not to use convenient IDE refactoring tools? In this paper, we investigate the raised questions through the design and implementation of a survey targeting 1,183 users of IntelliJ-based IDEs. Our quantitative and qualitative analysis of the survey results shows that almost two-thirds of developers spend more than one hour in a single session refactoring their code; that refactoring types vary greatly in popularity; and that a lot of developers would like to know more about IDE refactoring features but lack the means to do so. These results serve us internally to support the next generation of refactoring features, as well as can help our research community to establish new directions in the refactoring usability research.

[**Gousios2016**] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. Work practices and challenges in pull-based development. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 5 2016, DOI `10.1145/2884781.2884826`.

Abstract: The pull-based development model is an emerging way of contributing to distributed software projects that is gaining enormous popularity within the open source software (OSS) world. Previous work has examined this model by focusing on projects and their owners—we complement it by examining the work practices of project contributors and the challenges they face. We conducted a survey with 645 top contributors to active OSS projects using the pull-based model on GitHub, the prevalent social coding site. We also analyzed traces extracted from corresponding GitHub repositories. Our research shows that: contributors have a strong interest in maintaining awareness of project status to get inspiration and avoid duplicating work, but they do not actively propagate information; communication within pull requests is reportedly limited to low-level concerns and contributors often use communication channels external to pull requests; challenges are mostly social in nature, with most reporting poor responsiveness from integrators; and the increased transparency of this setting is a confirmed motivation to contribute. Based on these findings, we present recommendations for practitioners to streamline the contribution process and discuss potential future research directions.

[**Graziotin2014**] Daniel Graziotin, Xiaofeng Wang, and Pekka Abrahamsson. Happy software developers solve problems better: psychological measurements in empirical software engineering. *PeerJ Computer Science*, 2:e289, 3 2014, DOI `10.7717/peerj.289`.
Abstract: For more than thirty years, it has been claimed that a way to improve software developers' productivity and software quality is to focus on people and to provide incentives to make developers satisfied and happy. This claim has rarely been verified in software engineering research, which faces an additional challenge in comparison to more traditional engineering fields: software development is an intellectual activity and is dominated by often-neglected human factors (called human aspects in software engineering research). Among the many skills required for software development, developers must possess high analytical problem-solving skills and creativity for the software construction process. According to psychology research, affective states—emotions and moods—deeply influence the cognitive processing abilities and performance of workers, including creativity and analytical problem solving. Nonetheless, little research has investigated the correlation between the affective states, creativity, and analytical problem-solving performance of programmers. This article echoes the call to employ psychological measurements in software engineering research. We report a study with 42 participants to investigate the relationship between the affective states, creativity, and analytical problem-solving skills of software developers. The results offer support for the claim that happy developers are indeed better problem solvers in terms of their analytical abilities. The following contributions are made by this study: (1) providing a better understanding of the impact of affective states on the creativity and analytical problem-solving capacities of developers, (2) introducing and validating psychologi-

cal measurements, theories, and concepts of affective states, creativity, and analytical-problem-solving skills in empirical software engineering, and (3) raising the need for studying the human factors of software engineering by employing a multidisciplinary viewpoint.

[**Green1996**] Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 6 1996, DOI `10.1006/jvlc.1996.0009`.

Abstract: Abstract The cognitive dimensions framework is a broad-brush evaluation technique for interactive devices and for non-interactive notations. It sets out a small vocabulary of terms designed to capture the cognitively-relevant aspects of structure, and shows how they can be traded off against each other. The purpose of this paper is to propose the framework as an evaluation technique for visual programming environments. We apply it to two commercially-available dataflow languages (with further examples from other systems) and conclude that it is effective and insightful; other HCI-based evaluation techniques focus on different aspects and would make good complements. Insofar as the examples we used are representative, current VPLs are successful in achieving a good 'closeness of match', but designers need to consider the 'viscosity ' (resistance to local change) and the 'secondary notation' (possibility of conveying extra meaning by choice of layout, colour, etc.).

[**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.

[**Gulzar2016**] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. BigDebug: debugging primitives for interactive big data processing in Spark. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 5 2016, DOI 10.1145/2884781.2884813.

Abstract: Developers use cloud computing platforms to process a large quantity of data in parallel when developing big data analytics. Debugging the massive parallel computations that run in today's data-centers is time consuming and error-prone. To address this challenge, we design a set of interactive, real-time debugging primitives for big data processing in Apache Spark, the next generation data-intensive scalable cloud computing platform. This requires re-thinking the notion of step-through debugging in a traditional debugger such as gdb, because pausing the entire computation across distributed worker nodes causes significant delay and naively inspecting millions of records using a watchpoint is too time consuming for an end user.First, BigDebug's simulated breakpoints and on-demand watchpoints allow users to selectively examine distributed, intermediate data on the cloud with little overhead. Second, a user can also pinpoint a crash-inducing record and selectively resume relevant sub-computations after a quick fix. Third, a user can determine the root causes of errors (or delays) at the level of individual records through a fine-grained data provenance capability. Our evaluation shows that BigDebug scales to terabytes and its record-level tracing incurs less than 25% overhead on average. It determines crash culprits orders of magnitude more accurately and provides up to 100% time saving compared to the baseline replay debugger. The results show that BigDebug supports debugging at interactive speeds with minimal performance impact.

[**Han2021**] Junxiao Han, Shuiguang Deng, David Lo, Chen Zhi, Jianwei Yin, and Xin Xia. An empirical study of the landscape of open source projects in baidu, alibaba, and tencent. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2021, DOI 10.1109/icse-seip52600.2021.00039.

Abstract: Open source software has drawn more and more attention from researchers, developers and companies nowadays. Meanwhile, many Chinese technology companies are embracing open source and choosing to open source their projects. Nevertheless, most previous studies are concentrated on international companies such as Microsoft or Google, while the practical values of open source projects of Chinese technology companies remain unclear. To address this issue, we conduct a mixed-method study to investigate the landscape of projects open sourced by three large Chinese technology companies, namely Baidu, Alibaba, and Tencent (BAT). We study the categories and characteristics of open source projects, the developer's perceptions towards open sourcing effort for these companies, and the internationalization effort of their open source projects. We collected 1,000 open source projects that were open sourced by BAT in GitHub and performed an online survey that received 101 responses from developers of these

projects. Some key findings include: 1) BAT prefer to open source frontend development projects, 2) 88% of the respondents are positive towards open sourcing software projects in their respective companies, 3) 64% of the respondents reveal that the most common motivations for BAT to open source their projects are the desire to gain fame, expand their influence and gain recruitment advantage, 4) respondents believe that the most common internationalization effort is "providing an English version of readme files", 5) projects with more internationalization effort (i.e., include an English readme file) are more popular. Our findings provide directions for software engineering researchers and provide practical suggestions to software developers and Chinese technology companies.

[**Hanenberg2010**] Stefan Hanenberg. An experiment about static and dynamic type systems. In *Proc. International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM Press, 2010, DOI 10.1145/1869459.1869462.
Abstract: Although static type systems are an essential part in teach-ing and research in software engineering and computer science, there is hardly any knowledge about what the impact of static type systems on the development time or the resulting quality for a piece of software is. On the one hand there are authors that state that static type systems decrease an application's complexity and hence its development time (which means that the quality must be improved since developers have more time left in their projects). On the other hand there are authors that argue that static type systems increase development time (and hence decrease the code quality) since they restrict developers to express themselves in a desired way. This paper presents an empirical study with 49 subjects that studies the impact of a static type system for the development of a parser over 27 hours working time. In the experiments the existence of the static type system has neither a positive nor a negative impact on an application's development time (under the conditions of the experiment).

[**Hanenberg2013**] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, 19(5):1335–1382, 12 2013, DOI 10.1007/s10664-013-9289-1.
Abstract: Static type systems play an essential role in contemporary programming languages. Despite their importance, whether static type systems impact human software development capabilities remains open. One frequently mentioned argument in favor of static type systems is that they improve the maintainability of software systems—an often-used claim for which there is little empirical evidence. This paper describes an experiment that tests whether static type systems improve the maintainability of software systems, in terms of understanding undocumented code, fixing type errors, and fixing semantic errors. The results show rigorous empirical evidence that static types are indeed beneficial to these activities, except when fixing semantic errors. We further conduct an exploratory analysis of the

data in order to understand possible reasons for the effect of type systems on the three kinds of tasks used in this experiment. From the exploratory analysis, we conclude that developers using a dynamic type system tend to look at different files more frequently when doing programming tasks—which is a potential reason for the observed differences in time.

[**Hannay2010**] J.E. Hannay, E. Arisholm, H. Engvik, and D.I.K. Sjøberg. Effects of personality on pair programming. *IEEE Transactions on Software Engineering*, 36(1):61–80, 1 2010, DOI 10.1109/tse.2009.41.
Abstract: Personality tests in various guises are commonly used in recruitment and career counseling industries. Such tests have also been considered as instruments for predicting the job performance of software professionals both individually and in teams. However, research suggests that other human-related factors such as motivation, general mental ability, expertise, and task complexity also affect the performance in general. This paper reports on a study of the impact of the Big Five personality traits on the performance of pair programmers together with the impact of expertise and task complexity. The study involved 196 software professionals in three countries forming 98 pairs. The analysis consisted of a confirmatory part and an exploratory part. The results show that: (1) Our data do not confirm a meta-analysis-based model of the impact of certain personality traits on performance and (2) personality traits, in general, have modest predictive value on pair programming performance compared with expertise, task complexity, and country. We conclude that more effort should be spent on investigating other performance-related predictors such as expertise, and task complexity, as well as other promising predictors, such as programming skill and learning. We also conclude that effort should be spent on elaborating on the effects of personality on various measures of collaboration, which, in turn, may be used to predict and influence performance. Insights into such malleable, rather than static, factors may then be used to improve pair programming performance.

[**Harms2016**] Kyle James Harms, Jason Chen, and Caitlin L. Kelleher. Distractors in Parsons Problems decrease learning efficiency for young novice programmers. In *Proc. Conference on International Computing Education Research (ICER)*. ACM, 8 2016, DOI 10.1145/2960310.2960314.
Abstract: Parsons problems are an increasingly popular method for helping inexperienced programmers improve their programming skills. In Parsons problems, learners are given a set of programming statements that they must assemble into the correct order. Parsons problems commonly use distractors, extra statements that are not part of the solution. Yet, little is known about the effect distractors have on a learner's ability to acquire new programming skills. We present a study comparing the effectiveness of learning programming from Parsons problems with and without distractors. The results suggest that distractors decrease learning efficiency. We found that distractor participants showed no difference in transfer task performance compared to

those without distractors. However, the distractors increased learners cognitive load, decreased their success at completing Parsons problems by 26%, and increased learners' time on task by 14%.

[**Hata2019**] Hideaki Hata, Christoph Treude, Raula Gaikovina Kula, and Takashi Ishio. 9.6 million links in source code comments: purpose, evolution, and decay. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2019, DOI `10.1109/icse.2019.00123`.

Abstract: Links are an essential feature of the World Wide Web, and source code repositories are no exception. However, despite their many undisputed benefits, links can suffer from decay, insufficient versioning, and lack of bidirectional traceability. In this paper, we investigate the role of links contained in source code comments from these perspectives. We conducted a large-scale study of around 9.6 million links to establish their prevalence, and we used a mixed-methods approach to identify the links' targets, purposes, decay, and evolutionary aspects. We found that links are prevalent in source code repositories, that licenses, software homepages, and specifications are common types of link targets, and that links are often included to provide metadata or attribution. Links are rarely updated, but many link targets evolve. Almost 10% of the links included in source code comments are dead. We then submitted a batch of link-fixing pull requests to open source software repositories, resulting in most of our fixes being merged successfully. Our findings indicate that links in source code comments can indeed be fragile, and our work opens up avenues for future work to address these problems.

[**Hatton1994**] L. Hatton and A. Roberts. How accurate is scientific software? *IEEE Transactions on Software Engineering*, 20(10):785–797, 1994, DOI `10.1109/32.328993`.

Abstract: This paper describes some results of what, to the authors' knowledge, is the largest N-version programming experiment ever performed. The object of this ongoing four-year study is to attempt to determine just how consistent the results of scientific computation really are, and, from this, to estimate accuracy. The experiment is being carried out in a branch of the earth sciences known as seismic data processing, where 15 or so independently developed large commercial packages that implement mathematical algorithms from the same or similar published specifications in the same programming language (Fortran) have been developed over the last 20 years. The results of processing the same input dataset, using the same user-specified parameters, for nine of these packages is reported in this paper. Finally, feedback of obvious flaws was attempted to reduce the overall disagreement. The results are deeply disturbing. Whereas scientists like to think that their code is accurate to the precision of the arithmetic used, in this study, numerical disagreement grows at around the rate of 1% in average absolute difference per 4000 fines of implemented code, and, even worse, the nature of the disagreement is nonrandom. Furthermore, the seismic data processing industry has better than average quality standards for

its software development with both identifiable quality assurance functions and substantial test datasets.

[**Hatton1997**] L. Hatton. The t-experiments: errors in scientific software. In Ronald F. Boisvert, editor, *Quality of Numerical Software*, pages 12–31. Springer US, 1997, DOI `10.1007/978-1-5041-2940-4_2`.
Abstract: This paper covers two very large experiments carried out concurrently between 1990 and 1994, together known as the T-experiments. Experiment T1 had the objective of measuring the consistency of several million lines of scientific software written in C and Fortran 77 by static deep-flow analysis across many different industries and application areas, and experiment T2 had the objective of measuring the level of dynamic disagreement between independent implementations of the same algorithms acting on the same input data with the same parameters in just one of these industrial application areas. Experiment T1 showed that C and Fortran are riddled with statically detectable inconsistencies independent of the application area. For example, interface inconsistencies occur at the rate of one in every 7 interfaces on average in Fortran, and one in every 37 interfaces in C. They also show that Fortran components are typically 2.5 times bigger than C components, and that roughly 30% of the Fortran population and 10% of the C population would be deemed untestable by any standards. Experiment T2 was even more disturbing. Whereas scientists like to think that their results are accurate to the precision of the arithmetic used, in this study, the degree of agreement gradually degenerated from 6 significant figures to 1 significant figure during the computation. The reasons for this disagreement are laid squarely at the door of software failure, as other possible causes are considered and rejected. Taken with other evidence, these two experiments suggest that the results of scientific calculations involving significant amounts of software should be taken with several large pinches of salt.

[**Hayashi2019**] Junichi Hayashi, Yoshiki Higo, Shinsuke Matsumoto, and Shinji Kusumoto. Impacts of daylight saving time on software development. In *Proc. International Conference on Mining Software Repositories (MSR)*. IEEE, 5 2019, DOI `10.1109/msr.2019.00076`.
Abstract: Daylight saving time (DST) is observed in many countries and regions. DST is not considered on some software systems at the beginning of their developments, for example, software systems developed in regions where DST is not observed. However, such systems may have to consider DST at the requests of their users. Before now, there has been no study about the impacts of DST on software development. In this paper, we study the impacts of DST on software development by mining the repositories on GitHub. We analyze the date when the code related to DST is changed, and we analyze the regions where the developers applied the changes live. Furthermore, we classify the changes into some patterns.

[**Hazoom2021**] Moshe Hazoom, Vibhor Malik, and Ben Bogin. Text-to-SQL in the wild: A naturally-occurring dataset based on stack exchange data. In *Proc. Workshop on Natural Language Processing for Programming (NLP4Prog)*. Association for Computational Linguistics, 2021, DOI `10.18653/v1/2021.nlp4prog-1.9`.

Abstract: Most available semantic parsing datasets, comprising of pairs of natural utterances and logical forms, were collected solely for the purpose of training and evaluation of natural language understanding systems. As a result, they do not contain any of the richness and variety of natural-occurring utterances, where humans ask about data they need or are curious about. In this work, we release SEDE, a dataset with 12,023 pairs of utterances and SQL queries collected from real usage on the Stack Exchange website. We show that these pairs contain a variety of real-world challenges which were rarely reflected so far in any other semantic parsing dataset, propose an evaluation metric based on comparison of partial query clauses that is more suitable for real-world queries, and conduct experiments with strong baselines, showing a large gap between the performance on SEDE compared to other common datasets.

[**Hemmati2013**] Hadi Hemmati, Sarah Nadi, Olga Baysal, Oleksii Kononenko, Wei Wang, Reid Holmes, and Michael W. Godfrey. The MSR Cookbook: mining a decade of research. In *Proc. International Conference on Mining Software Repositories (MSR)*. IEEE, 5 2013, DOI `10.1109/msr.2013.6624048`.

Abstract: The Mining Software Repositories (MSR) research community has grown significantly since the first MSR workshop was held in 2004. As the community continues to broaden its scope and deepens its expertise, it is worthwhile to reflect on the best practices that our community has developed over the past decade of research. We identify these best practices by surveying past MSR conferences and workshops. To that end, we review all 117 full papers published in the MSR proceedings between 2004 and 2012. We extract 268 comments from these papers, and categorize them using a grounded theory methodology. From this evaluation, four high-level themes were identified: data acquisition and preparation, synthesis, analysis, and sharing/replication. Within each theme we identify several common recommendations, and also examine how these recommendations have evolved over the past decade. In an effort to make this survey a living artifact, we also provide a public forum that contains the extracted recommendations in the hopes that the MSR community can engage in a continuing discussion on our evolving best practices.

[**Hermans2011**] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Supporting professional spreadsheet users by generating leveled dataflow diagrams. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 5 2011, DOI `10.1145/1985793.1985855`.

Abstract: Thanks to their flexibility and intuitive programming model,

spreadsheets are widely used in industry, often for businesscritical applications. Similar to software developers, professional spreadsheet users demand support for maintaining and transferring their spreadsheets. In this paper, we first study the problems and information needs of professional spreadsheet users by means of a survey conducted at a large financial company. Based on these needs, we then present an approach that extracts this information from spreadsheets and presents it in a compact and easy to understand way, with leveled dataflow diagrams. Our approach comes with three different views on the dataflow that allow the user to analyze the dataflow diagrams in a top-down fashion. To evaluate the usefulness of the proposed approach, we conducted a series of interviews as well as nine case studies in an industrial setting. The results of the evaluation clearly indicate the demand for and usefulness of our approach in ease the understanding of spreadsheets.

[**Hermans2016**] Felienne Hermans and Efthimia Aivaloglou. Do code smells hamper novice programming? a controlled experiment on Scratch programs. In *Proc. International Conference on Program Comprehension (ICPC)*. IEEE, 5 2016, DOI `10.1109/icpc.2016.7503706`.
Abstract: Recently, block-based programming languages like Alice, Scratch and Blockly have become popular tools for programming education. There is substantial research showing that block-based languages are suitable for early programming education. But can block-based programs be smelly too? And does that matter to learners? In this paper we explore the code smells metaphor in the context of block-based programming language Scratch. We conduct a controlled experiment with 61 novice Scratch programmers, in which we divided the novices into three groups. One third receive a non-smelly program, while the other groups receive a program suffering from the Duplication or the Long Method smell respectively. All subjects then perform the same comprehension tasks on their program, after which we measure their time and correctness. The results of the experiment show that code smell indeed influence performance: subjects working on the program exhibiting code smells perform significantly worse, but the smells did not affect the time subjects needed. Investigating different types of tasks in more detail, we find that Long Method mainly decreases system understanding, while Duplication decreases the ease with which subjects modify Scratch programs.

[**Hermans2021**] Felienne Hermans. *The Programmer's Brain: What Every Programmer Needs to Know About Cognition*. Manning, 2021.
Abstract: Your brain responds in a predictable way when it encounters new or difficult tasks. This unique book teaches you concrete techniques rooted in cognitive science that will improve the way you learn and think about code.

[**Herraiz2010**] Israel Herraiz and Ahmed E. Hassan. Beyond lines of code: Do we need more complexity metrics? In Andy Oram and Greg Wilson, editors, *Making Software*. O'Reilly, 2010.

Abstract: Summarizes work on code complexity metrics and finds that there is little evidence any of them provide more information than simply counting lines of code.

[**Herzig2013**] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2013, DOI `10.1109/icse.2013.6606585`.
Abstract: In a manual examination of more than 7,000 issue reports from the bug databases of five open-source projects, we found 33.8% of all bug reports to be misclassified - that is, rather than referring to a code fix, they resulted in a new feature, an update to documentation, or an internal refactoring. This misclassification introduces bias in bug prediction models, confusing bugs and features: On average, 39% of files marked as defective actually never had a bug. We discuss the impact of this misclassification on earlier studies and recommend manual data validation for future studies.

[**Hindle2012**] Abram Hindle, Christian Bird, Thomas Zimmermann, and Nachiappan Nagappan. Relating requirements to implementation via topic analysis: do topics extracted from requirements make sense to managers and developers? In *Proc. International Conference on Software Maintenance (ICSM)*. IEEE, 9 2012, DOI `10.1109/icsm.2012.6405278`.
Abstract: Large organizations like Microsoft tend to rely on formal requirements documentation in order to specify and design the software products that they develop. These documents are meant to be tightly coupled with the actual implementation of the features they describe. In this paper we evaluate the value of high-level topic-based requirements traceability in the version control system, using Latent Dirichlet Allocation (LDA). We evaluate LDA topics on practitioners and check if the topics and trends extracted matches the perception that Program Managers and Developers have about the effort put into addressing certain topics. We found that effort extracted from version control that was relevant to a topic often matched the perception of the managers and developers of what occurred at the time. Furthermore we found evidence that many of the identified topics made sense to practitioners and matched their perception of what occurred. But for some topics, we found that practitioners had difficulty interpreting and labelling them. In summary, we investigate the high-level traceability of requirements topics to version control commits via topic analysis and validate with the actual stakeholders the relevance of these topics extracted from requirements.

[**Hindle2016**] Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 4 2016, DOI `10.1145/2902362`.
Abstract: Natural languages like English are rich, complex, and powerful. The highly creative and graceful use of languages like English and Tamil, by masters like Shakespeare and Avvaiyar, can certainly delight and inspire. But in practice, given cognitive constraints and the exigencies of daily

life, most human utterances are far simpler and much more repetitive and predictable. In fact, these utterances can be very usefully modeled using modern statistical methods. This fact has led to the phenomenal success of statistical approaches to speech recognition, natural language translation, question-answering, and text mining and comprehension. We begin with the conjecture that most software is also natural, in the sense that it is created by humans at work, with all the attendant constraints and limitations - and thus, like natural language, it is also likely to be repetitive and predictable. We then proceed to ask whether a) code can be usefully modeled by statistical language models and b) such models can be leveraged to support software engineers. Using the widely adopted n-gram model, we provide empirical evidence supportive of a positive answer to both these questions. We show that code is also very repetitive, and in fact even more so than natural languages. As an example use of the model, we have developed a simple code completion engine for Java that, despite its simplicity, already improves Eclipse's built-in completion capability. We conclude the paper by laying out a vision for future research in this area.

[**Hoda2021**] Rashina Hoda. Socio-technical grounded theory for software engineering. *IEEE Transactions on Software Engineering*, pages 1–1, 2021, DOI 10.1109/tse.2021.3106280.
Abstract: Grounded Theory (GT), a sociological research method designed to study social phenomena, is increasingly being used to investigate the human and social aspects of software engineering (SE). However, being written by and for sociologists, GT is often challenging for a majority of SE researchers to understand and apply. Additionally, SE researchers attempting ad hoc adaptations of traditional GT guidelines for modern socio-technical (ST) contexts often struggle in the absence of clear and relevant guidelines to do so, resulting in poor quality studies. To overcome these research community challenges and leverage modern research opportunities, this paper presents Socio-Technical Grounded Theory (STGT) designed to ease application and achieve quality outcomes. It defines what exactly is meant by an ST research context and presents the STGT guidelines that expand GT's philosophical foundations, provide increased clarity and flexibility in its methodological steps and procedures, define possible scope and contexts of application, encourage frequent reporting of a variety of interim, preliminary, and mature outcomes, and introduce nuanced evaluation guidelines for different outcomes. It is hoped that the SE research community and related ST disciplines such as computer science, data science, artificial intelligence, information systems, human computer/robot/AI interaction, human-centered emerging technologies (and increasingly other disciplines being transformed by rapid digitalisation and AI-based augmentation), will benefit from applying STGT to conduct quality research studies and systematically produce rich findings and mature theories with confidence.

[**Hofmeister2017**] Johannes Hofmeister, Janet Siegmund, and Daniel V. Holt. Shorter identifier names take longer to comprehend. In *Proc. International*

Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2 2017, DOI `10.1109/saner.2017.7884623`.

Abstract: Developers spend the majority of their time comprehending code, a process in which identifier names play a key role. Although many identifier naming styles exist, they often lack an empirical basis and it is not quite clear whether short or long identifier names facilitate comprehension. In this paper, we investigate the effect of different identifier naming styles (letters, abbreviations, words) on program comprehension, and whether these effects arise because of their length or their semantics. We conducted an experimental study with 72 professional C# developers, who looked for defects in source-code snippets. We used a within-subjects design, such that each developer saw all three versions of identifier naming styles and we measured the time it took them to find a defect. We found that words lead to, on average, 19% faster comprehension speed compared to letters and abbreviations, but we did not find a significant difference in speed between letters and abbreviations. The results of our study suggest that defects in code are more difficult to detect when code contains only letters and abbreviations. Words as identifier names facilitate program comprehension and can help to save costs and improve software quality.

[**Holmes2020**] Josie Holmes, Iftekhar Ahmed, Caius Brindescu, Rahul Gopinath, He Zhang, and Alex Groce. Using relative lines of code to guide automated test generation for python. *ACM Transactions on Software Engineering and Methodology*, 29(4):1–38, 10 2020, DOI `10.1145/3408896`.

Abstract: Raw lines of code (LOC) is a metric that does not, at first glance, seem extremely useful for automated test generation. It is both highly language-dependent and not extremely meaningful, semantically, within a language: one coder can produce the same effect with many fewer lines than another. However, relative LOC, between components of the same project, turns out to be a highly useful metric for automated testing. In this article, we make use of a heuristic based on LOC counts for tested functions to dramatically improve the effectiveness of automated test generation. This approach is particularly valuable in languages where collecting code coverage data to guide testing has a very high overhead. We apply the heuristic to property-based Python testing using the TSTL (Template Scripting Testing Language) tool. In our experiments, the simple LOC heuristic can improve branch and statement coverage by large margins (often more than 20%, up to 40% or more) and improve fault detection by an even larger margin (usually more than 75% and up to 400% or more). The LOC heuristic is also easy to combine with other approaches and is comparable to, and possibly more effective than, two well-established approaches for guiding random testing.

[**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.

[**Hora2021b**] Andre Hora. What code is deliberately excluded from test coverage and why? In *Proc. International Conference on Mining Software Repositories (MSR)*. IEEE, 5 2021, DOI `10.1109/msr52588.2021.00051`.
Abstract: Test coverage is largely used to assess test effectiveness. In practice, not all code is equally important for coverage analysis, for instance, code that will not be executed during tests is irrelevant and can actually harm the analysis. Some coverage tools provide support for code exclusion from coverage reports, however, we are not yet aware of what code tends to be excluded nor the reasons behind it. This can support the creation of more accurate coverage reports and reveal novel and harmful usage cases. In this paper, we provide the first empirical study to understand code exclusion practices in test coverage. We mine 55 Python projects and assess commit messages and code comments to detect rationales for exclusions. We find that (1) over 1/3 of the projects perform deliberate coverage exclusion; (2) 75% of the code are already created using the exclusion feature, while 25% add it over time; (3) developers exclude non-runnable, debug-only, and defensive code, but also platform-specific and conditional importing; and (4) most code is excluded because it is already untested, low-level, or complex. Finally, we discuss implications to improve coverage analysis and shed light on the existence of biased coverage reports.

[**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.

[**Huijgens2020**] Hennie Huijgens, Ayushi Rastogi, Ernst Mulders, Georgios Gousios, and Arie van Deursen. Questions for data scientists in software engineering: a replication. In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 11 2020, DOI 10.1145/3368089.3409717.
Abstract: In 2014, a Microsoft study investigated the sort of questions that data science applied to software engineering should answer. This resulted in

145 questions that developers considered relevant for data scientists to answer, thus providing a research agenda to the community. Fast forward to five years, no further studies investigated whether the questions from the software engineers at Microsoft hold for other software companies, including software-intensive companies with different primary focus (to which we refer as software-defined enterprises). Furthermore, it is not evident that the problems identified five years ago are still applicable, given the technological advances in software engineering. This paper presents a study at ING, a software-defined enterprise in banking in which over 15,000 IT staff provides in-house software solutions. This paper presents a comprehensive guide of questions for data scientists selected from the previous study at Microsoft along with our current work at ING. We replicated the original Microsoft study at ING, looking for questions that impact both software companies and software-defined enterprises and continue to impact software engineering. We also add new questions that emerged from differences in the context of the two companies and the five years gap in between. Our results show that software engineering questions for data scientists in the software-defined enterprise are largely similar to the software company, albeit with exceptions. We hope that the software engineering research community builds on the new list of questions to create a useful body of knowledge.

[**Hundhausen2011**] Christopher D. Hundhausen, Pawan Agarwal, and Michael Trevisan. Online vs. face-to-face pedagogical code reviews. In *Proc. Technical Symposium on Computer Science Education (SIGCSE)*. ACM Press, 2011, DOI `10.1145/1953163.1953201`.
Abstract: Given the increased importance of communication, teamwork, and critical thinking skills in the computing profession, we have been exploring studio-based instructional methods, in which students develop solutions and iteratively refine them through critical review by their peers and instructor. We have developed an adaptation of studio-based instruction for computing education called the pedagogical code review (PCR), which is modeled after the code inspection process used in the software industry. Unfortunately, PCRs are time-intensive, making them difficult to implement within a typical computing course. To address this issue, we have developed an online environment that allows PCRs to take place asynchronously outside of class. We conducted an empirical study that compared a CS 1 course with online PCRs against a CS 1 course with face-to-face PCRs. Our study had three key results: (a) in the course with face-to-face PCRs, student attitudes with respect to self-efficacy and peer learning were significantly higher; (b) in the course with face-to-face PCRs, students identified more substantive issues in their reviews; and (c) in the course with face-to-face PCRs, students were generally more positive about the value of PCRs. In light of our findings, we recommend specific ways online PCRs can be better designed.

[**Imam2021**] Ahmed Imam and Tapajit Dey. Tracking hackathon code creation and reuse. In *Proc. International Conference on Mining Software Reposito-*

ries (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.

[**Inozemtseva2014**] Laura Inozemtseva and Reid Holmes.   Coverage is not strongly correlated with test suite effectiveness.   In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 5 2014, DOI 10.1145/2568225.2568271.
Abstract: The coverage of a test suite is often used as a proxy for its ability to detect faults. However, previous studies that investigated the correlation between code coverage and test suite effectiveness have failed to reach a consensus about the nature and strength of the relationship between these test suite characteristics. Moreover, many of the studies were done with small or synthetic programs, making it unclear whether their results generalize to larger programs, and some of the studies did not account for the confounding influence of test suite size. In addition, most of the studies were done with adequate suites, which are are rare in practice, so the results may not generalize to typical test suites. We have extended these studies by evaluating the relationship between test suite size, coverage, and effectiveness for large Java programs. Our study is the largest to date in the literature: we generated 31,000 test suites for five systems consisting of up to 724,000 lines of source code. We measured the statement coverage, decision coverage, and modified condition coverage of these suites and used mutation testing to evaluate their fault detection effectiveness. We found that there is a low to moderate correlation between coverage and effectiveness when the number of test cases

in the suite is controlled for. In addition, we found that stronger forms of coverage do not provide greater insight into the effectiveness of the suite. Our results suggest that coverage, while useful for identifying under-tested parts of a program, should not be used as a quality target because it is not a good indicator of test suite effectiveness.

[**Jacobson2013**] Ivar Jacobson, Pan-Wei Ng, Paul E. McMahon, Ian Spence, and Svante Lidman. *The Essence of Software Engineering: Applying the SEMAT Kernel*. Addison-Wesley Professional, 2013.
Abstract: SEMAT (Software Engineering Methods and Theory) is an international initiative designed to identify a common ground, or universal standard, for software engineering. It is supported by some of the most distinguished contributors to the field. Creating a simple language to describe methods and practices, the SEMAT team expresses this common ground as a kernel—or framework—of elements essential to all software development. The Essence of Software Engineering introduces this kernel and shows how to apply it when developing software and improving a team's way of working. It is a book for software professionals, not methodologists. Its usefulness to development team members, who need to evaluate and choose the best practices for their work, goes well beyond the description or application of any single method.

[**Jalote2021**] Pankaj Jalote and Damodaram Kamma. Studying task processes for improving programmer productivity. *IEEE Transactions on Software Engineering*, 47(4):801–817, 4 2021, DOI 10.1109/tse.2019.2904230.
Abstract: Productivity of a software development organization can be enhanced by improving the software process, using better tools/technology, and enhancing the productivity of programmers. This work focuses on improving programmer productivity by studying the process used by a programmer for executing an assigned task, which we call the task process. We propose a general framework for studying the impact of task processes on programmer productivity and also the impact of transferring task processes of high-productivity programmers to average-productivity peers. We applied the framework to a few live projects in Robert Bosch Engineering and Business Solutions Limited, a CMMI Level 5 company. In each project, we identified two groups of programmers: high-productivity and average-productivity programmers. We requested each programmer to video capture their computer screen while executing his/her assigned tasks. We then analyzed these task videos to extract the task processes and then used them to identify the differences between the task processes used by the two groups. Some key differences were found between the task processes, which could account for the difference in productivities of the two groups. Similarities between the task processes were also analyzed quantitatively by modeling each task process as a Markov chain. We found that programmers from the same group used similar task processes, but the task processes of the two groups differed considerably. The task processes of high-productivity programmers were transferred to the average-productivity programmers by training them

on the key steps missing in their process but commonly present in the work of their high-productivity peers. A substantial productivity gain was found in the average-productivity programmers as a result of this transfer. The study shows that task processes of programmers impact their productivity, and it is possible to improve the productivity of average-productivity programmers by transferring task processes from high-productivity programmers to them.

[**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.

[**Jones2020**] Derek M. Jones. *Evidence-based Software Engineering: based on the publicly available data*. Knowledge Software, Ltd., 11 2020.
Abstract: This book discusses what is currently known about software engineering, based on an analysis of all the publicly available data. This aim is not as ambitious as it sounds, because there is not a great deal of data publicly available. The intent is to provide material that is useful to professional developers working in industry; until recently researchers in software engineering have been more interested in vanity work, promoted by ego and bluster. The material is organized in two parts, the first covering software engineering and the second the statistics likely to be needed for the analysis of software engineering data.

[**Jorgensen2011**] Magne Jørgensen and Stein Grimstad. the impact of irrelevant and misleading information on software development effort estimates: a randomized controlled field experiment. *IEEE Transactions on Software Engineering*, 37(5):695–707, 9 2011, DOI 10.1109/tse.2010.78.
Abstract: Studies in laboratory settings report that software development effort estimates can be strongly affected by effort-irrelevant and misleading information. To increase our knowledge about the importance of these effects in field settings, we paid 46 outsourcing companies from various countries to estimate the required effort of the same five software development projects. The companies were allocated randomly to either the original requirement specification or a manipulated version of the original requirement specification. The manipulations were as follows: 1) reduced length of requirement specification with no change of content, 2) information about the low effort spent on the development of the old system to be replaced, 3) information about the client's unrealistic expectations about low cost, and 4) a restriction of a short development period with start up a few months ahead. We found that the effect sizes in the field settings were much smaller than those found for similar manipulations in laboratory settings. Our findings suggest that we should be careful about generalizing to field settings the effect sizes found in laboratory settings. While laboratory settings can be useful to demonstrate the existence of an effect and better understand it, field studies may be needed to study the size and importance of these effects.

[**Jorgensen2012**] Magne Jørgensen and Stein Grimstad. Software development estimation biases: the role of interdependence. *IEEE Transactions on Software Engineering*, 38(3):677–693, 5 2012, DOI 10.1109/tse.2011.40.

Abstract: Software development effort estimates are frequently too low, which may lead to poor project plans and project failures. One reason for this bias seems to be that the effort estimates produced by software developers are affected by information that has no relevance for the actual use of effort. We attempted to acquire a better understanding of the underlying mechanisms and the robustness of this type of estimation bias. For this purpose, we hired 374 software developers working in outsourcing companies to participate in a set of three experiments. The experiments examined the connection between estimation bias and developer dimensions: self-construal (how one sees oneself), thinking style, nationality, experience, skill, education, sex, and organizational role. We found that estimation bias was present along most of the studied dimensions. The most interesting finding may be that the estimation bias increased significantly with higher levels of interdependence, i.e., with stronger emphasis on connectedness, social context, and relationships. We propose that this connection may be enabled by an activation of one's self-construal when engaging in effort estimation, and a connection between a more interdependent self-construal and increased search for indirect messages, lower ability to ignore irrelevant context, and a stronger emphasis on socially desirable responses.

[**Kamienski2021**] Arthur V. Kamienski, Luisa Palechor, Cor-Paul Bezemer, and Abram Hindle. PySStuBs: Characterizing single-statement bugs in popular open-source python projects. In *Proc. International Conference on Mining Software Repositories (MSR)*. IEEE, 5 2021, DOI `10.1109/msr52588.2021.00066`.
Abstract: Single-statement bugs (SStuBs) can have a severe impact on developer productivity. Despite usually being simple and not offering much of a challenge to fix, these bugs may still disturb a developer's workflow and waste precious development time. However, few studies have paid attention to these simple bugs, focusing instead on bugs of any size and complexity. In this study, we explore the occurrence of SStuBs in some of the most popular open-source Python projects on GitHub, while also characterizing their patterns and distribution. We further compare these bugs to SStuBs found in a previous study on Java Maven projects. We find that these Python projects have different SStuB patterns than the ones in Java Maven projects and identify 7 new SStuB patterns. Our results may help uncover the importance of understanding these bugs for the Python programming language, and how developers can handle them more effectively.

[**KanatAlexander2012**] Max Kanat-Alexander. *Code Simplicity: The Science of Software Development.* O'Reilly, 2012.
Abstract: Good software development results in simple code. Unfortunately, much of the code existing in the world today is far too complex. This concise guide helps you understand the fundamentals of good software development through universal laws—principles you can apply to any programming language or project from here to eternity.

[**Kapser2008**] Cory J. Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 7 2008, DOI `10.1007/s10664-008-9076-6`.

Abstract: Literature on the topic of code cloning often asserts that duplicating code within a software system is a bad practice, that it causes harm to the system's design and should be avoided. However, in our studies, we have found significant evidence that cloning is often used in a variety of ways as a principled engineering tool. For example, one way to evaluate possible new features for a system is to clone the affected subsystems and introduce the new features there, in a kind of sandbox testbed. As features mature and become stable within the experimental subsystems, they can be migrated incrementally into the stable code base; in this way, the risk of introducing instabilities in the stable version is minimized. This paper describes several patterns of cloning that we have observed in our case studies and discusses the advantages and disadvantages associated with using them. We also examine through a case study the frequencies of these clones in two medium-sized open source software systems, the Apache web server and the Gnumeric spreadsheet application. In this study, we found that as many as 71% of the clones could be considered to have a positive impact on the maintainability of the software system.

[**Kasi2013**] Bakhtiar Khan Kasi and Anita Sarma. Cassandra: proactive conflict minimization through optimized task scheduling. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2013, DOI `10.1109/icse.2013.6606619`.

Abstract: Software conflicts arising because of conflicting changes are a regular occurrence and delay projects. The main precept of workspace awareness tools has been to identify potential conflicts early, while changes are still small and easier to resolve. However, in this approach conflicts still occur and require developer time and effort to resolve. We present a novel conflict minimization technique that proactively identifies potential conflicts, encodes them as constraints, and solves the constraint space to recommend a set of conflict-minimal development paths for the team. Here we present a study of four open source projects to characterize the distribution of conflicts and their resolution efforts. We then explain our conflict minimization technique and the design and implementation of this technique in our prototype, Cassandra. We show that Cassandra would have successfully avoided a majority of conflicts in the four open source test subjects. We demonstrate the efficiency of our approach by applying the technique to a simulated set of scenarios with higher than normal incidence of conflicts.

[**Kavaler2019**] David Kavaler, Asher Trockman, Bogdan Vasilescu, and Vladimir Filkov. Tool choice matters: JavaScript quality assurance tools and usage outcomes in GitHub projects. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2019, DOI `10.1109/icse.2019.00060`.

Abstract: Quality assurance automation is essential in modern software development. In practice, this automation is supported by a multitude of tools that fit different needs and require developers to make decisions about which tool to choose in a given context. Data and analytics of the pros and cons can inform these decisions. Yet, in most cases, there is a dearth of empirical evidence on the effectiveness of existing practices and tool choices. We propose a general methodology to model the time-dependent effect of automation tool choice on four outcomes of interest: prevalence of issues, code churn, number of pull requests, and number of contributors, all with a multitude of controls. On a large data set of npm JavaScript projects, we extract the adoption events for popular tools in three task classes: linters, dependency managers, and coverage reporters. Using mixed methods approaches, we study the reasons for the adoptions and compare the adoption effects within each class, and sequential tool adoptions across classes. We find that some tools within each group are associated with more beneficial outcomes than others, providing an empirical perspective for the benefits of each. We also find that the order in which some tools are implemented is associated with varying outcomes.

[**Kernighan1979**] Brian W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. McGraw-Hill, 2nd edition, 1979.
Abstract: Lays out several dozen rules for good programming style; while examples are in FORTRAN, the rules apply to almost every language.

[**Kernighan1981**] Brian W. Kernighan and P. J. Plauger. *Software Tools in Pascal*. Addison-Wesley Professional, 1981.
Abstract: Shows readers how to build simple version of the the programming tools they use themselves, and in doing so shows how to think about software design.

[**Kernighan1983**] Brian W. Kernighan and Rob Pike. *The Unix Programming Environment*. Prentice-Hall, 1983.
Abstract: Explains the Unix "lots of little tools, easily recombined" approach to computing with lots of examples.

[**Khomh2012**] Foutse Khomh, Tejinder Dhaliwal, Ying Zou, and Bram Adams. Do faster releases improve software quality? an empirical case study of Mozilla Firefox. In *Proc. International Conference on Mining Software Repositories (MSR)*. IEEE, 6 2012, DOI 10.1109/msr.2012.6224279.
Abstract: Nowadays, many software companies are shifting from the traditional 18-month release cycle to shorter release cycles. For example, Google Chrome and Mozilla Firefox release new versions every 6 weeks. These shorter release cycles reduce the users' waiting time for a new release and offer better marketing opportunities to companies, but it is unclear if the quality of the software product improves as well, since shorter release cycles result in shorter testing periods. In this paper, we empirically study the development process of Mozilla Firefox in 2010 and 2011, a period during

which the project transitioned to a shorter release cycle. We compare crash rates, median uptime, and the proportion of post-release bugs of the versions that had a shorter release cycle with those having a traditional release cycle, to assess the relation between release cycle length and the software quality observed by the end user. We found that (1) with shorter release cycles, users do not experience significantly more post-release bugs and (2) bugs are fixed faster, yet (3) users experience these bugs earlier during software execution (the program crashes earlier).

[**Kiefer2015**] Marc Kiefer, Daniel Warzel, and Walter F. Tichy. An empirical study on parallelism in modern open-source projects. In *Proc. International Workshop on Software Engineering for Parallel Systems (SEPS)*. ACM, 10 2015, DOI `10.1145/2837476.2837481`.

Abstract: Writing parallel programs is hard, especially for inexperienced programmers. Parallel language features are still being added on a regular basis to most modern object-oriented languages and this trend is likely to continue. Being able to support developers with tools for writing and optimizing parallel programs requires a deep understanding of how programmers approach and implement parallelism. We present an empirical study of 135 parallel open-source projects in Java, C# and C++ ranging from small (¡ 1000 lines of code) to very large (¿ 2M lines of code) codebases. We examine the projects to find out how language features, synchronization mechanisms, parallel data structures and libraries are used by developers to express parallelism. We also determine which common parallel patterns are used and how the implemented solutions compare to typical textbook advice. The results show that similar parallel constructs are used equally often across languages, but usage also heavily depends on how easy to use a certain language feature is. Patterns that do not map well to a language are much rarer compared to other languages. Bad practices are prevalent in hobby projects but also occur in larger projects.

[**Kim2013**] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2013, DOI `10.1109/icse.2013.6606626`.

Abstract: Patch generation is an essential software maintenance task because most software systems inevitably have bugs that need to be fixed. Unfortunately, human resources are often insufficient to fix all reported and known bugs. To address this issue, several automated patch generation techniques have been proposed. In particular, a genetic-programming-based patch generation technique, GenProg, proposed by Weimer et al., has shown promising results. However, these techniques can generate nonsensical patches due to the randomness of their mutation operations. To address this limitation, we propose a novel patch generation approach, Pattern-based Automatic program Repair (Par), using fix patterns learned from existing human-written patches. We manually inspected more than 60,000 human-written patches and found there are several common fix patterns. Our ap-

proach leverages these fix patterns to generate program patches automatically. We experimentally evaluated Par on 119 real bugs. In addition, a user study involving 89 students and 164 developers confirmed that patches generated by our approach are more acceptable than those generated by GenProg. Par successfully generated patches for 27 out of 119 bugs, while GenProg was successful for only 16 bugs.

[**Kim2016**] Dohyeong Kim, Yonghwi Kwon, Peng Liu, I. Luk Kim, David Mitchel Perry, Xiangyu Zhang, and Gustavo Rodriguez-Rivera. Apex: automatic programming assignment error explanation. In *Proc. International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, 10 2016, DOI 10.1145/2983990.2984031.
Abstract: This paper presents Apex, a system that can automatically generate explanations for programming assignment bugs, regarding where the bugs are and how the root causes led to the runtime failures. It works by comparing the passing execution of a correct implementation (provided by the instructor) and the failing execution of the buggy implementation (submitted by the student). The technique overcomes a number of technical challenges caused by syntactic and semantic differences of the two implementations. It collects the symbolic traces of the executions and matches assignment statements in the two execution traces by reasoning about symbolic equivalence. It then matches predicates by aligning the control dependences of the matched assignment statements, avoiding direct matching of path conditions which are usually quite different. Our evaluation shows that Apex is every effective for 205 buggy real world student submissions of 4 programming assignments, and a set of 15 programming assignment type of buggy programs collected from stackoverflow.com, precisely pinpointing the root causes and capturing the causality for 94.5% of them. The evaluation on a standard benchmark set with over 700 student bugs shows similar results. A user study in the classroom shows that Apex has substantially improved student productivity.

[**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.

[**Kinshumann2011**] Kinshuman Kinshumann, Kirk Glerum, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging in the (very) large: ten years of implementation and experience. *Communications of the ACM*, 54(7):111–116, 7 2011, DOI 10.1145/1965724.1965749.
Abstract: Windows Error Reporting (WER) is a distributed system that automates the processing of error reports coming from an installed base of a billion machines. WER has collected billions of error reports in 10 years of operation. It collects error data automatically and classifies errors into buckets, which are used to prioritize developer effort and report fixes to users. WER uses a progressive approach to data collection, which minimizes overhead for most reports yet allows developers to collect detailed information when needed. WER takes advantage of its scale to use error statistics as a tool in debugging; this allows developers to isolate bugs that cannot be found at smaller scale. WER has been designed for efficient operation at large scale: one pair of database servers records all the errors that occur on all Windows computers worldwide.

[**Kinsman2021**] Timothy Kinsman, Mairieli Wessel, Marco A. Gerosa, and Christoph Treude. How do software developers use GitHub actions to automate their workflows? In *Proc. International Conference on Mining Software Repositories (MSR)*. IEEE, 5 2021, DOI 10.1109/msr52588.2021.00054.
Abstract: Automated tools are frequently used in social coding repositories to perform repetitive activities that are part of the distributed software development process. Recently, GitHub introduced GitHub Actions, a feature providing automated work-flows for repository maintainers. Although several Actions have been built and used by practitioners, relatively little has been done to evaluate them. Understanding and anticipating the effects of adopting such kind of technology is important for planning and management. Our research is the first to investigate how developers use Actions and how several activity indicators change after their adoption. Our results indicate that, although only a small subset of repositories adopted GitHub Actions to date, there is a positive perception of the technology. Our findings also indicate that the adoption of GitHub Actions increases the number of monthly rejected pull requests and decreases the monthly number of commits on

merged pull requests. These results are especially relevant for practitioners to understand and prevent undesirable effects on their projects.

[**Klotins2021**] Eriks Klotins, Michael Unterkalmsteiner, Panagiota Chatzipetrou, Tony Gorschek, Rafael Prikladnicki, Nirnaya Tripathi, and Leandro Bento Pompermaier. A progression model of software engineering goals, challenges, and practices in start-ups. *IEEE Transactions on Software Engineering*, 47(3):498–521, 3 2021, DOI `10.1109/tse.2019.2900213`.
Abstract: Context: Software start-ups are emerging as suppliers of innovation and software-intensive products. However, traditional software engineering practices are not evaluated in the context, nor adopted to goals and challenges of start-ups. As a result, there is insufficient support for software engineering in the start-up context. Objective: We aim to collect data related to engineering goals, challenges, and practices in start-up companies to ascertain trends and patterns characterizing engineering work in start-ups. Such data allows researchers to understand better how goals and challenges are related to practices. This understanding can then inform future studies aimed at designing solutions addressing those goals and challenges. Besides, these trends and patterns can be useful for practitioners to make more informed decisions in their engineering practice. Method: We use a case survey method to gather first-hand, in-depth experiences from a large sample of software start-ups. We use open coding and cross-case analysis to describe and identify patterns, and corroborate the findings with statistical analysis. Results: We analyze 84 start-up cases and identify 16 goals, 9 challenges, and 16 engineering practices that are common among start-ups. We have mapped these goals, challenges, and practices to start-up life-cycle stages (inception, stabilization, growth, and maturity). Thus, creating the progression model guiding software engineering efforts in start-ups. Conclusions: We conclude that start-ups to a large extent face the same challenges and use the same practices as established companies. However, the primary software engineering challenge in start-ups is to evolve multiple process areas at once, with a little margin for serious errors.

[**Kocaguneli2012**] Ekrem Kocaguneli, Tim Menzies, and Jacky W. Keung. On the value of ensemble effort estimation. *IEEE Transactions on Software Engineering*, 38(6):1403–1416, 11 2012, DOI `10.1109/tse.2011.111`.
Abstract: Background: Despite decades of research, there is no consensus on which software effort estimation methods produce the most accurate models. Aim: Prior work has reported that, given M estimation methods, no single method consistently outperforms all others. Perhaps rather than recommending one estimation method as best, it is wiser to generate estimates from ensembles of multiple estimation methods. Method: Nine learners were combined with 10 preprocessing options to generate $9 \times 10 = 90$ solo methods. These were applied to 20 datasets and evaluated using seven error measures. This identified the best n (in our case n = 13) solo methods that showed stable performance across multiple datasets and error measures. The top 2,

4, 8, and 13 solo methods were then combined to generate 12 multimethods, which were then compared to the solo methods. Results: 1) The top 10 (out of 12) multimethods significantly outperformed all 90 solo methods. 2) The error rates of the multimethods were significantly less than the solo methods. 3) The ranking of the best multimethod was remarkably stable. Conclusion: While there is no best single effort estimation method, there exist best combinations of such effort estimation methods.

[**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.

[**Kosar2011**] Tomaž Kosar, Marjan Mernik, and Jeffrey C. Carver. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical Software Engineering*, 17(3):276–304, 8 2011, DOI `10.1007/s10664-011-9172-x`.

Abstract: Domain-specific languages (DSLs) are often argued to have a simpler notation than general-purpose languages (GPLs), since the notation

is adapted to the specific problem domain. Consequently, the impact of domain relevance on the creation of the problem representation is believed to improve programmers' efficiency and accuracy when using DSLs compared with using similar solutions like application libraries in GPLs. Most of the common beliefs have been based upon qualitative conclusions drawn by developers. Rather than implementing the same problem in a DSL and in a GPL and comparing the efficiency and accuracy of each approach, developers often compare the implementation of a new program in a DSL to their previous experiences implementing similar programs in GPLs. Such a conclusion may or may not be valid. This paper takes a more skeptical approach to acceptance of those beliefs. By reporting on a family of three empirical studies comparing DSLs and GPLs in different domains. The results of the studies showed that when using a DSL, developers are more accurate and more efficient in program comprehension than when using a GPL. These results validate some of the long-held beliefs of the DSL community that until now were only supported by anecdotal evidence.

[**Kosar2018**] Tomaž Kosar, Sašo Gaberc, Jeffrey C. Carver, and Marjan Mernik. Program comprehension of domain-specific and general-purpose languages: replication of a family of experiments using integrated development environments. *Empirical Software Engineering*, 23(5):2734–2763, 2 2018, DOI 10.1007/s10664-017-9593-2.
Abstract: Domain-specific languages (DSLs) allow developers to write code at a higher level of abstraction compared with general-purpose languages (GPLs). Developers often use DSLs to reduce the complexity of GPLs. Our previous study found that developers performed program comprehension tasks more accurately and efficiently with DSLs than with corresponding APIs in GPLs. This study replicates our previous study to validate and extend the results when developers use IDEs to perform program comprehension tasks. We performed a dependent replication of a family of experiments. We made two specific changes to the original study: (1) participants used IDEs to perform the program comprehension tasks, to address a threat to validity in the original experiment and (2) each participant performed program comprehension tasks on either DSLs or GPLs, not both as in the original experiment. The results of the replication are consistent with and expanded the results of the original study. Developers are significantly more effective and efficient in tool-based program comprehension when using a DSL than when using a corresponding API in a GPL. The results indicate that, where a DSL is available, developers will perform program comprehension better using the DSL than when using the corresponding API in a GPL.

[**Krein2016**] Jonathan L. Krein, Lutz Prechelt, Natalia Juristo, Aziz Nanthaamornphong, Jeffrey C. Carver, Sira Vegas, Charles D. Knutson, Kevin D. Seppi, and Dennis L. Eggett. A multi-site joint replication of a design patterns experiment using moderator variables to generalize across contexts. *IEEE Transactions on Software Engineering*, 42(4):302–321, 4 2016, DOI 10.1109/tse.2015.2488625.

Abstract: Context. Several empirical studies have explored the benefits of software design patterns, but their collective results are highly inconsistent. Resolving the inconsistencies requires investigating moderators—i.e., variables that cause an effect to differ across contexts. Objectives. Replicate a design patterns experiment at multiple sites and identify sufficient moderators to generalize the results across prior studies. Methods. We perform a close replication of an experiment investigating the impact (in terms of time and quality) of design patterns (Decorator and Abstract Factory) on software maintenance. The experiment was replicated once previously, with divergent results. We execute our replication at four universities—spanning two continents and three countries—using a new method for performing distributed replications based on closely coordinated, small-scale instances ("joint replication"). We perform two analyses: 1) a post-hoc analysis of moderators, based on frequentist and Bayesian statistics; 2) an a priori analysis of the original hypotheses, based on frequentist statistics. Results. The main effect differs across the previous instances of the experiment and across the sites in our distributed replication. Our analysis of moderators (including developer experience and pattern knowledge) resolves the differences sufficiently to allow for cross-context (and cross-study) conclusions. The final conclusions represent 126 participants from five universities and 12 software companies, spanning two continents and at least four countries. Conclusions. The Decorator pattern is found to be preferable to a simpler solution during maintenance, as long as the developer has at least some prior knowledge of the pattern. For Abstract Factory, the simpler solution is found to be mostly equivalent to the pattern solution. Abstract Factory is shown to require a higher level of knowledge and/or experience than Decorator for the pattern to be beneficial.

[**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.

[**Lampel2021**] Johannes Lampel, Sascha Just, Sven Apel, and Andreas Zeller. When life gives you oranges: detecting and diagnosing intermittent job failures at mozilla. In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 8 2021, DOI `10.1145/3468264.3473931`.
Abstract: Continuous delivery of cloud systems requires constant running of jobs (build processes, tests, etc.). One issue that plagues this continuous integration (CI) process are intermittent failures - non-deterministic, false alarms that do not result from a bug in the software or job specification, but rather from issues in the underlying infrastructure. At Mozilla, such intermittent failures are called oranges as a reference to the color of the build status indicator. As such intermittent failures disrupt CI and lead to failures, they erode the developers' trust in the jobs. We present a novel approach that automatically classifies failing jobs to determine whether job execution failures arise from an actual software bug or were caused by flakiness in the job (e.g., test) or the underlying infrastructure. For this purpose, we train classification models using job telemetry data to diagnose failure patterns involving features such as runtime, cpu load, operating system version, or specific platform with high precision. In an evaluation on a set of Mozilla

CI jobs, our approach achieves precision scores of 73%, on average, across all data sets with some test suites achieving precision scores good enough for fully automated classification (i.e., precision scores of up to 100%), and recall scores of 82% on average (up to 94%).

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

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

[**LeGoues2018**] Claire Le Goues, Ciera Jaspan, Ipek Ozkaya, Mary Shaw, and Kathryn T. Stolee. Bridging the gap: From research to practical advice. *IEEE Software*, 35(5):50–57, 9 2018, DOI `10.1109/ms.2018.3571235`.

Abstract: Software developers need actionable guidance, but researchers rarely integrate diverse types of evidence in a way that indicates the recommendations' strength. A levels-ofevidence framework might allow researchers and practitioners to translate research results to a pragmatically useful form.

[**LeGoues2021**] Claire Le Goues, Michael Pradel, Abhik Roychoudhury, and Satish Chandra. Automatic program repair. *IEEE Software*, 38(4):22–27, 7 2021, DOI `10.1109/ms.2021.3072577`.

Abstract: An introduction to a special journal issue on automatic program repair.

[**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

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 devel-

opment 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.

[**Leitao2019**] Roxanne Leitão. Technology-facilitated intimate partner abuse: a qualitative analysis of data from online domestic abuse forums. *Human–Computer Interaction*, 36(3):203–242, 12 2019, DOI `10.1080/07370024.2019.1685883`.

Abstract: This article reports on a qualitative analysis of data gathered from three online discussion forums for victims and survivors of domestic abuse. The analysis focussed on technology-facilitated abuse and the findings cover three main themes, namely, 1) forms of technology-facilitated abuse being discussed on the forums, 2) the ways in which forum members are using technology within the context of intimate partner abuse, and 3) the digital privacy and security advice being exchanged between victims/survivors on the forums. The article concludes with a discussion on the dual role of digital technologies within the context of intimate partner abuse, on the challenges and advantages of digital ubiquity, as well as on the issues surrounding digital evidence of abuse, and the labor of managing digital privacy and security.

[**Lemire2021**] Daniel Lemire. Number parsing at a gigabyte per second. *Software: Practice and Experience*, 51(8):1700–1727, 5 2021, DOI `10.1002/spe.2984`.

Abstract: With disks and networks providing gigabytes per second, parsing decimal numbers from strings becomes a bottleneck. We consider the problem of parsing decimal numbers to the nearest binary floating-point value. The general problem requires variable-precision arithmetic. However, we need at most 17 digits to represent 64-bit standard floating-point numbers (IEEE 754). Thus, we can represent the decimal significand with a single 64-bit word. By combining the significand and precomputed tables, we can compute the nearest floating-point number using as few as one or two 64-bit multiplications. Our implementation can be several times faster than conventional functions present in standard C libraries on modern 64-bit systems (Intel, AMD, ARM, and POWER9). Our work is available as open

source software used by major systems such as Apache Arrow and Yandex ClickHouse. The Go standard library has adopted a version of our approach.

[**Levy2020**] Karen Levy and Bruce Schneier. Privacy threats in intimate relationships. *Journal of Cybersecurity*, 6(1), 1 2020, DOI `10.1093/cybsec/tyaa006`.
Abstract: This article provides an overview of intimate threats: a class of privacy threats that can arise within our families, romantic partnerships, close friendships, and caregiving relationships. Many common assumptions about privacy are upended in the context of these relationships, and many otherwise effective protective measures fail when applied to intimate threats. Those closest to us know the answers to our secret questions, have access to our devices, and can exercise coercive power over us. We survey a range of intimate relationships and describe their common features. Based on these features, we explore implications for both technical privacy design and policy, and offer design recommendations for ameliorating intimate privacy risks.

[**Lewis2013**] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. James Whitehead. Does bug prediction support human developers? findings from a Google case study. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2013, DOI `10.1109/icse.2013.6606583`.
Abstract: While many bug prediction algorithms have been developed by academia, they're often only tested and verified in the lab using automated means. We do not have a strong idea about whether such algorithms are useful to guide human developers. We deployed a bug prediction algorithm across Google, and found no identifiable change in developer behavior. Using our experience, we provide several characteristics that bug prediction algorithms need to meet in order to be accepted by human developers and truly change how developers evaluate their code.

[**Li2013**] Sihan Li, Hucheng Zhou, Haoxiang Lin, Tian Xiao, Haibo Lin, Wei Lin, and Tao Xie. A characteristic study on failures of production distributed data-parallel programs. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2013, DOI `10.1109/icse.2013.6606646`.
Abstract: SCOPE is adopted by thousands of developers from tens of different product teams in Microsoft Bing for daily web-scale data processing, including index building, search ranking, and advertisement display. A SCOPE job is composed of declarative SQL-like queries and imperative C# user-defined functions (UDFs), which are executed in pipeline by thousands of machines. There are tens of thousands of SCOPE jobs executed on Microsoft clusters per day, while some of them fail after a long execution time and thus waste tremendous resources. Reducing SCOPE failures would save significant resources. This paper presents a comprehensive characteristic study on 200 SCOPE failures/fixes and 50 SCOPE failures with debugging statistics from Microsoft Bing, investigating not only major failure types, failure sources, and fixes, but also current debugging practice.

Our major findings include (1) most of the failures (84.5%) are caused by defects in data processing rather than defects in code logic; (2) table-level failures (22.5%) are mainly caused by programmers' mistakes and frequent data-schema changes while row-level failures (62%) are mainly caused by exceptional data; (3) 93% fixes do not change data processing logic; (4) there are 8% failures with root cause not at the failure-exposing stage, making current debugging practice insufficient in this case. Our study results provide valuable guidelines for future development of data-parallel programs. We believe that these guidelines are not limited to SCOPE, but can also be generalized to other similar data-parallel platforms.

[**Liao2016**] Soohyun Nam Liao, Daniel Zingaro, Michael A. Laurenzano, William G. Griswold, and Leo Porter. Lightweight, early identification of at-risk CS1 students. In *Proc. Conference on International Computing Education Research (ICER)*. ACM, 8 2016, DOI `10.1145/2960310.2960315`.
Abstract: Being able to identify low-performing students early in the term may help instructors intervene or differently allocate course resources. Prior work in CS1 has demonstrated that clicker correctness in Peer Instruction courses correlates with exam outcomes and, separately, that machine learning models can be built based on early-term programming assessments. This work aims to combine the best elements of each of these approaches. We offer a methodology for creating models, based on in-class clicker questions, to predict cross-term student performance. In as early as week 3 in a 12-week CS1 course, this model is capable of correctly predicting students as being in danger of failing, or not, for 70% of the students, with only 17% of students misclassified as not at-risk when at-risk. Additional measures to ensure more broad applicability of the methodology, along with possible limitations, are explored.

[**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.

[**Lima2021b**] Igor Lima, Jefferson Silva, Breno Miranda, Gustavo Pinto, and Marcelo d'Amorim. Exposing bugs in JavaScript engines through test transplantation and differential testing. *Software Quality Journal*, 29(1):129–158, 1 2021, DOI `10.1007/s11219-020-09537-8`.
Abstract: JavaScript is a popular programming language today with several implementations competing for market dominance. Although a specification document and a conformance test suite exist to guide engine development, bugs occur and have important practical consequences. Implementing correct engines is challenging because the spec is intentionally incomplete and evolves frequently. This paper investigates the use of test transplantation and differential testing for revealing functional bugs in JavaScript engines. The former technique runs the regression test suite of a given engine on another engine. The latter technique fuzzes existing inputs and then compares the output produced by different engines with a differential oracle. We conducted experiments with engines from five major players—Apple, Facebook, Google, Microsoft, and Mozilla—to assess the effectiveness of test transplantation and differential testing. Our results indicate that both techniques revealed several bugs, many of which are confirmed by developers. We reported 35 bugs with test transplantation (23 of these bugs confirmed and 19 fixed) and reported 24 bugs with differential testing (17 of these confirmed and 10 fixed). Results indicate that most of these bugs affected two engines—Apple's JSC and Microsoft's ChakraCore (24 and 26 bugs, respectively). To summarize, our results show that test transplantation and differential testing are easy to apply and very effective in finding bugs in complex software, such as JavaScript engines.

[**LimaJunior2021**] Manoel Limeira Lima Júnior, Daricélio Soares, Alexandre Plastino, and Leonardo Murta. Predicting the lifetime of pull requests in open-source projects. *Journal of Software: Evolution and Process*, 33(6), 4 2021, DOI `10.1002/smr.2337`.
Abstract: A recent survey using industrial projects has shown that providing an estimate of the lifetime of pull requests to developers helps to speed up their conclusion. Previous work has explored pull request lifetime prediction in open-source projects using regression techniques but with a broad margin of error. The first objective of our work was to reduce the average error rate of the prediction obtained by the regression techniques so far. We performed experiments with different regression techniques and achieved a significant decrease in the mean error rate. The second objective of our work was to obtain a more effective and useful predictive model that can classify pull requests according to five discrete time intervals. We proposed new predictive

attributes for the estimation of the time intervals and employed attribute selection strategies to identify subsets of attributes that could improve the predictive behavior of the classifiers. Our classification approach achieved the best accuracy in all the 20 projects evaluated in comparison with the literature. The average accuracy was of 45.28% to predict pull request lifetime, with an average normalized improvement of 14.68% in relation to the majority class and 6.49% in relation to the state-of-the-art.

[**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.

[**Lo2015**] David Lo, Nachiappan Nagappan, and Thomas Zimmermann. How practitioners perceive the relevance of software engineering research. In *Proc. International Symposium on the Foundations of Software Engineering (FSE)*. ACM, 8 2015, DOI `10.1145/2786805.2786809`.

Abstract: The number of software engineering research papers over the last few years has grown significantly. An important question here is: how relevant is software engineering research to practitioners in the field? To address this question, we conducted a survey at Microsoft where we invited

3,000 industry practitioners to rate the relevance of research ideas contained in 571 ICSE, ESEC/FSE and FSE papers that were published over a five year period. We received 17,913 ratings by 512 practitioners who labelled ideas as essential, worthwhile, unimportant, or unwise. The results from the survey suggest that practitioners are positive towards studies done by the software engineering research community: 71% of all ratings were essential or worthwhile. We found no correlation between the citation counts and the relevance scores of the papers. Through a qualitative analysis of free text responses, we identify several reasons why practitioners considered certain research ideas to be unwise. The survey approach described in this paper is lightweight: on average, a participant spent only 22.5 minutes to respond to the survey. At the same time, the results can provide useful insight to conference organizers, authors, and participating practitioners.

[**Louis2020**] Annie Louis, Santanu Kumar Dash, Earl T. Barr, Michael D. Ernst, and Charles Sutton. Where should i comment my code?: a dataset and model for predicting locations that need comments. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 6 2020, DOI `10.1145/3377816.3381736`.
Abstract: Programmers should write code comments, but not on every line of code. We have created a machine learning model that suggests locations where a programmer should write a code comment. We trained it on existing commented code to learn locations that are chosen by developers. Once trained, the model can predict locations in new code. Our models achieved precision of 74% and recall of 13% in identifying comment-worthy locations. This first success opens the door to future work, both in the new where-to-comment problem and in guiding comment generation. Our code and data is available at http://groups.inf.ed.ac.uk/cup/comment-locator/.

[**Lunn2021**] Stephanie Lunn, Monique Ross, Zahra Hazari, Mark Allen Weiss, Michael Georgiopoulos, and Kenneth Christensen. The impact of technical interviews, and other professional and cultural experiences on students' computing identity. In *Proc. Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, 6 2021, DOI `10.1145/3430665.3456362`.
Abstract: Increasingly companies assess a computing candidate's capabilities using technical interviews (TIs). Yet students struggle to code on demand, and there is already an insufficient amount of computing graduates to meet industry needs. Therefore, it is important to understand students' perceptions of TIs, and other professional experiences (e.g., computing jobs). We surveyed 740 undergraduate computing students at three universities to examine their experiences with the hiring process, as well as the impact of professional and cultural experiences (e.g., familial support) on computing identity. We considered the interactions between these experiences and social identity for groups underrepresented in computing - women, Black/African American, and Hispanic/Latinx students. Among other findings, we observed that students that did not have positive experiences with TIs had a reduced

computing identity, but that facing discrimination during technical interviews had the opposite effect. Social support may play a role. Having friends in computing bolsters computing identity for Hispanic/Latinx students, as does a supportive home environment for women. Also, freelance computing jobs increase computing identity for Black/African American students. Our findings are intended to raise awareness of the best way for educators to help diverse groups of students to succeed, and to inform them of the experiences that may influence students' engagement, resilience, and computing identity development.

[**Luu2021**] Quang-Hung Luu, Man F. Lau, Sebastian P.H. Ng, and Tsong Yueh Chen. Testing multiple linear regression systems with metamorphic testing. *Journal of Systems and Software*, 182:111062, 12 2021, DOI `10.1016/j.jss.2021.111062`.
Abstract: Regression is one of the most commonly used statistical techniques. However, testing regression systems is a great challenge because of the absence of test oracle. In this paper, we show that Metamorphic Testing is an effective approach to test multiple linear regression systems. In doing so, we identify intrinsic mathematical properties of linear regression, and then propose 11 Metamorphic Relations to be used for testing. Their effectiveness is examined using mutation analysis with a range of different regression programs. We further look at how the testing could be adopted in a more effective way. Our work is applicable to examine the reliability of predictive systems based on regression that has been widely used in economics, engineering and science, as well as of the regression calculation manipulated by statistical users.

[**Ma2021**] Yuxing Ma, Tapajit Dey, Chris Bogart, Sadika Amreen, Marat Valiev, Adam Tutko, David Kennard, Russell Zaretzki, and Audris Mockus. World of code: enabling a research workflow for mining and analyzing the universe of open source VCS data. *Empirical Software Engineering*, 26(2), 2 2021, DOI `10.1007/s10664-020-09905-9`.
Abstract: Open source software (OSS) is essential for modern society and, while substantial research has been done on individual (typically central) projects, only a limited understanding of the periphery of the entire OSS ecosystem exists. For example, how are the tens of millions of projects in the periphery interconnected through. technical dependencies, code sharing, or knowledge flow? To answer such questions we: a) create a very large and frequently updated collection of version control data in the entire FLOSS ecosystems named World of Code (WoC), that can completely cross-reference authors, projects, commits, blobs, dependencies, and history of the FLOSS ecosystems and b) provide capabilities to efficiently correct, augment, query, and analyze that data. Our current WoC implementation is capable of being updated on a monthly basis and contains over 18B Git objects. To evaluate its research potential and to create vignettes for its usage, we employ WoC in conducting several research tasks. In particular, we find that it is capable

of supporting trend evaluation, ecosystem measurement, and the determination of package usage. We expect WoC to spur investigation into global properties of OSS development leading to increased resiliency of the entire OSS ecosystem. Our infrastructure facilitates the discovery of key technical dependencies, code flow, and social networks that provide the basis to determine the structure and evolution of the relationships that drive FLOSS activities and innovation.

[**Maalej2014**] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology*, 23(4):1–37, 9 2014, DOI `10.1145/2622669`.

Abstract: Research in program comprehension has evolved considerably over the past decades. However, only little is known about how developers practice program comprehension in their daily work. This article reports on qualitative and quantitative research to comprehend the strategies, tools, and knowledge used for program comprehension. We observed 28 professional developers, focusing on their comprehension behavior, strategies followed, and tools used. In an online survey with 1,477 respondents, we analyzed the importance of certain types of knowledge for comprehension and where developers typically access and share this knowledge. We found that developers follow pragmatic comprehension strategies depending on context. They try to avoid comprehension whenever possible and often put themselves in the role of users by inspecting graphical interfaces. Participants confirmed that standards, experience, and personal communication facilitate comprehension. The team size, its distribution, and open-source experience influence their knowledge sharing and access behavior. While face-to-face communication is preferred for accessing knowledge, knowledge is frequently shared in informal comments. Our results reveal a gap between research and practice, as we did not observe any use of comprehension tools and developers seem to be unaware of them. Overall, our findings call for reconsidering the research agendas towards context-aware tool support.

[**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.

[**Maenpaa2018**] Hanna Mäenpää, Simo Mäkinen, Terhi Kilamo, Tommi Mikkonen, Tomi Männistö, and Paavo Ritala. Organizing for openness: six models for developer involvement in hybrid OSS projects. *Journal of Internet Services and Applications*, 9(1), 8 2018, DOI `10.1186/s13174-018-0088-1`.
Abstract: This article examines organization and governance of commercially influenced Open Source Software development communities by presenting a multiple-case study of six contemporary, hybrid OSS projects. The findings provide in-depth understanding on how to design the participatory nature of the software development process, while understanding the factors that influence the delicate balance of openness, motivations, and governance. The results lay ground for further research on how to organize and manage developer communities where needs of the stakeholders are competing, yet complementary.

[**Majumder2019**] Suvodeep Majumder, Joymallya Chakraborty, Amritanshu Agrawal, and Tim Menzies. Why software projects need heroes: Lessons learned from 1100+ projects. *arxiv.org*, abs/1904.09954, 2019.
Abstract: A "hero" project is one where 80% or more of the contributions are made by the 20% of the developers. Those developers are called "hero" developers. In the literature, heroes projects are deprecated since they might cause bottlenecks in development and communication. However, there is little empirical evidence on this matter. Further, recent studies show that such hero projects are very prevalent. Accordingly, this paper explores the effect of having heroes in project, from a code quality perspective by analyzing 1000+ open source GitHub projects. Based on the analysis, this study finds that (a) majority of the projects are hero projects; and (b)the

commits from "hero developers" (who contribute most to the code) result in far fewer bugs than other developers. That is, contrary to the literature, heroes are standard and very useful part of modern open source projects.

[**Malik2019**] Mashkoor Malik, Alexandre C. G. Schimel, Giuseppe Masetti, Marc Roche, Julian Le Deunf, Margaret F.J. Dolan, Jonathan Beaudoin, Jean-Marie Augustin, Travis Hamilton, and Iain Parnum. Results from the first phase of the seafloor backscatter processing software inter-comparison project. *Geosciences*, 9(12):516, 12 2019, DOI 10.3390/geosciences9120516.

Abstract: Seafloor backscatter mosaics are now routinely produced from multibeam echosounder data and used in a wide range of marine applications. However, large differences (¿5 dB) can often be observed between the mosaics produced by different software packages processing the same dataset. Without transparency of the processing pipeline and the lack of consistency between software packages raises concerns about the validity of the final results. To recognize the source(s) of inconsistency between software, it is necessary to understand at which stage(s) of the data processing chain the differences become substantial. To this end, willing commercial and academic software developers were invited to generate intermediate processed backscatter results from a common dataset, for cross-comparison. The first phase of the study requested intermediate processed results consisting of two stages of the processing sequence: the one-value-per-beam level obtained after reading the raw data and the level obtained after radiometric corrections but before compensation of the angular dependence. Both of these intermediate results showed large differences between software solutions. This study explores the possible reasons for these differences and highlights the need for collaborative efforts between software developers and their users to improve the consistency and transparency of the backscatter data processing sequence.

[**Malloy2018**] Brian A. Malloy and James F. Power. An empirical analysis of the transition from Python 2 to Python 3. *Empirical Software Engineering*, 24(2):751–778, 7 2018, DOI 10.1007/s10664-018-9637-2.

Abstract: Python is one of the most popular and widely adopted programming languages in use today. In 2008 the Python developers introduced a new version of the language, Python 3.0, that was not backward compatible with Python 2, initiating a transitional phase for Python software developers. In this paper, we describe a study that investigates the degree to which Python software developers are making the transition from Python 2 to Python 3. We have developed a Python compliance analyser, PyComply, and have analysed a previously studied corpus of Python applications called Qualitas. We use PyComply to measure and quantify the degree to which Python 3 features are being used, as well as the rate and context of their adoption in the Qualitas corpus. Our results indicate that Python software developers are not exploiting the new features and advantages of Python 3, but rather are choosing to retain backward compatibility with Python 2.

Moreover, Python developers are confining themselves to a language subset, governed by the diminishing intersection of Python 2, which is not under development, and Python 3, which is under development with new features being introduced as the language continues to evolve.

[**Mangano2015**] Nicolas Mangano, Thomas D. LaToza, Marian Petre, and Andre van der Hoek. How software designers interact with sketches at the whiteboard. *IEEE Transactions on Software Engineering*, 41(2):135–156, 2 2015, DOI 10.1109/tse.2014.2362924.

Abstract: Whiteboard sketches play a crucial role in software development, helping to support groups of designers in reasoning about a software design problem at hand. However, little is known about these sketches and how they support design 'in the moment', particularly in terms of the relationships among sketches, visual syntactic elements within sketches, and reasoning activities. To address this gap, we analyzed 14 hours of design activity by eight pairs of professional software designers, manually coding over 4000 events capturing the introduction of visual syntactic elements into sketches, focus transitions between sketches, and reasoning activities. Our findings indicate that sketches serve as a rich medium for supporting design conversations. Designers often use general-purpose notations. Designers introduce new syntactic elements to record aspects of the design, or re-purpose sketches as the design develops. Designers constantly shift focus between sketches, using groups of sketches together that contain complementary information. Finally, sketches play an important role in supporting several types of reasoning activities (mental simulation, review of progress, consideration of alternatives). But these activities often leave no trace and rarely lead to sketch creation. We discuss the implications of these and other findings for the practice of software design at the whiteboard and for the creation of new electronic software design sketching tools.

[**Marinescu2011**] Cristina Marinescu. Are the classes that use exceptions defect prone? In *Proc. International Workshop on Principles on Software Evolution/Workshop on Software Evolution (IWPSE-EVOL)*. ACM Press, 2011, DOI 10.1145/2024445.2024456.

Abstract: Exception handling is a mechanism that highlights exceptional functionality of software systems. Currently many empirical studies point out that sometimes developers neglect exceptional functionality, minimizing its importance. In this paper we investigate if the design entities (classes) that use exceptions are more defect prone than the other classes. The results, based on analyzing three releases of Eclipse, show that indeed the classes that use exceptions are more defect prone than the other classes. Based on our results, developers are advertised to pay more attention to the way they handle exceptions.

[**Masood2020a**] Zainab Masood, Rashina Hoda, and Kelly Blincoe. How agile teams make self-assignment work: a grounded theory study. *Empirical Software Engineering*, 25(6):4962–5005, 9 2020, DOI

`10.1007/s10664-020-09876-x`.

Abstract: Self-assignment, a self-directed method of task allocation in which teams and individuals assign and choose work for themselves, is considered one of the hallmark practices of empowered, self-organizing agile teams. Despite all the benefits it promises, agile software teams do not practice it as regularly as other agile practices such as iteration planning and daily stand-ups, indicating that it is likely not an easy and straighforward practice. There has been very little empirical research on self-assignment. This Grounded Theory study explores how self-assignment works in agile projects. We collected data through interviews with 42 participants representing 28 agile teams from 23 software companies and supplemented these interviews with observations. Based on rigorous application of Grounded Theory analysis procedures such as open, axial, and selective coding, we present a comprehensive grounded theory of making self-assignment work that explains the (a) context and (b) causal conditions that give rise to the need for self-assignment, (c) a set of facilitating conditions that mediate how self-assignment may be enabled, (d) a set of constraining conditions that mediate how self-assignment may be constrained and which are overcome by a set of (e) strategies applied by agile teams, which in turn result in (f) a set of consequences, all in an attempt to make the central phenomenon, self-assignment, work. The findings of this study will help agile practitioners and companies understand different aspects of self-assignment and practice it with confidence regularly as a valuable practice. Additionally, it will help teams already practicing self-assignment to apply strategies to overcome the challenges they face on an everyday basis.

[**Masood2020b**] Zainab Masood, Rashina Hoda, and Kelly Blincoe. Real world scrum a grounded theory of variations in practice. *IEEE Transactions on Software Engineering*, pages 1–1, 2020, DOI `10.1109/tse.2020.3025317`.

Abstract: Scrum, the most popular agile method and project management framework, is widely reported to be used, adapted, misused, and abused in practice. However, not much is known about how Scrum actually works in practice, and critically, where, when, how and why it diverges from Scrum by the book. Through a Grounded Theory study involving semi-structured interviews of 45 participants from 30 companies and observations of five teams, we present our findings on how Scrum works in practice as compared to how it is presented in its formative books. We identify significant variations in these practices such as work breakdown, estimation, prioritization, assignment, the associated roles and artefacts, and discuss the underlying rationales driving the variations. Critically, we claim that not all variations are process misuse/abuse and propose a nuanced classification approach to understanding variations as standard, necessary, contextual, and clear deviations for successful Scrum use and adaptation.

[**Mattmann2015**] Chris A. Mattmann, Joshua Garcia, Ivo Krka, Daniel Popescu, and Nenad Medvidović. Revisiting the anatomy and physiol-

ogy of the grid. *Journal of Grid Computing*, 13(1):19–34, 1 2015, DOI `10.1007/s10723-015-9324-0`.

Abstract: A domain-specific software architecture (DSSA) represents an effective, generalized, reusable solution to constructing software systems within a given application domain. In this paper, we revisit the widely cited DSSA for the domain of grid computing. We have studied systems in this domain over the last ten years. During this time, we have repeatedly observed that, while individual grid systems are widely used and deemed successful, the grid DSSA is actually underspecified to the point where providing a precise answer regarding what makes a software system a grid system is nearly impossible. Moreover, every one of the existing purported grid technologies actually violates the published grid DSSA. In response to this, based on an analysis of the source code, documentation, and usage of eighteen of the most pervasive grid technologies, we have significantly refined the original grid DSSA. We demonstrate that this DSSA much more closely matches the grid technologies studied. Our refinements allow us to more definitively identify a software system as a grid technology, and distinguish it from software libraries, middleware, and frameworks.

[**May2019**] Anna May, Johannes Wachs, and Anikó Hannák. Gender differences in participation and reward on stack overflow. *Empirical Software Engineering*, 24(4):1997–2019, 2 2019, DOI `10.1007/s10664-019-09685-x`.

Abstract: Programming is a valuable skill in the labor market, making the underrepresentation of women in computing an increasingly important issue. Online question and answer platforms serve a dual purpose in this field: they form a body of knowledge useful as a reference and learning tool, and they provide opportunities for individuals to demonstrate credible, verifiable expertise. Issues, such as male-oriented site design or overrepresentation of men among the site's elite may therefore compound the issue of women's underrepresentation in IT. In this paper we audit the differences in behavior and outcomes between men and women on Stack Overflow, the most popular of these Q&A sites. We observe significant differences in how men and women participate in the platform and how successful they are. For example, the average woman has roughly half of the reputation points, the primary measure of success on the site, of the average man. Using an Oaxaca-Blinder decomposition, an econometric technique commonly applied to analyze differences in wages between groups, we find that most of the gap in success between men and women can be explained by differences in their activity on the site and differences in how these activities are rewarded. Specifically, 1) men give more answers than women and 2) are rewarded more for their answers on average, even when controlling for possible confounders such as tenure or buy-in to the site. Women ask more questions and gain more reward per question. We conclude with a hypothetical redesign of the site's scoring system based on these behavioral differences, cutting the reputation gap in half.

[**McGee2011**] Sharon McGee and Des Greer. Software requirements

change taxonomy: evaluation by case study. In *Proc. International Requirements Engineering Conference (RE)*. IEEE, 8 2011, DOI `10.1109/re.2011.6051641`.

Abstract: Although a number of requirements change classifications have been proposed in the literature, there is no empirical assessment of their practical value in terms of their capacity to inform change monitoring and management. This paper describes an investigation of the informative efficacy of a taxonomy of requirements change sources which distinguishes between changes arising from 'market', 'organisation', 'project vision', 'specification' and 'solution'. This investigation was effected through a case study where change data was recorded over a 16 month period covering the development lifecycle of a government sector software application. While insufficiency of data precluded an investigation of changes arising due to the change source of 'market', for the remainder of the change sources, results indicate a significant difference in cost, value to the customer and management considerations. Findings show that higher cost and value changes arose more often from 'organisation' and 'vision' sources; these changes also generally involved the co-operation of more stakeholder groups and were considered to be less controllable than changes arising from the 'specification' or 'solution' sources. Overall, the results suggest that monitoring and measuring change using this classification is a practical means to support change management, understanding and risk visibility.

[**McIntosh2011**] Shane McIntosh, Bram Adams, Thanh H.D. Nguyen, Yasutaka Kamei, and Ahmed E. Hassan. An empirical study of build maintenance effort. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 5 2011, DOI `10.1145/1985793.1985813`.

Abstract: The build system of a software project is responsible for transforming source code and other development artifacts into executable programs and deliverables. Similar to source code, build system specifications require maintenance to cope with newly implemented features, changes to imported Application Program Interfaces (APIs), and source code restructuring. In this paper, we mine the version histories of one proprietary and nine open source projects of different sizes and domain to analyze the overhead that build maintenance imposes on developers. We split our analysis into two dimensions: (1) Build Coupling, i.e., how frequently source code changes require build changes, and (2) Build Ownership, i.e., the proportion of developers responsible for build maintenance. Our results indicate that, despite the difference in scale, the build system churn rate is comparable to that of the source code, and build changes induce more relative churn on the build system than source code changes induce on the source code. Furthermore, build maintenance yields up to a 27% overhead on source code development and a 44% overhead on test development. Up to 79% of source code developers and 89% of test code developers are significantly impacted by build maintenance, yet investment in build experts can reduce the proportion of impacted developers to 22% of source code developers and 24%

of test code developers.

[**McIntosh2021**] Lukas McIntosh and Caroline D. Hardin. Do hackathon projects change the world? an empirical analysis of GitHub repositories. In *Proc. Technical Symposium on Computer Science Education (SIGCSE)*. ACM, 3 2021, DOI `10.1145/3408877.3432435`.
Abstract: Hackathons, the increasingly popular collaborative technology challenge events, are praised for producing modern solutions to real world problems. They have, however, recently been criticized for positing that serious real world problems can be solved in 24-48 hours of undergraduate coding. Projects created at hackathons are typically demos or proof-of-concepts, and little is known about the fate of them after the hackathon ends. Do they receive continued development in preparation for real world use and maintenance as part of actually being used, or are they abandoned? Since participants often use GitHub (Microsoft's popular version control system), it is possible to check. This quantitative, empirical study uses a series of Python scripts to complete a robust analysis of development patterns for all 11,889 of the U.S. based 2018-2019 Major League Hacking (MLH) affiliated hackathon projects which had GitHub repositories. Of these projects, approximately 85% of commits were made within the first month, and approximately 77% of the total commits occurred within the first week. Only 7% of projects had any activity 6 months after the event ended. Evaluated projects had an average of only 3.097 distinct commit dates, and the average of commits divided by the length of the development period was only 0.1. This indicates that few projects receive the post-event attention expected of an actively developed project. Finally, this study offers a dialogue of possible ways to reformat hackathons to help increase the average longevity of the development period for projects.

[**McLeod2011**] Laurie McLeod and Stephen G. MacDonell. Factors that affect software systems development project outcomes. *ACM Computing Surveys*, 43(4):1–56, 10 2011, DOI `10.1145/1978802.1978803`.
Abstract: Determining the factors that have an influence on software systems development and deployment project outcomes has been the focus of extensive and ongoing research for more than 30 years. We provide here a survey of the research literature that has addressed this topic in the period 1996–2006, with a particular focus on empirical analyses. On the basis of this survey we present a new classification framework that represents an abstracted and synthesized view of the types of factors that have been asserted as influencing project outcomes.

[**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 stud-

ies 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.

[**Meneely2011**] Andrew Meneely, Pete Rotella, and Laurie Williams. Does adding manpower also affect quality? an empirical, longitudinal analysis. In *Proc. International Symposium on Foundations of Software Engineering/International Symposium on the Foundations of Software Engineering (SIGSOFT/FSE)*. ACM Press, 2011, DOI `10.1145/2025113.2025128`.

Abstract: With each new developer to a software development team comes a greater challenge to manage the communication, coordination, and knowledge transfer amongst teammates. Fred Brooks discusses this challenge in The Mythical Man-Month by arguing that rapid team expansion can lead to a complex team organization structure. While Brooks focuses on productivity loss as the negative outcome, poor product quality is also a substantial concern. But if team expansion is unavoidable, can any quality impacts be mitigated? Our objective is to guide software engineering managers by empirically analyzing the effects of team size, expansion, and structure on product quality. We performed an empirical, longitudinal case study of a large Cisco networking product over a five year history. Over that time, the team underwent periods of no expansion, steady expansion, and accelerated expansion. Using team-level metrics, we quantified characteristics of team expansion, including team size, expansion rate, expansion acceleration, and modularity with respect to department designations. We examined statistical correlations between our monthly team-level metrics and monthly product-level metrics. Our results indicate that increased team size and linear growth are correlated with later periods of better product quality. However, periods

of accelerated team expansion are correlated with later periods of reduced software quality. Furthermore, our linear regression prediction model based on team metrics was able to predict the product's post-release failure rate within a 95% prediction interval for 38 out of 40 months. Our analysis provides insight for project managers into how the expansion of development teams can impact product quality.

[**Meng2013**] Na Meng, Miryung Kim, and Kathryn S. McKinley. Lase: locating and applying systematic edits by learning from examples. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2013, DOI `10.1109/icse.2013.6606596`.
Abstract: Adding features and fixing bugs often require systematic edits that make similar, but not identical, changes to many code locations. Finding all the relevant locations and making the correct edits is a tedious and error-prone process for developers. This paper addresses both problems using edit scripts learned from multiple examples. We design and implement a tool called LASE that (1) creates a context-aware edit script from two or more examples, and uses the script to (2) automatically identify edit locations and to (3) transform the code. We evaluate LASE on an oracle test suite of systematic edits from Eclipse JDT and SWT. LASE finds edit locations with 99% precision and 89% recall, and transforms them with 91% accuracy. We also evaluate LASE on 37 example systematic edits from other open source programs and find LASE is accurate and effective. Furthermore, we confirmed with developers that LASE found edit locations which they missed. Our novel algorithm that learns from multiple examples is critical to achieving high precision and recall; edit scripts created from only one example produce too many false positives, false negatives, or both. Our results indicate that LASE should help developers in automating systematic editing. Whereas most prior work either suggests edit locations or performs simple edits, LASE is the first to do both for nontrivial program edits.

[**Menzies2016**] Tim Menzies, William Nichols, Forrest Shull, and Lucas Layman. Are delayed issues harder to resolve? revisiting cost-to-fix of defects throughout the lifecycle. *Empirical Software Engineering*, 22(4):1903–1935, 11 2016, DOI `10.1007/s10664-016-9469-x`.
Abstract: Many practitioners and academics believe in a delayed issue effect (DIE); i.e. the longer an issue lingers in the system, the more effort it requires to resolve. This belief is often used to justify major investments in new development processes that promise to retire more issues sooner. This paper tests for the delayed issue effect in 171 software projects conducted around the world in the period from 2006–2014. To the best of our knowledge, this is the largest study yet published on this effect. We found no evidence for the delayed issue effect; i.e. the effort to resolve issues in a later phase was not consistently or substantially greater than when issues were resolved soon after their introduction. This paper documents the above study and explores reasons for this mismatch between this common rule of thumb and empirical data. In summary, DIE is not some constant across all projects.

Rather, DIE might be an historical relic that occurs intermittently only in certain kinds of projects. This is a significant result since it predicts that new development processes that promise to faster retire more issues will not have a guaranteed return on investment (depending on the context where applied), and that a long-held truth in software engineering should not be considered a global truism.

[**Meyer2014**] André N. Meyer, Thomas Fritz, Gail C. Murphy, and Thomas Zimmermann. Software developers' perceptions of productivity. In *Proc. International Symposium on the Foundations of Software Engineering (FSE)*. ACM, 11 2014, DOI `10.1145/2635868.2635892`.

Abstract: The better the software development community becomes at creating software, the more software the world seems to demand. Although there is a large body of research about measuring and investigating productivity from an organizational point of view, there is a paucity of research about how software developers, those at the front-line of software construction, think about, assess and try to improve their productivity. To investigate software developers' perceptions of software development productivity, we conducted two studies: a survey with 379 professional software developers to help elicit themes and an observational study with 11 professional software developers to investigate emergent themes in more detail. In both studies, we found that developers perceive their days as productive when they complete many or big tasks without significant interruptions or context switches. Yet, the observational data we collected shows our participants performed significant task and activity switching while still feeling productive. We analyze such apparent contradictions in our findings and use the analysis to propose ways to better support software developers in a retrospection and improvement of their productivity through the development of new tools and the sharing of best practices.

[**Meyer2021**] Andre N. Meyer, Earl T. Barr, Christian Bird, and Thomas Zimmermann. Today was a good day: The daily life of software developers. *IEEE Transactions on Software Engineering*, 47(5):863–880, 5 2021, DOI `10.1109/tse.2019.2904957`.

Abstract: What is a good workday for a software developer? What is a typical workday? We seek to answer these two questions to learn how to make good days typical. Concretely, answering these questions will help to optimize development processes and select tools that increase job satisfaction and productivity. Our work adds to a large body of research on how software developers spend their time. We report the results from 5,971 responses of professional developers at Microsoft, who reflected about what made their workdays good and typical, and self-reported about how they spent their time on various activities at work. We developed conceptual frameworks to help define and characterize developer workdays from two new perspectives: good and typical. Our analysis confirms some findings in previous work, including the fact that developers actually spend little time on development and developers' aversion for meetings and interruptions. It also discovered

new findings, such as that only 1.7 percent of survey responses mentioned emails as a reason for a bad workday, and that meetings and interruptions are only unproductive during development phases; during phases of planning, specification and release, they are common and constructive. One key finding is the importance of agency, developers' control over their workday and whether it goes as planned or is disrupted by external factors. We present actionable recommendations for researchers and managers to prioritize process and tool improvements that make good workdays typical. For instance, in light of our finding on the importance of agency, we recommend that, where possible, managers empower developers to choose their tools and tasks.

[**Miedema2021**] Daphne Miedema, Efthimia Aivaloglou, and George Fletcher. Identifying SQL misconceptions of novices: findings from a think-aloud study. In *Proc. Conference on International Computing Education Research (ICER)*. ACM, 8 2021, DOI `10.1145/3446871.3469759`.

Abstract: SQL is the most commonly taught database query language. While previous research has investigated the errors made by novices during SQL query formulation, the underlying causes for these errors have remained unexplored. Understanding the basic misconceptions held by novices which lead to these errors would help improve how we teach query languages to our students. In this paper we aim to identify the misconceptions that might be the causes of documented SQL errors that novices make. To this end, we conducted a qualitative think-aloud study to gather information on the thinking process of university students while solving query formulation problems. With the queries in hand, we analyzed the underlying causes for the errors made by our participants. In this paper we present the identified SQL misconceptions organized into four top-level categories: misconceptions based in previous course knowledge, generalization-based misconceptions, language-based misconceptions, and misconceptions due to an incomplete or incorrect mental model. A deep exploration of misconceptions can uncover gaps in instruction. By drawing attention to these, we aim to improve SQL education.

[**Miller2016**] Craig S. Miller and Amber Settle. Some trouble with transparency: an analysis of student errors with object-oriented Python. In *Proc. Conference on International Computing Education Research (ICER)*. ACM, 8 2016, DOI `10.1145/2960310.2960327`.

Abstract: We investigated implications of transparent mechanisms in the context of an introductory object-oriented programming course using Python. Here transparent mechanisms are those that reveal how the instance object in Python relates to its instance data. We asked students to write a new method for a provided Python class in an attempt to answer two research questions: 1) to what extent do Python's transparent OO mechanisms lead to student difficulties? and 2) what are common pitfalls in OO programming using Python that instructors should address? Our methodology also presented the correct answer to the students and solicited their comments on their submission. We conducted a content analysis to classify errors in

the student submissions. We find that most students had difficulty with the instance (self) object, either by omitting the parameter in the method definition, by failing to use the instance object when referencing attributes of the object, or both. Reference errors in general were more common than other errors, including misplaced returns and indentation errors. These issues may be connected to problems with parameter passing and using dot-notation, which we argue are prerequisites for OO development in Python.

[**Miller2020**] Barton Miller, Mengxiao Zhang, and Elisa Heymann. The relevance of classic fuzz testing: Have we solved this one? *IEEE Transactions on Software Engineering*, pages 1–1, 2020, DOI `10.1109/tse.2020.3047766`.

Abstract: As fuzz testing has passed its 30th anniversary, and in the face of the incredible progress in fuzz testing techniques and tools, the question arises if the classic, basic fuzz technique is still useful and applicable? In that tradition, we have updated the basic fuzz tools and testing scripts and applied them to a large collection of Unix utilities on Linux, FreeBSD, and MacOS. As before, our failure criteria was whether the program crashed or hung. We found that 9 crash or hang out of 74 utilities on Linux, 15 out of 78 utilities on FreeBSD, and 12 out of 76 utilities on MacOS. A total of 24 different utilities failed across the three platforms. We note that these failure rates are somewhat higher than our in previous 1995, 2000, and 2006 studies of the reliability of command line utilities. In the basic fuzz tradition, we debugged each failed utility and categorized the causes the failures. Classic categories of failures, such as pointer and array errors and not checking return codes, were still broadly present in the current results. In addition, we found a couple of new categories of failures appearing. We present examples of these failures to illustrate the programming practices that allowed them to happen. As a side note, we tested the limited number of utilities available in a modern programming language (Rust) and found them to be of no better reliability than the standard ones.

[**Mitropoulos2019**] Dimitris Mitropoulos, Panos Louridas, Vitalis Salis, and Diomidis Spinellis. Time present and time past: Analyzing the evolution of JavaScript code in the wild. In *Proc. International Conference on Mining Software Repositories (MSR)*. IEEE, 5 2019, DOI `10.1109/msr.2019.00029`.

Abstract: JavaScript is one of the web's key building blocks. It is used by the majority of web sites and it is supported by all modern browsers. We present the first large-scale study of client-side JavaScript code over time. Specifically, we have collected and analyzed a dataset containing daily snapshots of JavaScript code coming from Alexa's Top 10000 web sites (7.5 GB per day) for nine consecutive months, to study different temporal aspects of web client code. We found that scripts change often; typically every few days, indicating a rapid pace in web applications development. We also found that the lifetime of web sites themselves, measured as the time between JavaScript changes, is also short, in the same time scale. We then performed a qualitative analysis to investigate the nature of the changes that take place. We

found that apart from standard changes such as the introduction of new functions, many changes are related to online configuration management. In addition, we examined JavaScript code reuse over time and especially the widespread reliance on third-party libraries. Furthermore, we observed how quality issues evolve by employing established static analysis tools to identify potential software bugs, whose evolution we tracked over time. Our results show that quality issues seem to persist over time, while vulnerable libraries tend to decrease.

[**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.

[**Mockus2010**] Audris Mockus. Organizational volatility and its effects on software defects. In *Proc. International Symposium on the Foundations of Software Engineering (FSE)*. ACM Press, 2010, DOI `10.1145/1882291.1882311`.

Abstract: The key premise of an organization is to allow more efficient production, including production of high quality software. To achieve that, an organization defines roles and reporting relationships. Therefore, changes in organization's structure are likely to affect product's quality. We propose and investigate a relationship between developer-centric measures of organizational change and the probability of customer-reported defects in the context of a large software project. We find that the proximity to an organizational change is significantly associated with reductions in software quality. We also replicate results of several prior studies of software quality supporting findings that code, change, and developer characteristics affect fault-proneness. In contrast to prior studies we find that distributed development decreases quality. Furthermore, recent departures from an organiza-

tion were associated with increased probability of customer-reported defects, thus demonstrating that in the observed context the organizational change reduces product quality.

[**Moe2010**] Nils Brede Moe, Torgeir Dingsøyr, and Tore Dybå. A teamwork model for understanding an agile team: A case study of a Scrum project. *Information and Software Technology*, 52(5):480–491, 5 2010, DOI `10.1016/j.infsof.2009.11.004`.
Abstract: Context: Software development depends significantly on team performance, as does any process that involves human interaction. Objective: Most current development methods argue that teams should self-manage. Our objective is thus to provide a better understanding of the nature of self-managing agile teams, and the teamwork challenges that arise when introducing such teams. Method: We conducted extensive fieldwork for 9months in a software development company that introduced Scrum. We focused on the human sensemaking, on how mechanisms of teamwork were understood by the people involved. Results: We describe a project through Dickinson and McIntyre's teamwork model, focusing on the interrelations between essential teamwork components. Problems with team orientation, team leadership and coordination in addition to highly specialized skills and corresponding division of work were important barriers for achieving team effectiveness. Conclusion: Transitioning from individual work to self-managing teams requires a reorientation not only by developers but also by management. This transition takes time and resources, but should not be neglected. In addition to Dickinson and McIntyre's teamwork components, we found trust and shared mental models to be of fundamental importance.

[**Mokhov2018**] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. Build systems à la carte. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–29, 7 2018, DOI `10.1145/3236774`.
Abstract: Build systems are awesome, terrifying—and unloved. They are used by every developer around the world, but are rarely the object of study. In this paper we offer a systematic, and executable, framework for developing and comparing build systems, viewing them as related points in landscape rather than as isolated phenomena. By teasing apart existing build systems, we can recombine their components, allowing us to prototype new build systems with desired properties.

[**Moldon2021**] Lukas Moldon, Markus Strohmaier, and Johannes Wachs. How gamification affects software developers: Cautionary evidence from a natural experiment on GitHub. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2021, DOI `10.1109/icse43902.2021.00058`.
Abstract: We examine how the behavior of software developers changes in response to removing gamification elements from GitHub, an online platform for collaborative programming and software development. We find that the unannounced removal of daily activity streak counters from the user interface (from user profile pages) was followed by significant changes in behavior.

Long-running streaks of activity were abandoned and became less common. Weekend activity decreased and days in which developers made a single contribution became less common. Synchronization of streaking behavior in the platform's social network also decreased, suggesting that gamification is a powerful channel for social influence. Focusing on a set of software developers that were publicly pursuing a goal to make contributions for 100 days in a row, we find that some of these developers abandon this quest following the removal of the public streak counter. Our findings provide evidence for the significant impact of gamification on the behavior of developers on large collaborative programming and software development platforms. They urge caution: gamification can steer the behavior of software developers in unexpected and unwanted directions.

[**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.

[**Muhammad2019**] Hisham Muhammad, Lucas C. Villa Real, and Michael Homer. Taxonomy of package management in programming languages and operating systems. In *Proc. Workshop on Programming Languages and Operating Systems (PLOS)*. ACM, 10 2019, DOI `10.1145/3365137.3365402`.
Abstract: Package management is instrumental for programming languages and operating systems, and yet it is neglected by both areas as an implementation detail. For this reason, it lacks the same kind of conceptual organization: we lack terminology to classify them or to reason about their design trade-offs. In this paper, we share our experience in both OS and language-specific package manager development, categorizing families of package managers and discussing their design implications beyond particular implementations. We also identify possibilities in the still largely unexplored area of package manager interoperability.

[**MurphyHill2021**] Emerson Murphy-Hill, Ciera Jaspan, Caitlin Sadowski, David Shepherd, Michael Phillips, Collin Winter, Andrea Knight, Edward Smith, and Matthew Jorde. What predicts software developers' productivity? *IEEE Transactions on Software Engineering*, 47(3):582–594, 3 2021, DOI `10.1109/tse.2019.2900308`.
Abstract: Organizations have a variety of options to help their software developers become their most productive selves, from modifying office layouts, to investing in better tools, to cleaning up the source code. But which options will have the biggest impact? Drawing from the literature in software engineering and industrial/organizational psychology to identify factors that correlate with productivity, we designed a survey that asked 622 developers

across 3 companies about these productivity factors and about self-rated productivity. Our results suggest that the factors that most strongly correlate with self-rated productivity were non-technical factors, such as job enthusiasm, peer support for new ideas, and receiving useful feedback about job performance. Compared to other knowledge workers, our results also suggest that software developers' self-rated productivity is more strongly related to task variety and ability to work remotely.

[**Nagappan2008**] Nachiappan Nagappan, E. Michael Maximilien, Thirumalesh Bhat, and Laurie Williams. Realizing quality improvement through test driven development: results and experiences of four industrial teams. *Empirical Software Engineering*, 13(3):289–302, 2 2008, DOI 10.1007/s10664-008-9062-z.

Abstract: Test-driven development (TDD) is a software development practice that has been used sporadically for decades. With this practice, a software engineer cycles minute-by-minute between writing failing unit tests and writing implementation code to pass those tests. Test-driven development has recently re-emerged as a critical enabling practice of agile software development methodologies. However, little empirical evidence supports or refutes the utility of this practice in an industrial context. Case studies were conducted with three development teams at Microsoft and one at IBM that have adopted TDD. The results of the case studies indicate that the pre-release defect density of the four products decreased between 40% and 90% relative to similar projects that did not use the TDD practice. Subjectively, the teams experienced a 15–35% increase in initial development time after adopting TDD.

[**Nagappan2015**] Meiyappan Nagappan, Romain Robbes, Yasutaka Kamei, Éric Tanter, Shane McIntosh, Audris Mockus, and Ahmed E. Hassan. An empirical study of goto in C code from GitHub repositories. In *Proc. International Symposium on the Foundations of Software Engineering (FSE)*. ACM, 8 2015, DOI 10.1145/2786805.2786834.

Abstract: It is nearly 50 years since Dijkstra argued that goto obscures the flow of control in program execution and urged programmers to abandon the goto statement. While past research has shown that goto is still in use, little is known about whether goto is used in the unrestricted manner that Dijkstra feared, and if it is 'harmful' enough to be a part of a post-release bug. We, therefore, conduct a two part empirical study - (1) qualitatively analyze a statistically rep- resentative sample of 384 files from a population of almost 250K C programming language files collected from over 11K GitHub repositories and find that developers use goto in C files for error handling $(80.21\pm5\%)$ and cleaning up resources at the end of a procedure $(40.36\pm5\%)$; and (2) quantitatively analyze the commit history from the release branches of six OSS projects and find that no goto statement was re- moved/modified in the post-release phase of four of the six projects. We conclude that developers limit themselves to using goto appropriately in most cases, and not in

an unrestricted manner like Dijkstra feared, thus suggesting that goto does not appear to be harmful in practice.

[**Nakshatri2016**] Suman Nakshatri, Maithri Hegde, and Sahithi Thandra. Analysis of exception handling patterns in Java projects. In *Proc. International Conference on Mining Software Repositories (MSR)*. ACM, 5 2016, DOI 10.1145/2901739.2903499.

Abstract: Exception handling is a powerful tool provided by many programming languages to help developers deal with unforeseen conditions. Java is one of the few programming languages to enforce an additional compilation check on certain sub- classes of the Exception class through checked exceptions. As part of this study, empirical data was extracted from software projects developed in Java. The intent is to explore how developers respond to checked exceptions and identify common patterns used by them to deal with exceptions, checked or otherwise. Bloch's book "Effective Java" [1] was used as reference for best practices in exception handling. These recommendations were compared against results from the empirical data. Results of this study indicate that most programmers ignore checked exceptions and leave them un- noticed. Additionally, it is observed that classes higher in the exception class hierarchy are more frequently used as compared to specific exception subclasses.

[**Near2016**] Joseph P. Near and Daniel Jackson. Finding security bugs in web applications using a catalog of access control patterns. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 5 2016, DOI 10.1145/2884781.2884836.

Abstract: We propose a specification-free technique for finding missing security checks in web applications using a catalog of access control patterns in which each pattern models a common access control use case. Our implementation, SPACE, checks that every data exposure allowed by an application's code matches an allowed exposure from a security pattern in our catalog. The only user-provided input is a mapping from application types to the types of the catalog; the rest of the process is entirely automatic. In an evaluation on the 50 most watched Ruby on Rails applications on Github, SPACE reported 33 possible bugs—23 previously unknown security bugs, and 10 false positives.

[**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.

[**Nielebock2018**] Sebastian Nielebock, Dariusz Krolikowski, Jacob Krüger, Thomas Leich, and Frank Ortmeier. Commenting source code: is it worth it for small programming tasks? *Empirical Software Engineering*, 24(3):1418–1457, 11 2018, DOI `10.1007/s10664-018-9664-z`.
Abstract: Maintaining a program is a time-consuming and expensive task in software engineering. Consequently, several approaches have been proposed to improve the comprehensibility of source code. One of such approaches are comments in the code that enable developers to explain the program with their own words or predefined tags. Some empirical studies indicate benefits of comments in certain situations, while others find no benefits at all. Thus, the real effect of comments on software development remains uncertain. In this article, we describe an experiment in which 277 participants, mainly professional software developers, performed small programming tasks on differently commented code. Based on quantitative and qualitative feedback, we i) partly replicate previous studies, ii) investigate performances of differently experienced participants when confronted with varying types of comments, and iii) discuss the opinions of developers on comments. Our results indicate that comments seem to be considered more important in previous studies and by our participants than they are for small programming tasks. While other mechanisms, such as proper identifiers, are considered more helpful by our participants, they also emphasize the necessity of comments in certain situations.

[**Nussli2012**] Marc-Antoine Nüssli and Patrick Jermann. Effects of sharing text selections on gaze cross-recurrence and interaction quality in a pair programming task. In *Proc. Conference on Computer Supported Cooperative Work (CSCW)*. ACM Press, 2012, DOI `10.1145/2145204.2145371`.
Abstract: We present a dual eye-tracking study that demonstrates the effect of sharing selection among collaborators in a remote pair-programming

scenario. Forty pairs of engineering students completed several program understanding tasks while their gaze was synchronously recorded. The coupling of the programmers' focus of attention was measured by a cross-recurrence analysis of gaze that captures how much programmers look at the same sequence of spots within a short time span. A high level of gaze cross-recurrence is typical for pairs who actively engage in grounding efforts to build and maintain shared understanding. As part of their grounding efforts, programmers may use text selection to perform collaborative references. Broadcast selections serve as indexing sites for the selector as they attract non-selector's gaze shortly after they become visible. Gaze cross-recurrence is highest when selectors accompany their selections with speech to produce a multimodal reference.

[**Olejniczak2020**] Anthony J. Olejniczak and Molly J. Wilson. Who's writing open access (OA) articles? characteristics of OA authors at ph.d.-granting institutions in the united states. *Quantitative Science Studies*, 1(4):1429–1450, 12 2020, DOI `10.1162/qss_a_00091`.
Abstract: The open access (OA) publication movement aims to present research literature to the public at no cost and with no restrictions. While the democratization of access to scholarly literature is a primary focus of the movement, it remains unclear whether OA has uniformly democratized the corpus of freely available research, or whether authors who choose to publish in OA venues represent a particular subset of scholars—those with access to resources enabling them to afford article processing charges (APCs). We investigated the number of OA articles with article processing charges (APC OA) authored by 182,320 scholars with known demographic and institutional characteristics at American research universities across 11 broad fields of study. The results show, in general, that the likelihood for a scholar to author an APC OA article increases with male gender, employment at a prestigious institution (AAU member universities), association with a STEM discipline, greater federal research funding, and more advanced career stage (i.e., higher professorial rank). Participation in APC OA publishing appears to be skewed toward scholars with greater access to resources and job security.

[**Oliveira2020**] Edson Oliveira, Eduardo Fernandes, Igor Steinmacher, Marco Cristo, Tayana Conte, and Alessandro Garcia. Code and commit metrics of developer productivity: a study on team leaders perceptions. *Empirical Software Engineering*, 25(4):2519–2549, 4 2020, DOI `10.1007/s10664-020-09820-z`.
Abstract: Context Developer productivity is essential to the success of software development organizations. Team leaders use developer productivity information for managing tasks in a software project. Developer productivity metrics can be computed from software repositories data to support leaders' decisions. We can classify these metrics in code-based metrics, which rely on the amount of produced code, and commit-based metrics, which rely on commit activity. Although metrics can assist a leader, organizations usually neglect their usage and end up sticking to the leaders' subjective perceptions

only. Objective We aim to understand whether productivity metrics can complement the leaders' perceptions. We also aim to capture leaders' impressions about relevance and adoption of productivity metrics in practice. Method This paper presents a multi-case empirical study performed in two organizations active for more than 18 years. Eight leaders of nine projects have ranked the developers of their teams by productivity. We quantitatively assessed the correlation of leaders' rankings versus metric-based rankings. As a complement, we interviewed leaders for qualitatively understanding the leaders' impressions about relevance and adoption of productivity metrics given the computed correlations. Results Our quantitative data suggest a greater correlation of the leaders' perceptions with code-based metrics when compared to commit-based metrics. Our qualitative data reveal that leaders have positive impressions of code-based metrics and potentially would adopt them. Conclusions Data triangulation of productivity metrics and leaders' perceptions can strengthen the organization conviction about productive developers and can reveal productive developers not yet perceived by team leaders and probably underestimated in the organization.

[**Olsson2021**] Jesper Olsson, Erik Risfelt, Terese Besker, Antonio Martini, and Richard Torkar. Measuring affective states from technical debt. *Empirical Software Engineering*, 26(5), 7 2021, DOI `10.1007/s10664-021-09998-w`.
Abstract: Context: Software engineering is a human activity. Despite this, human aspects are under-represented in technical debt research, perhaps because they are challenging to evaluate. Objective: This study's objective was to investigate the relationship between technical debt and affective states (feelings, emotions, and moods) from software practitioners. Method: Forty participants (N=40) from twelve companies took part in a mixed-methods approach, consisting of a repeated-measures (r=5) experiment (n=200), a survey, and semi-structured interviews. From the qualitative data, it is clear that technical debt activates a substantial portion of the emotional spectrum and is psychologically taxing. Further, the practitioners' reactions to technical debt appear to fall in different levels of maturity. Results: The statistical analysis shows that different design smells (strong indicators of technical debt) negatively or positively impact affective states. Conclusions: We argue that human aspects in technical debt are important factors to consider, as they may result in, e.g., procrastination, apprehension, and burnout.

[**Overney2020**] Cassandra Overney, Jens Meinicke, Christian Kästner, and Bogdan Vasilescu. How to not get rich: an empirical study of donations in open source. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 6 2020, DOI `10.1145/3377811.3380410`.
Abstract: Open source is ubiquitous and many projects act as critical infrastructure, yet funding and sustaining the whole ecosystem is challenging. While there are many different funding models for open source and concerted efforts through foundations, donation platforms like PayPal, Patreon, and OpenCollective are popular and low-bar platforms to raise funds for open-source development. With a mixed-method study, we investigate the

emerging and largely unexplored phenomenon of donations in open source. Specifically, we quantify how commonly open-source projects ask for donations, statistically model characteristics of projects that ask for and receive donations, analyze for what the requested funds are needed and used, and assess whether the received donations achieve the intended outcomes. We find 25,885 projects asking for donations on GitHub, often to support engineering activities; however, we also find no clear evidence that donations influence the activity level of a project. In fact, we find that donations are used in a multitude of ways, raising new research questions about effective funding.

[**Palomba2021**] Fabio Palomba, Damian Andrew Tamburri, Francesca Arcelli Fontana, Rocco Oliveto, Andy Zaidman, and Alexander Serebrenik. Beyond technical aspects: How do community smells influence the intensity of code smells? *IEEE Transactions on Software Engineering*, 47(1):108–129, 1 2021, DOI `10.1109/tse.2018.2883603`.
Abstract: Code smells are poor implementation choices applied by developers during software evolution that often lead to critical flaws or failure. Much in the same way, community smells reflect the presence of organizational and socio-technical issues within a software community that may lead to additional project costs. Recent empirical studies provide evidence that community smells are often—if not always—connected to circumstances such as code smells. In this paper we look deeper into this connection by conducting a mixed-methods empirical study of 117 releases from 9 open-source systems. The qualitative and quantitative sides of our mixed-methods study were run in parallel and assume a mutually-confirmative connotation. On the one hand, we survey 162 developers of the 9 considered systems to investigate whether developers perceive relationship between community smells and the code smells found in those projects. On the other hand, we perform a fine-grained analysis into the 117 releases of our dataset to measure the extent to which community smells impact code smell intensity (i.e., criticality). We then propose a code smell intensity prediction model that relies on both technical and community-related aspects. The results of both sides of our mixed-methods study lead to one conclusion: community-related factors contribute to the intensity of code smells. This conclusion supports the joint use of community and code smells detection as a mechanism for the joint management of technical and social problems around software development communities.

[**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.

[**Pan2008**] Kai Pan, Sunghun Kim, and E. James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 8 2008, DOI `10.1007/s10664-008-9077-5`.
Abstract: Twenty-seven automatically extractable bug fix patterns are defined using the syntax components and context of the source code involved in bug fix changes. Bug fix patterns are extracted from the configuration management repositories of seven open source projects, all written in Java (Eclipse, Columba, JEdit, Scarab, ArgoUML, Lucene, and MegaMek). Defined bug fix patterns cover 45.7% to 63.3% of the total bug fix hunk pairs in these projects. The frequency of occurrence of each bug fix pattern is computed across all projects. The most common individual patterns are MC-DAP (method call with different actual parameter values) at 14.9–25.5%, IF-CC (change in if conditional) at 5.6–18.6%, and AS-CE (change of assignment expression) at 6.0–14.2%. A correlation analysis on the extracted pattern instances on the seven projects shows that six have very similar bug fix pattern frequencies. Analysis of if conditional bug fix sub-patterns shows a trend towards increasing conditional complexity in if conditional fixes. Analysis of five developers in the Eclipse projects shows overall consistency with project-level bug fix pattern frequencies, as well as distinct variations among developers in their rates of producing various bug patterns. Overall, data in the paper suggest that developers have difficulty with specific code situations at surprisingly consistent rates. There appear to be broad mechanisms causing the injection of bugs that are largely independent of the type of software being produced.

[**Pankratius2012**] Victor Pankratius, Felix Schmidt, and Gilda Garreton. Combining functional and imperative programming for multicore software: an empirical study evaluating Scala and Java. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 6 2012, DOI `10.1109/icse.2012.6227200`.

Abstract: Recent multi-paradigm programming languages combine functional and imperative programming styles to make software development easier. Given today's proliferation of multicore processors, parallel programmers are supposed to benefit from this combination, as many difficult problems can be expressed more easily in a functional style while others match an imperative style. Due to a lack of empirical evidence from controlled studies, however, important software engineering questions are largely unanswered. Our paper is the first to provide thorough empirical results by using Scala and Java as a vehicle in a controlled comparative study on multicore software development. Scala combines functional and imperative programming while Java focuses on imperative shared-memory programming. We study thirteen programmers who worked on three projects, including an industrial application, in both Scala and Java. In addition to the resulting 39 Scala programs and 39 Java programs, we obtain data from an industry software engineer who worked on the same project in Scala. We analyze key issues such as effort, code, language usage, performance, and programmer satisfaction. Contrary to popular belief, the functional style does not lead to bad performance. Average Scala run-times are comparable to Java, lowest run-times are sometimes better, but Java scales better on parallel hardware. We confirm with statistical significance Scala's claim that Scala code is more compact than Java code, but clearly refute other claims of Scala on lower programming effort and lower debugging effort. Our study also provides explanations for these observations and shows directions on how to improve multi-paradigm languages in the future.

[**Parnin2012**] Chris Parnin and Spencer Rugaber. Programmer information needs after memory failure. In *Proc. International Conference on Program Comprehension (ICPC)*. IEEE, 6 2012, DOI `10.1109/icpc.2012.6240479`.
Abstract: Despite its vast capacity and associative powers, the human brain does not deal well with interruptions. Particularly in situations where information density is high, such as during a programming task, recovering from an interruption requires extensive time and effort. Although modern program development environments have begun to recognize this problem, none of these tools take into account the brain's structure and limitations. In this paper, we present a conceptual framework for understanding the strengths and weaknesses of human memory, particularly with respect to it ability to deal with work interruptions. The framework explains empirical results obtained from experiments in which programmers were interrupted while working. Based on the framework, we discuss programmer information needs that development tools must satisfy and suggest several memory aids such tools could provide. We also describe our prototype implementation of these memory aids.

[**Passos2021**] Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesus Alejandro Padilla. A study of feature scattering in the linux kernel. *IEEE Transactions on Software Engineering*, 47(1):146–164, 1 2021, DOI

10.1109/tse.2018.2884911.
Abstract: Feature code is often scattered across a software system. Scattering is not necessarily bad if used with care, as witnessed by systems with highly scattered features that evolved successfully. Feature scattering, often realized with a pre-processor, circumvents limitations of programming languages and software architectures. Unfortunately, little is known about the principles governing scattering in large and long-living software systems. We present a longitudinal study of feature scattering in the Linux kernel, complemented by a survey with 74, and interviews with nine Linux kernel developers. We analyzed almost eight years of the kernel's history, focusing on its largest subsystem: device drivers. We learned that the ratio of scattered features remained nearly constant and that most features were introduced without scattering. Yet, scattering easily crosses subsystem boundaries, and highly scattered outliers exist. Scattering often addresses a performance-maintenance tradeoff (alleviating complicated APIs), hardware design limitations, and avoids code duplication. While developers do not consciously enforce scattering limits, they actually improve the system design and refactor code, thereby mitigating pre-processor idiosyncrasies or reducing its use.

[**Patitsas2016**] Elizabeth Patitsas, Jesse Berlin, Michelle Craig, and Steve Easterbrook. Evidence that computer science grades are not bimodal. In *Proc. Conference on International Computing Education Research (ICER)*. ACM, 8 2016, DOI 10.1145/2960310.2960312.
Abstract: It is commonly thought that CS grades are bimodal. We statistically analyzed 778 distributions of final course grades from a large research university, and found only 5.8% of the distributions passed tests of multimodality. We then devised a psychology experiment to understand why CS educators believe their grades to be bimodal. We showed 53 CS professors a series of histograms displaying ambiguous distributions and asked them to categorize the distributions. A random half of participants were primed to think about the fact that CS grades are commonly thought to be bimodal; these participants were more likely to label ambiguous distributions as "bimodal". Participants were also more likely to label distributions as bimodal if they believed that some students are innately predisposed to do better at CS. These results suggest that bimodal grades are instructional folklore in CS, caused by confirmation bias and instructor beliefs about their students.

[**Patra2021**] Jibesh Patra and Michael Pradel. Semantic bug seeding: a learning-based approach for creating realistic bugs. In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 8 2021, DOI 10.1145/3468264.3468623.
Abstract: When working on techniques to address the wide-spread problem of software bugs, one often faces the need for a large number of realistic bugs in real-world programs. Such bugs can either help evaluate an approach, e.g., in form of a bug benchmark or a suite of program mutations, or even help build the technique, e.g., in learning-based bug detection. Because

gathering a large number of real bugs is difficult, a common approach is to rely on automatically seeded bugs. Prior work seeds bugs based on syntactic transformation patterns, which often results in unrealistic bugs and typically cannot introduce new, application-specific code tokens. This paper presents SemSeed, a technique for automatically seeding bugs in a semantics-aware way. The key idea is to imitate how a given real-world bug would look like in other programs by semantically adapting the bug pattern to the local context. To reason about the semantics of pieces of code, our approach builds on learned token embeddings that encode the semantic similarities of identifiers and literals. Our evaluation with real-world JavaScript software shows that the approach effectively reproduces real bugs and clearly outperforms a semantics-unaware approach. The seeded bugs are useful as training data for learning-based bug detection, where they significantly improve the bug detection ability. Moreover, we show that SemSeed-created bugs complement existing mutation testing operators, and that our approach is efficient enough to seed hundreds of thousands of bugs within an hour.

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

[**Peng2021**] Yun Peng, Yu Zhang, and Mingzhe Hu. An empirical study for common language features used in Python projects. In *Proc. International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 3 2021, DOI `10.1109/saner50967.2021.00012`.

Abstract: As a dynamic programming language, Python is widely used in many fields. For developers, various language features affect programming experience. For researchers, they affect the difficulty of developing tasks such as bug finding and compilation optimization. Former research has shown that programs with Python dynamic features are more change-prone. However, we know little about the use and impact of Python language features in real-world Python projects. To resolve these issues, we systematically analyze Python language features and propose a tool named PYSCAN to automatically identify the use of 22 kinds of common Python language features in 6 categories in Python source code. We conduct an empirical study on 35 popular Python projects from eight application domains, covering over 4.3 million lines of code, to investigate the the usage of these language features in the project. We find that single inheritance, decorator, keyword argument, for loops and nested classes are top 5 used language features. Meanwhile different domains of projects may prefer some certain language features. For example, projects in DevOps use exception handling frequently. We also conduct in-depth manual analysis to dig extensive using patterns of frequently but differently used language features: exceptions, decorators and nested classes/functions. We find that developers care most about ImportError when handling exceptions. With the empirical results and in-depth analysis, we conclude with some suggestions and a discussion of implications for three groups of persons in Python community: Python designers, Python compiler designers and Python developers.

[**PerezDeRosso2013**] Santiago Perez De Rosso and Daniel Jackson. What's wrong with Git? In *Proc. Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (ONWARD)*. ACM Press, 2013, DOI `10.1145/2509578.2509584`.
Abstract: It is commonly asserted that the success of a software development project, and the usability of the final product, depend on the quality of the concepts that underlie its design. Yet this hypothesis has not been systematically explored by researchers, and conceptual design has not played the central role in the research and teaching of software engineering that one might expect. As part of a new research project to explore conceptual design, we are engaging in a series of case studies. This paper reports on the early stages of our first study, on the Git version control system. Despite its widespread adoption, Git puzzles even experienced developers and is not regarded as easy to use. In an attempt to understand the root causes of its complexity, we analyze its conceptual model and identify some undesirable properties; we then propose a reworking of the conceptual model that forms the basis of (the first version of) Gitless, an ongoing effort to redesign Git and experiment with the effects of conceptual simplifications.

[**PerezDeRosso2016**] Santiago Perez De Rosso and Daniel Jackson. Purposes, concepts, misfits, and a redesign of Git. In *Proc. International Conference on Object-Oriented Programming Systems Languages and Applications (OOP-SLA)*. ACM, 10 2016, DOI `10.1145/2983990.2984018`.

Abstract: Git is a widely used version control system that is powerful but complicated. Its complexity may not be an inevitable consequence of its power but rather evidence of flaws in its design. To explore this hypothesis, we analyzed the design of Git using a theory that identifies concepts, purposes, and misfits. Some well-known difficulties with Git are described, and explained as misfits in which underlying concepts fail to meet their intended purpose. Based on this analysis, we designed a reworking of Git (called Gitless) that attempts to remedy these flaws. To correlate misfits with issues reported by users, we conducted a study of Stack Overflow questions. And to determine whether users experienced fewer complications using Gitless in place of Git, we conducted a small user study. Results suggest our approach can be profitable in identifying, analyzing, and fixing design problems.

[**Petre2013**] Marian Petre. UML in practice. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2013, DOI `10.1109/icse.2013.6606618`.
Abstract: UML has been described by some as "the lingua franca of software engineering". Evidence from industry does not necessarily support such endorsements. How exactly is UML being used in industry—if it is? This paper presents a corpus of interviews with 50 professional software engineers in 50 companies and identifies 5 patterns of UML use.

[**Philip2012**] Kavita Philip, Medha Umarji, Megha Agarwala, Susan Elliott Sim, Rosalva Gallardo-Valencia, Cristina V. Lopes, and Sukanya Ratanotayanon. Software reuse through methodical component reuse and amethodical snippet remixing. In *Proc. Conference on Computer Supported Cooperative Work (CSCW)*. ACM Press, 2012, DOI `10.1145/2145204.2145407`.
Abstract: Every method for developing software is a prescriptive model. Applying a deconstructionist analysis to methods reveals that there are two texts, or sets of assumptions and ideals: a set that is privileged by the method and a second set that is left out, or marginalized by the method. We apply this analytical lens to software reuse, a technique in software development that seeks to expedite one's own project by using programming artifacts created by others. By analyzing the methods prescribed by Component-Based Software Engineering (CBSE), we arrive at two texts: Methodical CBSE and Amethodical Remixing. Empirical data from four studies on code search on the web draws attention to four key points of tension: status of component boundaries; provenance of source code; planning and process; and evaluation criteria for candidate code. We conclude the paper with a discussion of the implications of this work for the limits of methods, structure of organizations that reuse software, and the design of search engines for source code.

[**Piantadosi2020**] Valentina Piantadosi, Fabiana Fierro, Simone Scalabrino, Alexander Serebrenik, and Rocco Oliveto. How does code readability change during software evolution? *Empirical Software Engineering*, 25(6):5374–5412, 9 2020, DOI `10.1007/s10664-020-09886-9`.
Abstract: Code reading is one of the most frequent activities in software

maintenance. Such an activity aims at acquiring information from the code and, thus, it is a prerequisite for program comprehension: developers need to read the source code they are going to modify before implementing changes. As the code changes, so does its readability; however, it is not clear yet how code readability changes during software evolution. To understand how code readability changes when software evolves, we studied the history of 25 open source systems. We modeled code readability evolution by defining four states in which a file can be at a certain point of time (non-existing, other-name, readable, and unreadable). We used the data gathered to infer the probability of transitioning from one state to another one. In addition, we also manually checked a significant sample of transitions to compute the performance of the state-of-the-art readability prediction model we used to calculate the transition probabilities. With this manual analysis, we found that the tool correctly classifies all the transitions in the majority of the cases, even if there is a loss of accuracy compared to the single-version readability estimation. Our results show that most of the source code files are created readable. Moreover, we observed that only a minority of the commits change the readability state. Finally, we manually carried out qualitative analysis to understand what makes code unreadable and what developers do to prevent this. Using our results we propose some guidelines (i) to reduce the risk of code readability erosion and (ii) to promote best practices that make code readable.

[**Pietri2019**] Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. The software heritage graph dataset: public software development under one roof. In *Proc. International Conference on Mining Software Repositories (MSR)*. IEEE, 5 2019, DOI 10.1109/msr.2019.00030.

Abstract: Software Heritage is the largest existing public archive of software source code and accompanying development history: it currently spans more than five billion unique source code files and one billion unique commits, coming from more than 80 million software projects. This paper introduces the Software Heritage graph dataset: a fully-deduplicated Merkle DAG representation of the Software Heritage archive. The dataset links together file content identifiers, source code directories, Version Control System (VCS) commits tracking evolution over time, up to the full states of VCS repositories as observed by Software Heritage during periodic crawls. The dataset's contents come from major development forges (including GitHub and GitLab), FOSS distributions (e.g., Debian), and language-specific package managers (e.g., PyPI). Crawling information is also included, providing timestamps about when and where all archived source code artifacts have been observed in the wild. The Software Heritage graph dataset is available in multiple formats, including downloadable CSV dumps and Apache Parquet files for local use, as well as a public instance on Amazon Athena interactive query service for ready-to-use powerful analytical processing. Source code file contents are cross-referenced at the graph leaves, and can be retrieved through individual requests using the Software Heritage archive API.

[**Pizard2021**] Sebastián Pizard, Fernando Acerenza, Ximena Otegui, Silvana Moreno, Diego Vallespir, and Barbara Kitchenham. Training students in evidence-based software engineering and systematic reviews: a systematic review and empirical study. *Empirical Software Engineering*, 26(3), 3 2021, DOI `10.1007/s10664-021-09953-9`.

Abstract: Context Although influential in academia, evidence-based software engineering (EBSE) has had little impact on industry practice. We found that other disciplines have identified lack of training as a significant barrier to Evidence-Based Practice. Objective To build and assess an EBSE training proposal suitable for students with more than 3 years of computer science/software engineering university-level training. Method We performed a systematic literature review (SLR) of EBSE teaching initiatives and used the SLR results to help us to develop and evaluate an EBSE training proposal. The course was based on the theory of learning outcomes and incorporated a large practical content related to performing an SLR. We ran the course with 10 students and based course evaluation on student performance and opinions of both students and teachers. We assessed knowledge of EBSE principles from the mid-term and final tests, as well as evaluating the SLRs produced by the student teams. We solicited student opinions about the course and its value via a student survey, a team survey, and a focus group. The teachers' viewpoint was collected in a debriefing meeting. Results Our SLR identified 14 relevant primary studies. The primary studies emphasized the importance of practical examples (usually based on the SLR process) and used a variety of evaluation methods, but lacked any formal education methodology. We identified 54 learning outcomes covering aspects of EBSE and the SLR method. All 10 students passed the course. Our course evaluation showed that a large percentage of the learning outcomes established for training were accomplished. Conclusions The course proved suitable for students to understand the EBSE paradigm and to be able to apply it to a limited-scope practical assignment. Our learning outcomes, course structure, and course evaluation process should help to improve the effectiveness and comparability of future studies of EBSE training. However, future courses should increase EBSE training related to the use of SLR results.

[**Porter2013**] Leo Porter, Cynthia Bailey Lee, and Beth Simon. Halving fail rates using peer instruction. In *Proc. Technical Symposium on Computer Science Education (SIGCSE)*. ACM Press, 2013, DOI `10.1145/2445196.2445250`.

Abstract: Peer Instruction (PI) is a teaching method that supports student-centric classrooms, where students construct their own understanding through a structured approach featuring questions with peer discussions. PI has been shown to increase learning in STEM disciplines such as physics and biology. In this report we look at another indicator of student success the rate at which students pass the course or, conversely, the rate at which they fail. Evaluating 10 years of instruction of 4 different courses spanning 16 PI course instances, we find that adoption of the PI methodology in the

classroom reduces fail rates by a per-course average of 61% (20% reduced to 7%) compared to standard instruction (SI). Moreover, we also find statistically significant improvements within-instructor. For the same instructor teaching the same course, we find PI decreases the fail rate, on average, by 67% (from 23% to 8%) compared to SI. As an in-situ study, we discuss the various threats to the validity of this work and consider implications of wide-spread adoption of PI in computing programs.

[**Posnett2011**] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. Got issues? do new features and code improvements affect defects? In *Proc. Working Conference on Reverse Engineering (WCRE)*. IEEE, 10 2011, DOI `10.1109/wcre.2011.33`.

Abstract: There is a perception that when new features are added to a system that those added and modified parts of the source-code are more fault prone. Many have argued that new code and new features are defect prone due to immaturity, lack of testing, as well unstable requirements. Unfortunately most previous work does not investigate the link between a concrete requirement or new feature and the defects it causes, in particular the feature, the changed code and the subsequent defects are rarely investigated. In this paper we investigate the relationship between improvements, new features and defects recorded within an issue tracker. A manual case study is performed to validate the accuracy of these issue types. We combine defect issues and new feature issues with the code from version-control systems that introduces these features, we then explore the relationship of new features with the fault-proneness of their implementations. We describe properties and produce models of the relationship between new features and fault proneness, based on the analysis of issue trackers and version-control systems. We find, surprisingly, that neither improvements nor new features have any significant effect on later defect counts, when controlling for size and total number of changes.

[**Prabhu2011**] Prakash Prabhu, Yun Zhang, Soumyadeep Ghosh, David I. August, Jialu Huang, Stephen Beard, Hanjun Kim, Taewook Oh, Thomas B. Jablin, Nick P. Johnson, Matthew Zoufaly, Arun Raman, Feng Liu, and David Walker. A survey of the practice of computational science. In *Proc. Supercomputing*. ACM Press, 2011, DOI `10.1145/2063348.2063374`.

Abstract: Computing plays an indispensable role in scientific research. Presently, researchers in science have different problems, needs, and beliefs about computation than professional programmers. In order to accelerate the progress of science, computer scientists must understand these problems, needs, and beliefs. To this end, this paper presents a survey of scientists from diverse disciplines, practicing computational science at a doctoral-granting university with very high re search activity. The survey covers many things, among them, prevalent programming practices within this scientific community, the importance of computational power in different fields, use of tools to enhance performance and soft ware productivity, computational resources leveraged, and prevalence of parallel computation. The results reveal several

patterns that suggest interesting avenues to bridge the gap between scientific researchers and programming tools developers.

[**Prana2018**] Gede Artha Azriadi Prana, Christoph Treude, Ferdian Thung, Thushari Atapattu, and David Lo. Categorizing the content of GitHub README files. *Empirical Software Engineering*, 24(3):1296–1327, 10 2018, DOI `10.1007/s10664-018-9660-3`.

Abstract: README files play an essential role in shaping a developer's first impression of a software repository and in documenting the software project that the repository hosts. Yet, we lack a systematic understanding of the content of a typical README file as well as tools that can process these files automatically. To close this gap, we conduct a qualitative study involving the manual annotation of 4,226 README file sections from 393 randomly sampled GitHub repositories and we design and evaluate a classifier and a set of features that can categorize these sections automatically. We find that information discussing the 'What' and 'How' of a repository is very common, while many README files lack information regarding the purpose and status of a repository. Our multi-label classifier which can predict eight different categories achieves an F1 score of 0.746. To evaluate the usefulness of the classification, we used the automatically determined classes to label sections in GitHub README files using badges and showed files with and without these badges to twenty software professionals. The majority of participants perceived the automated labeling of sections based on our classifier to ease information discovery. This work enables the owners of software repositories to improve the quality of their documentation and it has the potential to make it easier for the software development community to discover relevant information in GitHub README files.

[**Pritchard2015**] David Pritchard. Frequency distribution of error messages. In *Proc. Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*. ACM, 10 2015, DOI `10.1145/2846680.2846681`.

Abstract: Which programming error messages are the most common? We investigate this question, motivated by writing error explanations for novices. We consider large data sets in Python and Java that include both syntax and run-time errors. In both data sets, after grouping essentially identical messages, the error message frequencies empirically resemble Zipf-Mandelbrot distributions. We use a maximum-likelihood approach to fit the distribution parameters. This gives one possible way to contrast languages or compilers quantitatively.

[**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 ad-

vancement 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.

[**Racheva2010**] Zornitza Racheva, Maya Daneva, Klaas Sikkel, Andrea Herrmann, and Roel Wieringa. Do we know enough about requirements prioritization in agile projects: insights from a case study. In *Proc. International Requirements Engineering Conference (RE)*. IEEE, 9 2010, DOI `10.1109/re.2010.27`.
Abstract: Requirements prioritization is an essential mechanism of agile software development approaches. It maximizes the value delivered to the clients and accommodates changing requirements. This paper presents results of an exploratory cross-case study on agile prioritization and business value delivery processes in eight software organizations. We found that some explicit and fundamental assumptions of agile requirement prioritization approaches, as described in the agile literature on best practices, do not hold in all agile project contexts in our study. These are (i) the driving role of the client in the value creation process, (ii) the prevailing position of business value as a main prioritization criterion, (iii) the role of the prioritization process for project goal achievement. This implies that these assumptions have to be reframed and that the approaches to requirements prioritization for value creation need to be extended.

[**Ragkhitwetsagul2021**] Chaiyong Ragkhitwetsagul, Jens Krinke, Matheus Paixao, Giuseppe Bianco, and Rocco Oliveto. Toxic code snippets on Stack Overflow. *IEEE Transactions on Software Engineering*, 47(3):560–581, 3 2021, DOI `10.1109/tse.2019.2900307`.
Abstract: Online code clones are code fragments that are copied from software projects or online sources to Stack Overflow as examples. Due to an absence of a checking mechanism after the code has been copied to Stack Overflow, they can become toxic code snippets, e.g., they suffer from being outdated or violating the original software license. We present a study of online code clones on Stack Overflow and their toxicity by incorporating two developer surveys and a large-scale code clone detection. A survey of 201 high-reputation Stack Overflow answerers (33 percent response rate) showed that 131 participants (65 percent) have ever been notified of outdated code and 26 of them (20 percent) rarely or never fix the code. 138 answerers (69

percent) never check for licensing conflicts between their copied code snippets and Stack Overflow's CC BY-SA 3.0. A survey of 87 Stack Overflow visitors shows that they experienced several issues from Stack Overflow answers: mismatched solutions, outdated solutions, incorrect solutions, and buggy code. 85 percent of them are not aware of CC BY-SA 3.0 license enforced by Stack Overflow, and 66 percent never check for license conflicts when reusing code snippets. Our clone detection found online clone pairs between 72,365 Java code snippets on Stack Overflow and 111 open source projects in the curated Qualitas corpus. We analysed 2,289 non-trivial online clone candidates. Our investigation revealed strong evidence that 153 clones have been copied from a Qualitas project to Stack Overflow. We found 100 of them (66 percent) to be outdated, of which 10 were buggy and harmful for reuse. Furthermore, we found 214 code snippets that could potentially violate the license of their original software and appear 7,112 times in 2,427 GitHub projects.

[**Rahman2011**] Foyzur Rahman and Premkumar Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 5 2011, DOI `10.1145/1985793.1985860`.
Abstract: Recent research indicates that "people" factors such as ownership, experience, organizational structure, and geographic distribution have a big impact on software quality. Understanding these factors, and properly deploying people resources can help managers improve quality outcomes. This paper considers the impact of code ownership and developer experience on software quality. In a large project, a file might be entirely owned by a single developer, or worked on by many. Some previous research indicates that more developers working on a file might lead to more defects. Prior research considered this phenomenon at the level of modules or files, and thus does not tease apart and study the effect of contributions of different developers to each module or file. We exploit a modern version control system to examine this issue at a fine-grained level. Using version history, we examine contributions to code fragments that are actually repaired to fix bugs. Are these code fragments "implicated" in bugs the result of contributions from many? or from one? Does experience matter? What type of experience? We find that implicated code is more strongly associated with a single developer's contribution; our findings also indicate that an author's specialized experience in the target file is more important than general experience. Our findings suggest that quality control efforts could be profitably targeted at changes made by single developers with limited prior experience on that file.

[**Rahman2013**] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2013, DOI `10.1109/icse.2013.6606589`.
Abstract: Defect prediction techniques could potentially help us to focus quality-assurance efforts on the most defect-prone files. Modern statistical

tools make it very easy to quickly build and deploy prediction models. Software metrics are at the heart of prediction models; understanding how and especially why different types of metrics are effective is very important for successful model deployment. In this paper we analyze the applicability and efficacy of process and code metrics from several different perspectives. We build many prediction models across 85 releases of 12 large open source projects to address the performance, stability, portability and stasis of different sets of metrics. Our results suggest that code metrics, despite widespread use in the defect prediction literature, are generally less useful than process metrics for prediction. Second, we find that code metrics have high stasis; they don't change very much from release to release. This leads to stagnation in the prediction models, leading to the same files being repeatedly predicted as defective; unfortunately, these recurringly defective files turn out to be comparatively less defect-dense.

[**Rahman2020a**] Akond Rahman, Effat Farhana, Chris Parnin, and Laurie Williams. Gang of eight: a defect taxonomy for infrastructure as code scripts. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 6 2020, DOI `10.1145/3377811.3380409`.
Abstract: Defects in infrastructure as code (IaC) scripts can have serious consequences, for example, creating large-scale system outages. A taxonomy of IaC defects can be useful for understanding the nature of defects, and identifying activities needed to fix and prevent defects in IaC scripts. The goal of this paper is to help practitioners improve the quality of infrastructure as code (IaC) scripts by developing a defect taxonomy for IaC scripts through qualitative analysis. We develop a taxonomy of IaC defects by applying qualitative analysis on 1,448 defect-related commits collected from open source software (OSS) repositories of the Openstack organization. We conduct a survey with 66 practitioners to assess if they agree with the identified defect categories included in our taxonomy. We quantify the frequency of identified defect categories by analyzing 80,425 commits collected from 291 OSS repositories spanning across 2005 to 2019. Our defect taxonomy for IaC consists of eight categories, including a category specific to IaC called idempotency (i.e., defects that lead to incorrect system provisioning when the same IaC script is executed multiple times). We observe the surveyed 66 practitioners to agree most with idempotency. The most frequent defect category is configuration data i.e., providing erroneous configuration data in IaC scripts. Our taxonomy and the quantified frequency of the defect categories may help in advancing the science of IaC script quality.

[**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 counterproductive 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 Mi-

lanova, 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.

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

Abstract: Background: Statistical concepts and techniques are often applied incorrectly, even in mature disciplines such as medicine or psychology. Surprisingly, there are very few works that study statistical problems in software engineering (SE). Aim: Assess the existence of statistical errors in SE experiments. Method: Compile the most common statistical errors in experimental disciplines. Survey experiments published in ICSE to assess whether errors occur in high quality SE publications. Results: The same errors as identified in others disciplines were found in ICSE experiments, where 30 of the reviewed papers included several error types such as: a) missing statistical hypotheses, b) missing sample size calculation, c) failure to assess statistical test assumptions, and d) uncorrected multiple testing. This rather large error rate is greater for research papers where experiments are confined to the validation section. The origin of the errors can be traced back to: a) researchers not having sufficient statistical training, and b) a profusion of exploratory research. Conclusions: This paper provides preliminary evidence that SE research suffers from the same statistical problems as other experimental disciplines. However, the SE community appears to be unaware of any shortcomings in its experiments, whereas other disciplines work hard to avoid these threats. Further research is necessary to find the underlying causes and set up corrective measures, but there are some poten-

tially effective actions and are a priori easy to implement: a) improve the statistical training of SE researchers, and b) enforce quality assessment and reporting guidelines in SE publications.

[**Rico2021**] Sergio Rico, Elizabeth Bjarnason, Emelie Engström, Martin Höst, and Per Runeson. A case study of industry–academia communication in a joint software engineering research project. *Journal of Software: Evolution and Process*, 7 2021, DOI `10.1002/smr.2372`.

Abstract: Empirical software engineering research relies on good communication with industrial partners. Conducting joint research both requires and contributes to bridging the communication gap between industry and academia (IA) in software engineering. This study aims to explore communication between the two parties in such a setting. To better understand what facilitates good IA communication and what project outcomes such communication promotes, we performed a case study, in the context of a long-term IA joint project, followed by a validating survey among practitioners and researchers with experience of working in similar settings. We identified five facilitators of IA communication and nine project outcomes related to this communication. The facilitators concern the relevance of the research, practitioners' attitude and involvement in research, frequency of communication and longevity of the collaboration. The project outcomes promoted by this communication include, for researchers, changes in teaching and new scientific venues, and for practitioners, increased awareness, changes to practice, and new tools and source code. Besides, both parties gain new knowledge and develop social-networks through IA communication. Our study presents empirically based insights that can provide advise on how to improve communication in IA research projects and thus the co-creation of software engineering knowledge that is anchored in both practice and research.

[**Rigby2011**] Peter C. Rigby and Margaret-Anne Storey. Understanding broadcast based peer review on open source software projects. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 5 2011, DOI `10.1145/1985793.1985867`.

Abstract: Software peer review has proven to be a successful technique in open source software (OSS) development. In contrast to industry, where reviews are typically assigned to specific individuals, changes are broadcast to hundreds of potentially interested stakeholders. Despite concerns that reviews may be ignored, or that discussions will deadlock because too many uninformed stakeholders are involved, we find that this approach works well in practice. In this paper, we describe an empirical study to investigate the mechanisms and behaviours that developers use to find code changes they are competent to review. We also explore how stakeholders interact with one another during the review process. We manually examine hundreds of reviews across five high profile OSS projects. Our findings provide insights into the simple, community-wide techniques that developers use to effectively

manage large quantities of reviews. The themes that emerge from our study are enriched and validated by interviewing long-serving core developers.

[**Rigby2016**] Peter C. Rigby, Yue Cai Zhu, Samuel M. Donadelli, and Audris Mockus. Quantifying and mitigating turnover-induced knowledge loss. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 5 2016, DOI 10.1145/2884781.2884851.
Abstract: The utility of source code, as of other knowledge artifacts, is predicated on the existence of individuals skilled enough to derive value by using or improving it. Developers leaving a software project deprive the project of the knowledge of the decisions they have made. Previous research shows that the survivors and newcomers maintaining abandoned code have reduced productivity and are more likely to make mistakes. We focus on quantifying the extent of abandoned source files and adapt methods from financial risk analysis to assess the susceptibility of the project to developer turnover. In particular, we measure the historical loss distribution and find (1) that projects are susceptible to losses that are more than three times larger than the expected loss. Using historical simulations we find (2) that projects are susceptible to large losses that are over five times larger than the expected loss. We use Monte Carlo simulations of disaster loss scenarios and find (3) that simplistic estimates of the "truck factor" exaggerate the potential for loss. To mitigate loss from developer turnover, we modify Cataldo et al's coordination requirements matrices. We find (4) that we can recommend the correct successor 34% to 48% of the time. We also find that having successors reduces the expected loss by as much as 15%. Our approach helps large projects assess the risk of turnover thereby making risk more transparent and manageable.

[**Rigger2020**] Manuel Rigger and Zhendong Su. Finding bugs in database systems via query partitioning. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 11 2020, DOI 10.1145/3428279.
Abstract: Logic bugs in Database Management Systems (DBMSs) are bugs that cause an incorrect result for a given query, for example, by omitting a row that should be fetched. These bugs are critical, since they are likely to go unnoticed by users. We propose Query Partitioning, a general and effective approach for finding logic bugs in DBMSs. The core idea of Query Partitioning is to, starting from a given original query, derive multiple, more complex queries (called partitioning queries), each of which computes a partition of the result. The individual partitions are then composed to compute a result set that must be equivalent to the original query's result set. A bug in the DBMS is detected when these result sets differ. Our intuition is that due to the increased complexity, the partitioning queries are more likely to stress the DBMS and trigger a logic bug than the original query. As a concrete instance of a partitioning strategy, we propose Ternary Logic Partitioning (TLP), which is based on the observation that a boolean predicate p can either evaluate to TRUE, FALSE, or NULL. Accordingly, a query can be

decomposed into three partitioning queries, each of which computes its result on rows or intermediate results for which p, NOT p, and p IS NULL hold. This technique is versatile, and can be used to test WHERE, GROUP BY, as well as HAVING clauses, aggregate functions, and DISTINCT queries. As part of an extensive testing campaign, we found 175 bugs in widely-used DBMSs such as MySQL, TiDB, SQLite, and CockroachDB, 125 of which have been fixed. Notably, 77 of these were logic bugs, while the remaining were error and crash bugs. We expect that the effectiveness and wide applicability of Query Partitioning will lead to its broad adoption in practice, and the formulation of additional partitioning strategies.

[**Rivers2016**] Kelly Rivers, Erik Harpstead, and Ken Koedinger. Learning curve analysis for programming. In *Proc. Conference on International Computing Education Research (ICER)*. ACM, 8 2016, DOI `10.1145/2960310.2960333`.
Abstract: The recent surge in interest in using educational data mining on student written programs has led to discoveries about which compiler errors students encounter while they are learning how to program. However, less attention has been paid to the actual code that students produce. In this paper, we investigate programming data by using learning curve analysis to determine which programming elements students struggle with the most when learning in Python. Our analysis extends the traditional use of learning curve analysis to include less structured data, and also reveals new possibilities for when to teach students new programming concepts. One particular discovery is that while we find evidence of student learning in some cases (for example, in function definitions and comparisons), there are other programming elements which do not demonstrate typical learning. In those cases, we discuss how further changes to the model could affect both demonstrated learning and our understanding of the different concepts that students learn.

[**Robillard2010**] Martin P. Robillard and Rob DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 12 2010, DOI `10.1007/s10664-010-9150-8`.
Abstract: Large APIs can be hard to learn, and this can lead to decreased programmer productivity. But what makes APIs hard to learn? We conducted a mixed approach, multi-phased study of the obstacles faced by Microsoft developers learning a wide variety of new APIs. The study involved a combination of surveys and in-person interviews, and collected the opinions and experiences of over 440 professional developers. We found that some of the most severe obstacles faced by developers learning new APIs pertained to the documentation and other learning resources. We report on the obstacles developers face when learning new APIs, with a special focus on obstacles related to API documentation. Our qualitative analysis elicited five important factors to consider when designing API documentation: documentation of intent; code examples; matching APIs with scenarios; penetrability of the API; and format and presentation. We analyzed how these factors can be interpreted to prioritize API documentation development efforts

[**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.

[**RodriguezPerez2018**] Gema Rodríguez-Pérez, Gregorio Robles, and Jesús M. González-Barahona. Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the SZZ algorithm. *Information and Software Technology*, 99:164–176, 7 2018, DOI `10.1016/j.infsof.2018.03.009`.
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 as-

sumption, "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.

[**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.

[**Rossbach2010**] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? *ACM SIGPLAN Notices*, 45(5):47–56, 5 2010, DOI 10.1145/1837853.1693462.

Abstract: Chip multi-processors (CMPs) have become ubiquitous, while tools that ease concurrent programming have not. The promise of increased performance for all applications through ever more parallel hardware requires good tools for concurrent programming, especially for average programmers. Transactional memory (TM) has enjoyed recent interest as a tool that can help programmers program concurrently. The transactional memory (TM) research community is heavily invested in the claim that programming with transactional memory is easier than alternatives (like locks), but evidence for or against the veracity of this claim is scant. In this paper, we describe a user-study in which 237 undergraduate students in an operating systems course implement the same programs using coarse and fine-grain locks, monitors, and transactions. We surveyed the students after the assignment, and examined their code to determine the types and frequency of programming errors for each synchronization technique. Inexperienced programmers found baroque syntax a barrier to entry for transactional programming. On average, subjective evaluation showed that students found transactions harder to use than coarse-grain locks, but slightly easier to use than fine-grained locks. Detailed examination of synchronization errors in the students' code tells a rather different story. Overwhelmingly, the number and types of programming errors the students made was much lower for transactions than for locks. On a similar programming problem, over 70% of students made errors with fine-grained locking, while less than 10% made errors with transactions.

[**Russo2020**] Daniel Russo and Klaas-Jan Stol. Gender differences in personality traits of software engineers. *IEEE Transactions on Software Engineering*, pages 1–1, 2020, DOI 10.1109/tse.2020.3003413.

Abstract: There is a growing body of gender studies in software engineering to understand diversity and inclusion issues, as diversity is recognized to be a key issue to healthy teams and communities. A second factor often

linked to team performance is personality, which has received far more attention. Very few studies, however, have focused on the intersection of these two fields. Hence, we set out to study gender differences in personality traits of software engineers. Through a survey study we collected personality data, using the HEXACO model, of 483 software engineers. The data were analyzed using a Bayesian independent sample t-test and network analysis. The results suggest that women score significantly higher in Openness to Experience, Honesty-Humility, and Emotionality than men. Further, men show higher psychopathic traits than women. Based on these findings, we develop a number of propositions that can guide future research.

[**Sadowski2019**] Caitlin Sadowski and Thomas Zimmermann, editors. *Rethinking Productivity in Software Engineering*. Apress, 2019.
Abstract: This open access book collects the wisdom of the 2017 Dagstuhl seminar on productivity in software engineering, a meeting of community leaders, who came together with the goal of rethinking traditional definitions and measures of productivity. The results of their work, Rethinking Productivity in Software Engineering, includes chapters covering definitions and core concepts related to productivity, guidelines for measuring productivity in specific contexts, best practices and pitfalls, and theories and open questions on productivity. You'll benefit from the many short chapters, each offering a focused discussion on one aspect of productivity in software engineering.

[**Sambasivan2021**] Nithya Sambasivan, Shivani Kapania, Hannah Highfill, Diana Akrong, Praveen Paritosh, and Lora M Aroyo. "everyone wants to do the model work, not the data work": Data cascades in high-stakes AI. In *Proc. Conference on Human Factors in Computing Systems (HFCS)*. ACM, 5 2021, DOI 10.1145/3411764.3445518.
Abstract: AI models are increasingly applied in high-stakes domains like health and conservation. Data quality carries an elevated significance in high-stakes AI due to its heightened downstream impact, impacting predictions like cancer detection, wildlife poaching, and loan allocations. Paradoxically, data is the most under-valued and de-glamorised aspect of AI. In this paper, we report on data practices in high-stakes AI, from interviews with 53 AI practitioners in India, East and West African countries, and USA. We define, identify, and present empirical evidence on Data Cascades—compounding events causing negative, downstream effects from data issues—triggered by conventional AI/ML practices that undervalue data quality. Data cascades are pervasive (92% prevalence), invisible, delayed, but often avoidable. We discuss HCI opportunities in designing and incentivizing data excellence as a first-class citizen of AI, resulting in safer and more robust systems for all.

[**Sarker2019**] Farhana Sarker, Bogdan Vasilescu, Kelly Blincoe, and Vladimir Filkov. Socio-technical work-rate increase associates with changes in work patterns in online projects. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2019, DOI 10.1109/icse.2019.00099.

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

[**Scalabrino2018**] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. A comprehensive model for code readability. *Journal of Software: Evolution and Process*, 30(6):e1958, 6 2018, DOI `10.1002/smr.1958`.
Abstract: Unreadable code could compromise program comprehension, and it could cause the introduction of bugs. Code consists of mostly natural language text, both in identifiers and comments, and it is a particular form of text. Nevertheless, the models proposed to estimate code readability take into account only structural aspects and visual nuances of source code, such as line length and alignment of characters. In this paper, we extend our previous work in which we use textual features to improve code readability models. We introduce 2 new textual features, and we reassess the readability prediction power of readability models on more than 600 code snippets manually evaluated, in terms of readability, by 5K+ people. We also replicate a study by Buse and Weimer on the correlation between readability and FindBugs warnings, evaluating different models on 20 software systems, for a total of 3M lines of code. The results demonstrate that (1) textual features complement other features and (2) a model containing all the features achieves a significantly higher accuracy as compared with all the other state-of-the-art models. Also, readability estimation resulting from a more accurate model, ie, the combined model, is able to predict more accurately FindBugs warnings.

[**Scalabrino2021**] Simone Scalabrino, Gabriele Bavota, Christopher Vendome,

Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. Automatically assessing code understandability. *IEEE Transactions on Software Engineering*, 47(3):595–613, 3 2021, DOI `10.1109/tse.2019.2901468`.

Abstract: Understanding software is an inherent requirement for many maintenance and evolution tasks. Without a thorough understanding of the code, developers would not be able to fix bugs or add new features timely. Measuring code understandability might be useful to guide developers in writing better code, and could also help in estimating the effort required to modify code components. Unfortunately, there are no metrics designed to assess the understandability of code snippets. In this work, we perform an extensive evaluation of 121 existing as well as new code-related, documentation-related, and developer-related metrics. We try to (i) correlate each metric with understandability and (ii) build models combining metrics to assess understandability. To do this, we use 444 human evaluations from 63 developers and we obtained a bold negative result: none of the 121 experimented metrics is able to capture code understandability, not even the ones assumed to assess quality attributes apparently related, such as code readability and complexity. While we observed some improvements while combining metrics in models, their effectiveness is still far from making them suitable for practical applications. Finally, we conducted interviews with five professional developers to understand the factors that influence their ability to understand code snippets, aiming at identifying possible new metrics.

[**Scanniello2017**] Giuseppe Scanniello, Michele Risi, Porfirio Tramontana, and Simone Romano. Fixing faults in C and Java source code. *ACM Transactions on Software Engineering and Methodology*, 26(2):1–43, 10 2017, DOI `10.1145/3104029`.

Abstract: We carried out a family of controlled experiments to investigate whether the use of abbreviated identifier names, with respect to full-word identifier names, affects fault fixing in C and Java source code. This family consists of an original (or baseline) controlled experiment and three replications. We involved 100 participants with different backgrounds and experiences in total. Overall results suggested that there is no difference in terms of effort, effectiveness, and efficiency to fix faults, when source code contains either only abbreviated or only full-word identifier names. We also conducted a qualitative study to understand the values, beliefs, and assumptions that inform and shape fault fixing when identifier names are either abbreviated or full-word. We involved in this qualitative study six professional developers with 1–3 years of work experience. A number of insights emerged from this qualitative study and can be considered a useful complement to the quantitative results from our family of experiments. One of the most interesting insights is that developers, when working on source code with abbreviated identifier names, adopt a more methodical approach to identify and fix faults by extending their focus point and only in a few cases do they expand abbreviated identifiers.

[**Schweinsberg2021**] Martin Schweinsberg, Michael Feldman, Nicola Staub,

Olmo R. van den Akker, Robbie C.M. van Aert, Marcel A.L.M. van Assen, Yang Liu, Tim Althoff, Jeffrey Heer, Alex Kale, Zainab Mohamed, Hashem Amireh, Vaishali Venkatesh Prasad, Abraham Bernstein, Emily Robinson, Kaisa Snellman, S. Amy Sommer, Sarah M.G. Otner, David Robinson, Nikhil Madan, Raphael Silberzahn, Pavel Goldstein, Warren Tierney, Toshio Murase, Benjamin Mandl, Domenico Viganola, Carolin Strobl, Catherine B.C. Schaumans, Stijn Kelchtermans, Chan Naseeb, S. Mason Garrison, Tal Yarkoni, C.S. Richard Chan, Prestone Adie, Paulius Alaburda, Casper Albers, Sara Alspaugh, Jeff Alstott, Andrew A. Nelson, Eduardo Ariño de la Rubia, Adbi Arzi, Štěpán Bahník, Jason Baik, Laura Winther Balling, Sachin Banker, David AA Baranger, Dale J. Barr, Brenda Barros-Rivera, Matt Bauer, Enuh Blaise, Lisa Boelen, Katerina Bohle Carbonell, Robert A. Briers, Oliver Burkhard, Miguel-Angel Canela, Laura Castrillo, Timothy Catlett, Olivia Chen, Michael Clark, Brent Cohn, Alex Coppock, Natàlia Cugueró-Escofet, Paul G. Curran, Wilson Cyrus-Lai, David Dai, Giulio Valentino Dalla Riva, Henrik Danielsson, Rosaria de F.S.M. Russo, Niko de Silva, Curdin Derungs, Frank Dondelinger, Carolina Duarte de Souza, B. Tyson Dube, Marina Dubova, Ben Mark Dunn, Peter Adriaan Edelsbrunner, Sara Finley, Nick Fox, Timo Gnambs, Yuanyuan Gong, Erin Grand, Brandon Greenawalt, Dan Han, Paul H.P. Hanel, Antony B. Hong, David Hood, Justin Hsueh, Lilian Huang, Kent N. Hui, Keith A. Hultman, Azka Javaid, Lily Ji Jiang, Jonathan Jong, Jash Kamdar, David Kane, Gregor Kappler, Erikson Kaszubowski, Christopher M. Kavanagh, Madian Khabsa, Bennett Kleinberg, Jens Kouros, Heather Krause, Angelos-Miltiadis Krypotos, Dejan Lavbič, Rui Ling Lee, Timothy Leffel, Wei Yang Lim, Silvia Liverani, Bianca Loh, Dorte Lønsmann, Jia Wei Low, Alton Lu, Kyle MacDonald, Christopher R. Madan, Lasse Hjorth Madsen, Christina Maimone, Alexandra Mangold, Adrienne Marshall, Helena Ester Matskewich, Kimia Mavon, Katherine L. McLain, Amelia A. McNamara, Mhairi McNeill, Ulf Mertens, David Miller, Ben Moore, Andrew Moore, Eric Nantz, Ziauddin Nasrullah, Valentina Nejkovic, Colleen S Nell, Andrew Arthur Nelson, Gustav Nilsonne, Rory Nolan, Christopher E. O'Brien, Patrick O'Neill, Kieran O'Shea, Toto Olita, Jahna Otterbacher, Diana Palsetia, Bianca Pereira, Ivan Pozdniakov, John Protzko, Jean-Nicolas Reyt, Travis Riddle, Amal (Akmal) Ridhwan Omar Ali, Ivan Ropovik, Joshua M. Rosenberg, Stephane Rothen, Michael Schulte-Mecklenbeck, Nirek Sharma, Gordon Shotwell, Martin Skarzynski, William Stedden, Victoria Stodden, Martin A. Stoffel, Scott Stoltzman, Subashini Subbaiah, Rachael Tatman, Paul H. Thibodeau, Sabina Tomkins, Ana Valdivia, Gerrieke B. Druijff van de Woestijne, Laura Viana, Florence Villesèche, W. Duncan Wadsworth, Florian Wanders, Krista Watts, Jason D Wells, Christopher E. Whelpley, Andy Won, Lawrence Wu, Arthur Yip, Casey Youngflesh, Ju-Chi Yu, Arash Zandian, Leilei Zhang, Chava Zibman, and Eric Luis Uhlmann. Same data, different conclusions: Radical dispersion in empirical results when independent analysts operationalize and test the same hypothesis. *Organizational Behavior and Human Decision Processes*, 165:228–249, 7 2021, DOI

Abstract: In this crowdsourced initiative, independent analysts used the same dataset to test two hypotheses regarding the effects of scientists' gender and professional status on verbosity during group meetings. Not only the analytic approach but also the operationalizations of key variables were left unconstrained and up to individual analysts. For instance, analysts could choose to operationalize status as job title, institutional ranking, citation counts, or some combination. To maximize transparency regarding the process by which analytic choices are made, the analysts used a platform we developed called DataExplained to justify both preferred and rejected analytic paths in real time. Analyses lacking sufficient detail, reproducible code, or with statistical errors were excluded, resulting in 29 analyses in the final sample. Researchers reported radically different analyses and dispersed empirical outcomes, in a number of cases obtaining significant effects in opposite directions for the same research question. A Boba multiverse analysis demonstrates that decisions about how to operationalize variables explain variability in outcomes above and beyond statistical choices (e.g., covariates). Subjective researcher decisions play a critical role in driving the reported empirical results, underscoring the need for open data, systematic robustness checks, and transparency regarding both analytic paths taken and not taken. Implications for organizations and leaders, whose decision making relies in part on scientific findings, consulting reports, and internal analyses by data scientists, are discussed.

[**Sedano2017**] Todd Sedano, Paul Ralph, and Cecile Peraire. Software development waste. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2017, DOI `10.1109/icse.2017.20`.
Abstract: Context: Since software development is a complex sociotechnical activity that involves coordinating different disciplines and skill sets, it provides ample opportunities for waste to emerge. Waste is any activity that produces no value for the customer or user. Objective: The purpose of this paper is to identify and describe different types of waste in software development. Method: Following Constructivist Grounded Theory, we conducted a two-year five-month participant-observation study of eight software development projects at Pivotal, a software development consultancy. We also interviewed 33 software engineers, interaction designers, and product managers, and analyzed one year of retrospection topics. We iterated between analysis and theoretical sampling until achieving theoretical saturation. Results: This paper introduces the first empirical waste taxonomy. It identifies nine wastes and explores their causes, underlying tensions, and overall relationship to the waste taxonomy found in Lean Software Development. Limitations: Grounded Theory does not support statistical generalization. While the proposed taxonomy appears widely applicable, organizations with different software development cultures may experience different waste types. Conclusion: Software development projects manifest nine types of waste: building the wrong feature or product, mismanaging the

backlog, rework, unnecessarily complex solutions, extraneous cognitive load, psychological distress, waiting/multitasking, knowledge loss, and ineffective communication.

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

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

Abstract: A sound Decision-Making (DM) process is key to the successful governance of software projects. In many Open Source Software Development (OSSD) communities, DM processes lie buried amongst vast amounts of publicly available data. Hidden within this data lie the rationale for decisions that led to the evolution and maintenance of software products. While there have been some efforts to extract DM processes from publicly available data, the rationale behind 'how' the decisions are made have seldom been explored. Extracting the rationale for these decisions can facilitate transparency (by making them known), and also promote accountability on the part of decision-makers. This work bridges this gap by means of a large-scale study that unearths the rationale behind decisions from Python development email archives comprising about 1.5 million emails. This paper makes two main contributions. First, it makes a knowledge contribution by

unearthing and presenting the rationale behind decisions made. Second, it makes a methodological contribution by presenting a heuristics-based rationale extraction system called Rationale Miner that employs multiple heuristics, and follows a data-driven, bottom-up approach to infer the rationale behind specific decisions (e.g., whether a new module is implemented based on core developer consensus or benevolent dictator's pronouncement). Our approach can be applied to extract rationale in other OSSD communities that have similar governance structures.

[**Sharp2016**] Helen Sharp, Yvonne Dittrich, and Cleidson R. B. de Souza. The role of ethnographic studies in empirical software engineering. *IEEE Transactions on Software Engineering*, 42(8):786–804, 8 2016, DOI 10.1109/tse.2016.2519887.
Abstract: Ethnography is a qualitative research method used to study people and cultures. It is largely adopted in disciplines outside software engineering, including different areas of computer science. Ethnography can provide an in-depth understanding of the socio-technological realities surrounding everyday software development practice, i.e., it can help to uncover not only what practitioners do, but also why they do it. Despite its potential, ethnography has not been widely adopted by empirical software engineering researchers, and receives little attention in the related literature. The main goal of this paper is to explain how empirical software engineering researchers would benefit from adopting ethnography. This is achieved by explicating four roles that ethnography can play in furthering the goals of empirical software engineering: to strengthen investigations into the social and human aspects of software engineering; to inform the design of software engineering tools; to improve method and process development; and to inform research programmes. This article introduces ethnography, explains its origin, context, strengths and weaknesses, and presents a set of dimensions that position ethnography as a useful and usable approach to empirical software engineering research. Throughout the paper, relevant examples of ethnographic studies of software practice are used to illustrate the points being made.

[**Sholler2019**] Dan Sholler, Igor Steinmacher, Denae Ford, Mara Averick, Mike Hoye, and Greg Wilson. Ten simple rules for helping newcomers become contributors to open projects. *PLOS Computational Biology*, 15(9):e1007296, 9 2019, DOI 10.1371/journal.pcbi.1007296.
Abstract: To survive and thrive, a community must attract new members, retain them, and help them be productive. As openness becomes the norm in research, software development, and education, knowing how to do this has become a essential skill for principal investigators and community managers alike. A growing body of knowledge in sociology, anthropology, education, and software engineering can guide decisions about how to facilitate this.

[**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.

[**Simon2021**] Simon and Juha Sorva. How concrete should an abstract be? In *Proc. Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, 6 2021, DOI `10.1145/3430665.3456342`.

Abstract: For many decades the abstract has served as a standalone summary of an academic publication, one that succinctly informs readers of what they might expect to find upon reading the paper. While some publication venues require abstracts to conform with a specified structure, many others, including ITiCSE, leave the structure entirely to the paper's authors. In this paper we report on the components identified in the abstracts of ITiCSE's full papers and working group reports. We examine the abstracts of all 1496 of these publications from 25 years of ITiCSE to determine what structural elements they employ. We also construct something of an ethos of computing education by compiling assertions from the introductions of many abstracts. We find, among other things, that very few abstracts include all of the components that are recommended in a structured abstract; that a number of abstracts consist of nothing but background; that nearly half of abstracts do not include any results; and that nearly five percent of abstracts include references, despite often not having an associated reference list. As an example from the ethos, we find that industry wants people with soft skills, and it is important that we teach our students these skills. Our analysis will guide future ITiCSE authors as they consider how to formulate their own abstracts.

[**Sliwerski2005**] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proc. International Conference on Mining Software Repositories (MSR)*. ACM Press, 2005, DOI `10.1145/1083142.1083147`.

Abstract: As a software system evolves, programmers make changes that sometimes cause problems. We analyze CVS archives for fix-inducing changes—changes that lead to problems, indicated by fixes. We show how to automatically locate fix-inducing changes by linking a version archive (such as CVS) to a bug database (such as Bugzilla). In a first investigation of the Mozilla and Eclipse history, it turns out that fix-inducing changes show distinct patterns with respect to their size and the day of week they were applied.

[**Sobrinho2021**] Elder Vicente de Paulo Sobrinho, Andrea De Lucia, and Marcelo de Almeida Maia. A systematic literature review on bad smells–5 w's: Which, when, what, who, where. *IEEE Transactions on Software Engineering*, 47(1):17–66, 1 2021, DOI `10.1109/tse.2018.2880977`.

Abstract: Bad smells are sub-optimal code structures that may represent problems needing attention. We conduct an extensive literature review on bad smells relying on a large body of knowledge from 1990 to 2017. We show that some smells are much more studied in the literature than others, and also that some of them are intrinsically inter-related (which). We give a perspective on how the research has been driven across time (when). In particular, while the interest in duplicated code emerged before the reference publications by Fowler and Beck and by Brown et al., other types of bad smells only started to be studied after these seminal publications, with an increasing trend in the last decade. We analyzed aims, findings, and respective experimental settings, and observed that the variability of these elements may be responsible for some apparently contradictory findings on bad smells (what). Moreover, we could observe that, in general, papers tend to study different types of smells at once. However, only a small percentage of those papers actually investigate possible relations between the respective smells (co-studies), i.e., each smell tends to be studied in isolation. Despite of a few relations between some types of bad smells have been investigated, there are other possible relations for further investigation. We also report that authors have different levels of interest in the subject, some of them publishing sporadically and others continuously (who). We observed that scientific connections are ruled by a large "small world" connected graph among researchers and several small disconnected graphs. We also found that the communities studying duplicated code and other types of bad smells are largely separated. Finally, we observed that some venues are more likely to disseminate knowledge on Duplicate Code (which often is listed as a conference topic on its own), while others have a more balanced distribution among other smells (where). Finally, we provide a discussion on future directions for bad smell research.

[**Soremekun2021**] Ezekiel Soremekun, Lukas Kirschner, Marcel Böhme, and Andreas Zeller. Locating faults with program slicing: an empirical analysis. *Empirical Software Engineering*, 26(3), 4 2021, DOI `10.1007/s10664-020-09931-7`.

Abstract: Statistical fault localization is an easily deployed technique for

quickly determining candidates for faulty code locations. If a human programmer has to search the fault beyond the top candidate locations, though, more traditional techniques of following dependencies along dynamic slices may be better suited. In a large study of 457 bugs (369 single faults and 88 multiple faults) in 46 open source C programs, we compare the effectiveness of statistical fault localization against dynamic slicing. For single faults, we find that dynamic slicing was eight percentage points more effective than the best performing statistical debugging formula; for 66% of the bugs, dynamic slicing finds the fault earlier than the best performing statistical debugging formula. In our evaluation, dynamic slicing is more effective for programs with single fault, but statistical debugging performs better on multiple faults. Best results, however, are obtained by a hybrid approach : If programmers first examine at most the top five most suspicious locations from statistical debugging, and then switch to dynamic slices, on average, they will need to examine 15% (30 lines) of the code. These findings hold for 18 most effective statistical debugging formulas and our results are independent of the number of faults (i.e. single or multiple faults) and error type (i.e. artificial or real errors).

[**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.

[**Spadini2019**] Davide Spadini, Fabio Palomba, Tobias Baum, Stefan Hanenberg, Magiel Bruntink, and Alberto Bacchelli. Test-driven code review: An empirical study. In *Proc. International Conference on Software Engineering*

144

(ICSE). IEEE, 5 2019, DOI `10.1109/icse.2019.00110`.

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

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

Abstract: In contemporary code review, the comments put by reviewers on a specific code change are immediately visible to the other reviewers involved. Could this visibility prime new reviewers' attention (due to the human's proneness to availability bias), thus biasing the code review outcome? In this study, we investigate this topic by conducting a controlled experiment with 85 developers who perform a code review and a psychological experiment. With the psychological experiment, we find that $\approx$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

[**Spasic2020**] Mirko Spasić and Milena Vujošević Janičić. Verification supported refactoring of embedded SQL. *Software Quality Journal*, 29(3):629–665, 6 2020, DOI `10.1007/s11219-020-09517-y`.

Abstract: Improving code quality without changing its functionality, e.g., by refactoring or optimization, is an everyday programming activity. Good programming practice requires that each such change should be followed by a

check if the change really preserves the code behavior. If such a check is performed by testing, it can be time consuming and still cannot guarantee the absence of differences in behavior between two versions of the code. Hence, tools that could automatically verify code equivalence would be of great help. An area that we are focused on is embedded sql programming. There are a number of approaches for dealing with equivalence of either pairs of imperative code fragments or pairs of sql statements. However, in database-driven applications, simultaneous changes (changes that include both sql and a host language code) are also present and important. Such changes can preserve the overall equivalence without preserving equivalence of these two parts considered separately. In this paper, we propose an automated approach for dealing with equivalence of programs after such changes, a problem that is hardly tackled in literature. Our approach uses our custom first-order logic modeling of sql queries that corresponds to imperative semantics. The approach generates equivalence conditions that can be efficiently checked using smt solvers or first-order logic provers. We implemented the proposed approach as a framework sqlav, which is publicly available and open source.

[**Spiegler2021**] Simone V. Spiegler, Christoph Heinecke, and Stefan Wagner. An empirical study on changing leadership in agile teams. *Empirical Software Engineering*, 26(3), 3 2021, DOI `10.1007/s10664-021-09949-5`.

Abstract: An increasing number of companies aim to enable their development teams to work in an agile manner. When introducing agile teams, companies face several challenges. This paper explores the kind of leadership needed to support teams to work in an agile way. One theoretical agile leadership concept describes a Scrum Master who is supposed to empower the team to lead itself. Empirical findings on such a leadership role are controversial. We still have not understood how leadership unfolds in a team that is by definition self-organizing. Further exploration is needed to better understand leadership in agile teams. Our goal is to explore how leadership changes while the team matures using the example of the Scrum Master. Through a grounded theory study containing 75 practitioners from 11 divisions at the Robert Bosch GmbH we identified a set of nine leadership roles that are transferred from the Scrum Master to the Development Team while it matures. We uncovered that a leadership gap and a supportive internal team climate are enablers of the role transfer process, whereas role conflicts may diminish the role transfer. To make the Scrum Master change in a mature team, team members need to receive trust and freedom to take on a leadership role which was previously filled by the Scrum Master. We conclude with practical implications for managers, Product Owners, Development Teams and Scrum Masters which they can apply in real settings.

[**Spinellis2016**] Diomidis Spinellis, Panos Louridas, and Maria Kechagia. The evolution of c programming practices. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 5 2016, DOI `10.1145/2884781.2884799`.

Abstract: Tracking long-term progress in engineering and applied science

allows us to take stock of things we have achieved, appreciate the factors that led to them, and set realistic goals for where we want to go. We formulate seven hypotheses associated with the long term evolution of C programming in the Unix operating system, and examine them by extracting, aggregating, and synthesising metrics from 66 snapshots obtained from a synthetic software configuration management repository covering a period of four decades. We found that over the years developers of the Unix operating system appear to have evolved their coding style in tandem with advancements in hardware technology, promoted modularity to tame rising complexity, adopted valuable new language features, allowed compilers to allocate registers on their behalf, and reached broad agreement regarding code formatting. The progress we have observed appears to be slowing or even reversing prompting the need for new sources of innovation to be discovered and followed.

[**Spinellis2021**] Diomidis Spinellis and Paris Avgeriou. Evolution of the unix system architecture: An exploratory case study. *IEEE Transactions on Software Engineering*, 47(6):1134–1163, 6 2021, DOI 10.1109/tse.2019.2892149.
Abstract: Unix has evolved for almost five decades, shaping modern operating systems, key software technologies, and development practices. Studying the evolution of this remarkable system from an architectural perspective can provide insights on how to manage the growth of large, complex, and long-lived software systems. Along main Unix releases leading to the FreeBSD lineage we examine core architectural design decisions, the number of features, and code complexity, based on the analysis of source code, reference documentation, and related publications. We report that the growth in size has been uniform, with some notable outliers, while cyclomatic complexity has been religiously safeguarded. A large number of Unix-defining design decisions were implemented right from the very early beginning, with most of them still playing a major role. Unix continues to evolve from an architectural perspective, but the rate of architectural innovation has slowed down over the system's lifetime. Architectural technical debt has accrued in the forms of functionality duplication and unused facilities, but in terms of cyclomatic complexity it is systematically being paid back through what appears to be a self-correcting process. Some unsung architectural forces that shaped Unix are the emphasis on conventions over rigid enforcement, the drive for portability, a sophisticated ecosystem of other operating systems and development organizations, and the emergence of a federated architecture, often through the adoption of third-party subsystems. These findings have led us to form an initial theory on the architecture evolution of large, complex operating system software.

[**Staples2013**] Mark Staples, Rafal Kolanski, Gerwin Klein, Corey Lewis, June Andronick, Toby Murray, Ross Jeffery, and Len Bass. Formal specifications better than function points for code sizing. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2013, DOI

10.1109/icse.2013.6606692.

Abstract: Size and effort estimation is a significant challenge for the management of large-scale formal verification projects. We report on an initial study of relationships between the sizes of artefacts from the development of seL4, a formally-verified embedded systems microkernel. For each API function we first determined its COSMIC Function Point (CFP) count (based on the seL4 user manual), then sliced the formal specifications and source code, and performed a normalised line count on these artefact slices. We found strong and significant relationships between the sizes of the artefact slices, but no significant relationships between them and the CFP counts. Our finding that CFP is poorly correlated with lines of code is based on just one system, but is largely consistent with prior literature. We find CFP is also poorly correlated with the size of formal specifications. Nonetheless, lines of formal specification correlate with lines of source code, and this may provide a basis for size prediction in future formal verification projects. In future work we will investigate proof sizing.

[**Stefik2011**] Andreas Stefik, Susanna Siebert, Melissa Stefik, and Kim Slattery. An empirical comparison of the accuracy rates of novices using the Quorum, Perl, and Randomo programming languages. In *Proc. Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*. ACM Press, 2011, DOI 10.1145/2089155.2089159.

Abstract: We present here an empirical study comparing the accuracy rates of novices writing software in three programming languages: Quorum, Perl, and Randomo. The first language, Quorum, we call an evidence-based programming language, where the syntax, semantics, and API designs change in correspondence to the latest academic research and literature on programming language usability. Second, while Perl is well known, we call Randomo a Placebo-language, where some of the syntax was chosen with a random number generator and the ASCII table. We compared novices that were programming for the first time using each of these languages, testing how accurately they could write simple programs using common program constructs (e.g., loops, conditionals, functions, variables, parameters). Results showed that while Quorum users were afforded significantly greater accuracy compared to those using Perl and Randomo, Perl users were unable to write programs more accurately than those using a language designed by chance.

[**Stefik2013**] Andreas Stefik and Susanna Siebert. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education*, 13(4):1–40, 11 2013, DOI 10.1145/2534973.

Abstract: Recent studies in the literature have shown that syntax remains a significant barrier to novice computer science students in the field. While this syntax barrier is known to exist, whether and how it varies across programming languages has not been carefully investigated. For this article, we conducted four empirical studies on programming language syntax as part of a larger analysis into the, so called, programming language wars. We first

present two surveys conducted with students on the intuitiveness of syntax, which we used to garner formative clues on what words and symbols might be easy for novices to understand. We followed up with two studies on the accuracy rates of novices using a total of six programming languages: Ruby, Java, Perl, Python, Randomo, and Quorum. Randomo was designed by randomly choosing some keywords from the ASCII table (a metaphorical placebo). To our surprise, we found that languages using a more traditional C-style syntax (both Perl and Java) did not afford accuracy rates significantly higher than a language with randomly generated keywords, but that languages which deviate (Quorum, Python, and Ruby) did. These results, including the specifics of syntax that are particularly problematic for novices, may help teachers of introductory programming courses in choosing appropriate first languages and in helping students to overcome the challenges they face with syntax.

[**Stol2018**] Klaas-Jan Stol and Brian Fitzgerald. The ABC of software engineering research. *ACM Transactions on Software Engineering and Methodology*, 27(3):1–51, 10 2018, DOI `10.1145/3241743`.
Abstract: A variety of research methods and techniques are available to SE researchers, and while several overviews exist, there is consistency neither in the research methods covered nor in the terminology used. Furthermore, research is sometimes critically reviewed for characteristics inherent to the methods. We adopt a taxonomy from the social sciences, termed here the ABC framework for SE research, which offers a holistic view of eight archetypal research strategies. ABC refers to the research goal that strives for generalizability over Actors (A) and precise measurement of their Behavior (B), in a realistic Context (C). The ABC framework uses two dimensions widely considered to be key in research design: the level of obtrusiveness of the research and the generalizability of research findings. We discuss metaphors for each strategy and their inherent limitations and potential strengths. We illustrate these research strategies in two key SE domains, global software engineering and requirements engineering, and apply the framework on a sample of 75 articles. Finally, we discuss six ways in which the framework can advance SE research.

[**Stolee2011**] Kathryn T. Stolee and Sebastian Elbaum. Refactoring pipe-like mashups for end-user programmers. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 5 2011, DOI `10.1145/1985793.1985805`.
Abstract: Mashups are becoming increasingly popular as end users are able to easily access, manipulate, and compose data from many web sources. We have observed, however, that mashups tend to suffer from deficiencies that propagate as mashups are reused. To address these deficiencies, we would like to bring some of the benefits of software engineering techniques to the end users creating these programs. In this work, we focus on identifying code smells indicative of the deficiencies we observed in web mashups programmed in the popular Yahoo! Pipes environment. Through an empirical study, we

explore the impact of those smells on end-user programmers and observe that users generally prefer mashups without smells. We then introduce refactorings targeting those smells, reducing the complexity of the mashup programs, increasing their abstraction, updating broken data sources and dated components, and standardizing their structures to fit the community development patterns. Our assessment of a large sample of mashups shows that smells are present in 81% of them and that the proposed refactorings can reduce the number of smelly mashups to 16%, illustrating the potential of refactoring to support the thousands of end users programming mashups.

[**Stray2021**] Viktoria Stray, Raluca Florea, and Lucas Paruch. Exploring human factors of the agile software tester. *Software Quality Journal*, 6 2021, DOI `10.1007/s11219-021-09561-2`.
Abstract: Although extensive research has been conducted on the characteristics of the agile developer, little attention has been given to the features of the software-testing role. This paper explores the human factors of the software testers working in agile projects through a qualitative study focusing on how these factors are perceived. We interviewed 22 agile software practitioners working in three international companies: 14 testers, five developers, and three designers. Additionally, we observed 11 meetings and daily work of 13 participants in one of the companies. Our findings show that the views on the human factors shaping the agile software tester's role were crystallized into seven traits, which the agile team members saw as central for the software-testing role: the ability to see the whole picture, good communication skills, detail-orientation, structuredness, creativeness, curiosity, and adaptability. The testers spent half their day communicating and learned how to mitigate the fact that they had to bring bad news to other project members. They also facilitated communication between the business side and development. Based on our results, we propose the seven traits as dimensions to consider for organizations recruiting agile software testers, as well as a reference for IT and non-IT professionals considering a software-testing career.

[**Stylos2007**] Jeffrey Stylos and Steven Clarke. Usability implications of requiring parameters in objects' constructors. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2007, DOI `10.1109/icse.2007.92`.
Abstract: The usability of APIs is increasingly important to programmer productivity. Based on experience with usability studies of specific APIs, techniques were explored for studying the usability of design choices common to many APIs. A comparative study was performed to assess how professional programmers use APIs with required parameters in objects' constructors as opposed to parameterless "default" constructors. It was hypothesized that required parameters would create more usable and self- documenting APIs by guiding programmers toward the correct use of objects and preventing errors. However, in the study, it was found that, contrary to expectations, programmers strongly preferred and were more effective with APIs that did not require constructor parameters. Participants' behavior was analyzed using

the cognitive dimensions framework, and revealing that required constructor parameters interfere with common learning strategies, causing undesirable premature commitment.

[**Taipalus2018**] Toni Taipalus, Mikko Siponen, and Tero Vartiainen. Errors and complications in SQL query formulation. *ACM Transactions on Computing Education*, 18(3):1–29, 9 2018, DOI 10.1145/3231712.
Abstract: SQL is taught in almost all university level database courses, yet SQL has received relatively little attention in educational research. In this study, we present a database management system independent categorization of SQL query errors that students make in an introductory database course. We base the categorization on previous literature, present a class of logical errors that has not been studied in detail, and review and complement these findings by analyzing over 33,000 SQL queries submitted by students. Our analysis verifies error findings presented in previous literature and reveals new types of errors, namely logical errors recurring in similar manners among different students. We present a listing of fundamental SQL query concepts we have identified and based our exercises on, a categorization of different errors and complications, and an operational model for designing SQL exercises.

[**Taipalus2021**] Toni Taipalus, Hilkka Grahn, and Hadi Ghanbari. Error messages in relational database management systems: A comparison of effectiveness, usefulness, and user confidence. *Journal of Systems and Software*, 181:111034, 11 2021, DOI 10.1016/j.jss.2021.111034.
Abstract: The database and the database management system (DBMS) are two of the main components of any information system. Structured Query Language (SQL) is the most popular query language for retrieving data from the database, as well as for many other data management tasks. During system development and maintenance, software developers use a considerable amount of time to interpret compiler error messages. The quality of these error messages has been demonstrated to affect software development effectiveness, and correctly formulating queries and fixing them when needed is an important task for many software developers. In this study, we set out to investigate how participants (N = 152) experienced the qualities of error messages of four popular DBMSs in terms of error message effectiveness, perceived usefulness for finding and fixing errors, and error recovery confidence. Our results show differences between the DBMSs by three of the four metrics, and indicate a discrepancy between objective effectiveness and subjective usefulness. The results suggest that although error messages have perceived differences in terms of usefulness for finding and fixing errors, these differences may not necessarily result in differences in query fixing success rates.

[**Tamburri2020**] Damian Andrew Tamburri, Kelly Blincoe, Fabio Palomba, and Rick Kazman. "the canary in the coal mine. . . " a cautionary tale from the decline of SourceForge. *Software: Practice and Experience*, 50(10):1930–

1951, 7 2020, DOI `10.1002/spe.2874`.

Abstract: Forges are online collaborative platforms to support the development of distributed open source software. While once mighty keepers of open source vitality, software forges are rapidly becoming less and less relevant. For example, of the top 10 forges in 2011, only one survives today—SourceForge—the biggest of them all, but its numbers are dropping and its community is tenuous at best. Through mixed-methods research, this article chronicles and analyze the software practice and experiences of the project's history—in particular its architectural and community/organizational decisions. We discovered a number of suboptimal social and architectural decisions and circumstances that, may have led to SourceForge's demise. In addition, we found evidence suggesting that the impact of such decisions could have been monitored, reduced, and possibly avoided altogether. The use of sociotechnical insights needs to become a basic set of design and software/organization monitoring principles that tell a cautionary tale on what to measure and what not to do in the context of large-scale software forge and community design and management.

[**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.

[**Tan2020b**] Jie Tan, Daniel Feitosa, Paris Avgeriou, and Mircea Lungu. Evolution of technical debt remediation in python: A case study on the apache software ecosystem. *Journal of Software: Evolution and Process*, 33(4), 11 2020, DOI `10.1002/smr.2319`.
Abstract: In recent years, the evolution of software ecosystems and the detection of technical debt received significant attention by researchers from both industry and academia. While a few studies that analyze various aspects of technical debt evolution already exist, to the best of our knowledge, there is no large-scale study that focuses on the remediation of technical debt over time in Python projects—that is, one of the most popular programming languages at the moment. In this paper, we analyze the evolution of technical debt in 44 Python open-source software projects belonging to the Apache Software Foundation. We focus on the type and amount of technical debt that is paid back. The study required the mining of over 60K commits, detailed code analysis on 3.7K system versions, and the analysis of almost 43K fixed issues. The findings show that most of the repayment effort goes into testing, documentation, complexity, and duplication removal. Moreover, more than half of the Python technical debt is short term being repaid in less than 2 months. In particular, the observations that a minority of rules account for the majority of issues fixed and spent effort suggest that addressing those kinds of debt in the future is important for research and practice.

[**Tao2021**] Yida Tao, Zhihui Chen, Yepang Liu, Jifeng Xuan, Zhiwu Xu, and Shengchao Qin. Demystifying "bad" error messages in data science libraries. In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 8 2021, DOI `10.1145/3468264.3468560`.
Abstract: Error messages are critical starting points for debugging. Unfortunately, they seem to be notoriously cryptic, confusing, and uninformative. Yet, it still remains a mystery why error messages receive such bad reputations, especially given that they are merely very short pieces of natural language text. In this paper, we empirically demystify the causes and fixes of "bad" error messages, by qualitatively studying 201 Stack Overflow threads and 335 GitHub issues. We specifically focus on error messages encountered in data science development, which is an increasingly important but not well studied domain. We found that the causes of "bad" error messages are far more complicated than poor phrasing or flawed articulation of error message content. Many error messages are inherently and inevitably misleading or uninformative, since libraries do not know user intentions and cannot "see" external errors. Fixes to error-message-related issues mostly involve source code changes, while exclusive message content updates only take up a small portion. In addition, whether an error message is informative or helpful is not always clear-cut; even error messages that clearly pinpoint faults and resolutions can still cause confusion for certain users. These findings thus call for a more in-depth investigation on how error messages should be

evaluated and improved in the future.

[**Tew2011**] Allison Elliott Tew and Mark Guzdial. The FCS1: a language independent assessment of CS1 knowledge. In *Proc. Technical Symposium on Computer Science Education (SIGCSE)*. ACM Press, 2011, DOI `10.1145/1953163.1953200`.

Abstract: A primary goal of many CS education projects is to determine the extent to which a given intervention has had an impact on student learning. However, computing lacks valid assessments for pedagogical or research purposes. Without such valid assessments, it is difficult to accurately measure student learning or establish a relationship between the instructional setting and learning outcomes. We developed the Foundational CS1 (FCS1) Assessment instrument, the first assessment instrument for introductory computer science concepts that is applicable across a variety of current pedagogies and programming languages. We applied methods from educational and psychological test development, adapting them as necessary to fit the disciplinary context. We conducted a large scale empirical study to demonstrate that pseudo-code was an appropriate mechanism for achieving programming language independence. Finally, we established the validity of the assessment using a multi-faceted argument, combining interview data, statistical analysis of results on the assessment, and CS1 exam scores.

[**Thongtanunam2016**] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 5 2016, DOI `10.1145/2884781.2884852`.

Abstract: Code ownership establishes a chain of responsibility for modules in large software systems. Although prior work uncovers a link between code ownership heuristics and software quality, these heuristics rely solely on the authorship of code changes. In addition to authoring code changes, developers also make important contributions to a module by reviewing code changes. Indeed, recent work shows that reviewers are highly active in modern code review processes, often suggesting alternative solutions or providing updates to the code changes. In this paper, we complement traditional code ownership heuristics using code review activity. Through a case study of six releases of the large Qt and OpenStack systems, we find that: (1) 67%-86% of developers did not author any code changes for a module, but still actively contributed by reviewing 21%-39% of the code changes, (2) code ownership heuristics that are aware of reviewing activity share a relationship with software quality, and (3) the proportion of reviewers without expertise shares a strong, increasing relationship with the likelihood of having post-release defects. Our results suggest that reviewing activity captures an important aspect of code ownership, and should be included in approximations of it in future studies.

[**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.

[**Tomassi2019**] David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-Gonzalez. BugSwarm: Mining and continuously growing a dataset of reproducible failures and fixes. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2019, DOI `10.1109/icse.2019.00048`.

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

[**Tourani2017**] Parastou Tourani, Bram Adams, and Alexander Serebrenik. Code of conduct in open source projects. In *Proc. International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2 2017, DOI `10.1109/saner.2017.7884606`.

Abstract: Open source projects rely on collaboration of members from all around the world using web technologies like GitHub and Gerrit. This mixture of people with a wide range of backgrounds including minorities like women, ethnic minorities, and people with disabilities may increase the risk of offensive and destroying behaviours in the community, potentially leading affected project members to leave towards a more welcoming and friendly environment. To counter these effects, open source projects increasingly are turning to codes of conduct, in an attempt to promote their expectations and standards of ethical behaviour. In this first of its kind empirical study of codes of conduct in open source software projects, we investigated the role, scope and influence of codes of conduct through a mixture of quantitative and qualitative analysis, supported by interviews with practitioners. We found that the top codes of conduct are adopted by hundreds to thousands of projects, while all of them share 5 common dimensions.

[**Trang2021**] Simon Trang and Welf H. Weiger. The perils of gamification: Does engaging with gamified services increase users' willingness to disclose personal information? *Computers in Human Behavior*, 116:106644, 3 2021, DOI `10.1016/j.chb.2020.106644`.

Abstract: The increasing use of gamification in the digital service landscape has caught the attention of practitioners and marketers alike. Alarmingly, most of the empirical research has attested to the benefits of such gamified service (e.g. apps) use while neglecting to address potential drawbacks. This research suggests that users of gamified apps end up being more likely to share private information with firms, thus threatening their own personal information privacy. Against this background, the present study links gamification to information disclosure and demonstrates that if a gamified service conveys experiences of, for instance, social comparison, it can indeed lead to greater willingness to disclose personal information. This relationship can be explained by the users' increased resource depletion through cognitive absorption (i.e. the concentration of one's entire affective, cognitive, and physical resources on the task at hand). The results further indicate that engaging with gamified apps indeed affects the situational processing of privacy-related decisions (i.e. calculating benefits vs. risks) and the role of dispositional antecedents: In states of deep cognitive absorption, users disclose even more information when they perceive privacy benefits (i.e., situational) and even less when they have high privacy concerns (i.e., dispositional).

[**Tregubov2017**] Alexey Tregubov, Barry Boehm, Natalia Rodchenko, and Jo Ann Lane. Impact of task switching and work interruptions on software development processes. In *Proc. International Conference on Software and System Process (ICSSP)*. ACM, 7 2017, DOI `10.1145/3084100.3084116`.

Abstract: Software developers often work on multiple projects and tasks throughout a work day, which may affect their productivity and quality of work. Knowing how working on several projects at a time affects productivity can improve cost and schedule estimations. It also can provide additional insights for better work scheduling and the development process. We want to achieve a better productivity without losing the benefits of work interruptions and multitasking for developers involved in the process. To understand how the development process can be improved, first, we identify work interruptions that mostly have a negative effect on productivity, second, we need to quantitatively evaluate impact of multitasking (task switching, work context switching) and work interruptions on productivity. In this research we study cross-project multitasking among the developers working on multiple projects in an educational setting. We propose a way to evaluate the number of cross-project interruptions among software developers using self-reported work logs. This paper describes the research that found: a) software developers involved in two or more projects on average spend 17% of their development effort on cross-project interruptions, b) the amount of effort spent on interruptions is overestimated by the G. Weinberg's heuristic, c) the correlation between the number of projects and effort spent by developers on cross-project interruptions is relatively weak, and d) there is strong correlation between the number of projects and the number of interruptions developers reported.

[**Turcotte2020**] Alexi Turcotte, Aviral Goel, Filip Křikava, and Jan Vitek. Designing types for R, empirically. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–25, 11 2020, DOI `10.1145/3428249`.
Abstract: The R programming language is widely used in a variety of domains. It was designed to favor an interactive style of programming with minimal syntactic and conceptual overhead. This design is well suited to data analysis, but a bad fit for tools such as compilers or program analyzers. In particular, R has no type annotations, and all operations are dynamically checked at run-time. The starting point for our work are the two questions: what expressive power is needed to accurately type R code? and which type system is the R community willing to adopt? Both questions are difficult to answer without actually experimenting with a type system. The goal of this paper is to provide data that can feed into that design process. To this end, we perform a large corpus analysis to gain insights in the degree of polymorphism exhibited by idiomatic R code and explore potential benefits that the R community could accrue from a simple type system. As a starting point, we infer type signatures for 25,215 functions from 412 packages among the most widely used open source R libraries. We then conduct an evaluation on 8,694 clients of these packages, as well as on end-user code from the Kaggle data science competition website.

[**Vanhanen2007**] Jari Vanhanen and Harri Korpi. Experiences of using pair programming in an agile project. In *Proc. Hawaii International Conference on System Sciences (HICSS)*. IEEE, 2007, DOI `10.1109/hicss.2007.218`.

Abstract: The interest in pair programming (PP) has increased recently, e.g. by the popularization of agile software development. However, many practicalities of PP are poorly understood. We present experiences of using PP extensively in an industrial project. The fact that the team had a limited number of high-end workstations forced it in a positive way to quick deployment and rigorous use of PP. The developers liked PP and learned it easily. Initially, the pairs were not rotated frequently but adopting daily, random rotation improved the situation. Frequent rotation seemed to improve knowledge transfer. The driver/navigator roles were switched seldom, but still the partners communicated actively. The navigator rarely spotted defects during coding, but the released code contained almost no defects. Test-driven development and design in pairs possibly decreased defects. The developers considered that PP improved quality and knowledge transfer, and was better suited for complex tasks than for easy tasks

[**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.

[**Wang2016**] Xinyu Wang, Sumit Gulwani, and Rishabh Singh. FIDEX: filtering spreadsheet data using examples. In *Proc. International Conference on Object-Oriented Programming Systems Languages and Applications (OOP-SLA)*. ACM, 10 2016, DOI `10.1145/2983990.2984030`.
Abstract: Data filtering in spreadsheets is a common problem faced by

millions of end-users. The task of data filtering requires a computational model that can separate intended positive and negative string instances. We present a system, FIDEX, that can efficiently learn desired data filtering expressions from a small set of positive and negative string examples. There are two key ideas of our approach. First, we design an expressive DSL to represent disjunctive filter expressions needed for several real-world data filtering tasks. Second, we develop an efficient synthesis algorithm for incrementally learning consistent filter expressions in the DSL from very few positive and negative examples. A DAG-based data structure is used to succinctly represent a large number of filter expressions, and two corresponding operators are defined for algorithmically handling positive and negative examples, namely, the intersection and subtraction operators. FIDEX is able to learn data filters for 452 out of 460 real-world data filtering tasks in real time (0.22s), using only 2.2 positive string instances and 2.7 negative string instances on average.

[**Wang2020**] Peipei Wang, Chris Brown, Jamie A. Jennings, and Kathryn T. Stolee. An empirical study on regular expression bugs. In *Proc. International Conference on Mining Software Repositories (MSR)*. ACM, 6 2020, DOI 10.1145/3379597.3387464.
Abstract: Understanding the nature of regular expression (regex) issues is important to tackle practical issues developers face in regular expression usage. Knowledge about the nature and frequency of various types of regular expression issues, such as those related to performance, API misuse, and code smells, can guide testing, inform documentation writers, and motivate refactoring efforts. However, beyond ReDoS (Regular expression Denial of Service), little is known about to what extent regular expression issues affect software development and how these issues are addressed in practice. This paper presents a comprehensive empirical study of 350 merged regex-related pull requests from Apache, Mozilla, Facebook, and Google GitHub repositories. Through classifying the root causes and manifestations of those bugs, we show that incorrect regular expression behavior is the dominant root cause of regular expression bugs (165/356, 46.3%). The remaining root causes are incorrect API usage (9.3%) and other code issues that require regular expression changes in the fix (29.5%). By studying the code changes of regex-related pull requests, we observe that fixing regular expression bugs is nontrivial as it takes more time and more lines of code to fix them compared to the general pull requests. The results of this study contribute to a broader understanding of the practical problems faced by developers when using regular expressions.

[**Washburn2016**] Michael Washburn, Pavithra Sathiyanarayanan, Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. What went right and what went wrong: an analysis of 155 postmortems from game development. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 5 2016, DOI 10.1145/2889160.2889253.

Abstract: In game development, software teams often conduct postmortems to reflect on what went well and what went wrong in a project. The postmortems are shared publicly on gaming sites or at developer conferences. In this paper, we present an analysis of 155 postmortems published on the gaming site Gamasutra.com. We identify characteristics of game development, link the characteristics to positive and negative experiences in the postmortems and distill a set of best practices and pitfalls for game development.

[**WeillTessier2021**] Pierre Weill-Tessier, Alexandra Lucia Costache, and Neil C. C. Brown. Usage of the Java language by novices over time: implications for tool and language design. In *Proc. Technical Symposium on Computer Science Education (SIGCSE)*. ACM, 3 2021, DOI 10.1145/3408877.3432408.
Abstract: Java is a popular programming language for teaching at university level. BlueJ is a popular tool for teaching Java to beginners. We provide several analyses of Java use in BlueJ to answer three questions: what use is made of different parts of Java by beginners when learning to program; how has this pattern of use changed between 2013 and 2019 in a longstanding language such as Java; and to what extent do beginners follow the specific style that BlueJ is designed to guide them into? These analyses allow us to see what features are important in object-oriented introductory programming languages, which could inform language and tool designers—and see to what extent the design of these programming tools can have an effect on the way the language is used. We find that many beginners disobey the guidelines that BlueJ promotes, and that patterns of Java use are generally stable over time—but we do see decreased exception use and a change in target application domains away from GUI programming towards text processing. We conclude that programming languages for novices could have fewer built-in types but should retain rich libraries.

[**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

160

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.

[**Weir2021**] Charles Weir, Ingolf Becker, and Lynne Blair. A passion for security: Intervening to help software developers. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2021, DOI `10.1109/icse-seip52600.2021.00011`.
Abstract: While the techniques to achieve secure, privacy-preserving software are now well understood, evidence shows that many software development teams do not use them: they lack the 'security maturity' to assess security needs and decide on appropriate tools and processes; and they lack the ability to negotiate with product management for the required resources. This paper describes a measuring approach to assess twelve aspects of this security maturity; its use to assess the impact of a lightweight package of workshops designed to increase security maturity; and a novel approach within that package to support developers in resource negotiation. Based on trials in eight organizations, involving over 80 developers, this paper demonstrates that (1) development teams can notably improve their security maturity even in the absence of security specialists; and (2) suitably guided, developers can find effective ways to promote security to product management. Empowering developers to make their own decisions and promote security in this way offers a powerful grassroots approach to improving the security of software worldwide.

[**Wessel2020**] Mairieli Wessel, Alexander Serebrenik, Igor Wiese, Igor Steinmacher, and Marco A. Gerosa. Effects of adopting code review bots on pull requests to OSS projects. In *Proc. International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 9 2020, DOI `10.1109/icsme46990.2020.00011`.
Abstract: Software bots, which are widely adopted by Open Source Software (OSS) projects, support developers on several activities, including code review. However, as with any new technology adoption, bots may impact group dynamics. Since understanding and anticipating such effects is important for planning and management, we investigate how several activity indicators change after the adoption of a code review bot. We employed a regression discontinuity design on 1,194 software projects from GitHub. Our results indicate that the adoption of code review bots increases the number of monthly merged pull requests, decreases monthly non-merged pull requests, and decreases communication among developers. Practitioners and maintainers may leverage our results to understand, or even predict, bot effects on their projects' social interactions.

[**Wicherts2011**] Jelte M. Wicherts, Marjan Bakker, and Dylan Molenaar. Willingness to share research data is related to the strength of the evidence and the quality of reporting of statistical results. *PLoS ONE*, 6(11):e26828, 11 2011, DOI `10.1371/journal.pone.0026828`.
Abstract: Background The widespread reluctance to share published research data is often hypothesized to be due to the authors' fear that reanalysis may expose errors in their work or may produce conclusions that contradict their own. However, these hypotheses have not previously been studied systematically. Methods and Findings We related the reluctance to share research data for reanalysis to 1148 statistically significant results reported in 49 papers published in two major psychology journals. We found the reluctance to share data to be associated with weaker evidence (against the null hypothesis of no effect) and a higher prevalence of apparent errors in the reporting of statistical results. The unwillingness to share data was particularly clear when reporting errors had a bearing on statistical significance. Conclusions Our findings on the basis of psychological papers suggest that statistical results are particularly hard to verify when reanalysis is more likely to lead to contrasting conclusions. This highlights the importance of establishing mandatory data archiving policies.

[**Wilkerson2012**] Jerod W. Wilkerson, Jay F. Nunamaker, and Rick Mercer. Comparing the defect reduction benefits of code inspection and test-driven development. *IEEE Transactions on Software Engineering*, 38(3):547–560, 5 2012, DOI `10.1109/tse.2011.46`.
Abstract: This study is a quasi experiment comparing the software defect rates and implementation costs of two methods of software defect reduction: code inspection and test-driven development. We divided participants, consisting of junior and senior computer science students at a large Southwestern university, into four groups using a two-by-two, between-subjects, factorial design and asked them to complete the same programming assignment using either test-driven development, code inspection, both, or neither. We compared resulting defect counts and implementation costs across groups. We found that code inspection is more effective than test-driven development at reducing defects, but that code inspection is also more expensive. We also found that test-driven development was no more effective at reducing defects than traditional programming methods.

[**Xu2015**] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software. In *Proc. International Symposium on the Foundations of Software Engineering (FSE)*. ACM, 8 2015, DOI `10.1145/2786805.2786852`.
Abstract: Configuration problems are not only prevalent, but also severely impair the reliability of today's system software. One fundamental reason is the ever-increasing complexity of configuration, reflected by the large number of configuration parameters ("knobs"). With hundreds of knobs, configuring system software to ensure high reliability and performance becomes a

162

daunting, error-prone task. This paper makes a first step in understanding a fundamental question of configuration design: "do users really need so many knobs?" To provide the quantitatively answer, we study the configuration settings of real-world users, including thousands of customers of a commercial storage system (Storage-A), and hundreds of users of two widely-used open-source system software projects. Our study reveals a series of interesting findings to motivate software architects and developers to be more cautious and disciplined in configuration design. Motivated by these findings, we provide a few concrete, practical guidelines which can significantly reduce the configuration space. Take Storage-A as an example, the guidelines can remove 51.9% of its parameters and simplify 19.7% of the remaining ones with little impact on existing users. Also, we study the existing configuration navigation methods in the context of "too many knobs" to understand their effectiveness in dealing with the over-designed configuration, and to provide practices for building navigation support in system software.

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

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

[**Yin2011**] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proc. Inter-*

national Symposium on Foundations of Software Engineering/International Symposium on the Foundations of Software Engineering (SIGSOFT/FSE). ACM Press, 2011, DOI `10.1145/2025113.2025121`.

Abstract: Software bugs affect system reliability. When a bug is exposed in the field, developers need to fix them. Unfortunately, the bug-fixing process can also introduce errors, which leads to buggy patches that further aggravate the damage to end users and erode software vendors' reputation. This paper presents a comprehensive characteristic study on incorrect bug-fixes from large operating system code bases including Linux, OpenSolaris, FreeBSD and also a mature commercial OS developed and evolved over the last 12 years, investigating not only themistake patterns during bug-fixing but also the possible human reasons in the development process when these incorrect bug-fixes were introduced. Our major findings include: (1) at least 14.8%–24.4% of sampled fixes for post-release bugs in these large OSes are incorrect and have made impacts to end users. (2) Among several common bug types, concurrency bugs are the most difficult to fix correctly: 39% of concurrency bug fixes are incorrect. (3) Developers and reviewers for incorrect fixes usually do not have enough knowledge about the involved code. For example, 27% of the incorrect fixes are made by developers who have never touched the source code files associated with the fix. Our results provide useful guidelines to design new tools and also to improve the development process. Based on our findings, the commercial software vendor whose OS code we evaluated is building a tool to improve the bug fixing and code reviewing process.

[**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.

[**Yu2021**] Zhongxing Yu, Chenggang Bai, Lionel Seinturier, and Martin Monperrus. Characterizing the usage, evolution and impact of java annotations in practice. *IEEE Transactions on Software Engineering*, 47(5):969–986, 5 2021, DOI `10.1109/tse.2019.2910516`.

Abstract: Annotations have been formally introduced into Java since Java 5. Since then, annotations have been widely used by the Java community for different purposes, such as compiler guidance and runtime processing. Despite the ever-growing use, there is still limited empirical knowledge about the actual usage of annotations in practice, the changes made to annotations during software evolution, and the potential impact of annotations on code quality. To fill this gap, we perform the first large-scale empirical study about Java annotations on 1,094 notable open-source projects hosted on GitHub. Our study systematically investigates annotation usage, annotation evolution, and annotation impact, and generates 10 novel and important findings. We also present the implications of our findings, which shed light for developers, researchers, tool builders, and language or library designers in order to improve all facets of Java annotation engineering.

[**Yuan2014**] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures—an analysis of production failures in distributed data-intensive systems. In *Proc. Symposium on Operating System Design and Implementation (OSDI)*, 2014, DOI `10.13140/2.1.2044.2889`.

Abstract: Large, production quality distributed systems still fail periodically, and do so sometimes catastrophically, where most or all users experience an outage or data loss. We present the result of a comprehensive study investigating 198 randomly selected, user-reported failures that occurred on Cassandra, HBase, Hadoop Distributed File System (HDFS), Hadoop MapReduce, and Redis, with the goal of understanding how one or multiple faults eventually evolve into a user-visible failures. We found that from a testing point of view, almost all failures require only 3 or fewer nodes to reproduce, which is good news considering that these services typically run on a very large number of nodes. However, multiple inputs are needed to trigger the failures with the order between them being important. Finally, we found the error logs of these systems typically contain sufficient data on both the errors and the input events that triggered the failure, enabling the diagnose and the reproduction of the production failures—often with unit tests. We found the majority of catastrophic failures could easily have been prevented by performing simple testing on error handling code—the last line of defense—even without an understanding of the software design. We extracted three simple rules from the bugs that have lead to some of the catastrophic failures, and developed a static checker, Aspirator, capable of

locating these bugs. Over 30% of the catastrophic failures would have been prevented had Aspirator been used and the identified bugs fixed. Running Aspirator on the code of 9 distributed systems located 143 bugs and bad practices that have been fixed or confirmed by the developers.

[**Zampetti2020**] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering*, 25(2):1095–1135, 1 2020, DOI `10.1007/s10664-019-09785-8`.
Abstract: Continuous Integration (CI) has been claimed to introduce several benefits in software development, including high software quality and reliability. However, recent work pointed out challenges, barriers and bad practices characterizing its adoption. This paper empirically investigates what are the bad practices experienced by developers applying CI. The investigation has been conducted by leveraging semi-structured interviews of 13 experts and mining more than 2,300 Stack Overflow posts. As a result, we compiled a catalog of 79 CI bad smells belonging to 7 categories related to different dimensions of a CI pipeline management and process. We have also investigated the perceived importance of the identified bad smells through a survey involving 26 professional developers, and discussed how the results of our study relate to existing knowledge about CI bad practices. Whilst some results, such as the poor usage of branches, confirm existing literature, the study also highlights uncovered bad practices, e.g., related to static analysis tools or the abuse of shell scripts, and contradict knowledge from existing literature, e.g., about avoiding nightly builds. We discuss the implications of our catalog of CI bad smells for (i) practitioners, e.g., favor specific, portable tools over hacking, and do not ignore nor hide build failures, (ii) educators, e.g., teach CI culture, not just technology, and teach CI by providing examples of what not to do, and (iii) researchers, e.g., developing support for failure analysis, as well as automated CI bad smell detectors.

[**Zavgorodniaia2021**] Albina Zavgorodniaia, Raj Shrestha, Juho Leinonen, Arto Hellas, and John Edwards. Morning or evening? an examination of circadian rhythms of CS1 students. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 5 2021, DOI `10.1109/icse-seet52601.2021.00036`.
Abstract: Circadian rhythms are the cycles of our internal clock that play a key role in governing when we sleep and when we are active. A related concept is chronotype, which is a person's natural tendency toward activity at certain times of day and typically governs when the individual is most alert and productive. In this work we investigate chronotypes in the setting of an Introductory Computer Programming (CS1) course. Using keystroke data collected from students we investigate the existence of chronotypes through unsupervised learning. The chronotypes we find align with those of typical populations reported in the literature and our results support correlations of certain chronotypes to academic achievement. We also find a lack of support

for the still-popular stereotype of a computer programmer as a night owl. The analyses are conducted on data from two universities, one in the US and one in Europe, that use different teaching methods. In comparison of the two contexts, we look into programming assignment design and administration that may promote better programming practices among students in terms of procrastination and effort.

[**Zhang2020**] Haoxiang Zhang, Shaowei Wang, Tse-Hsun Chen, and Ahmed E. Hassan. Reading answers on stack overflow: Not enough! *IEEE Transactions on Software Engineering*, pages 1–1, 2020, DOI `10.1109/tse.2019.2954319`.

Abstract: Stack Overflow is one of the most active communities for developers to share their programming knowledge. Answers posted on Stack Overflow help developers solve issues during software development. In addition to posting answers, users can also post comments to further discuss their associated answers. As of Aug 2017, there are 32.3 million comments that are associated with answers, forming a large collection of crowdsourced repository of knowledge on top of the commonly-studied Stack Overflow answers. In this study, we wish to understand how the commenting activities contribute to the crowdsourced knowledge. We investigate what users discuss in comments, and analyze the characteristics of the commenting dynamics, (i.e., the timing of commenting activities and the roles of commenters). We find that: 1) the majority of comments are informative and thus can enhance their associated answers from a diverse range of perspectives. However, some comments contain content that is discouraged by Stack Overflow. 2) The majority of commenting activities occur after the acceptance of an answer. More than half of the comments are fast responses occurring within one day of the creation of an answer, while later comments tend to be more informative. Most comments are rarely integrated back into their associated answers, even though such comments are informative. 3) Insiders (i.e., users who posted questions/answers before posting a comment in a question thread) post the majority of comments within one month, and outsiders (i.e., users who never posted any question/answer before posting a comment) post the majority of comments after one month. Inexperienced users tend to raise limitations and concerns while experienced users tend to enhance the answer through commenting. Our study provides insights into the commenting activities in terms of their content, timing, and the individuals who perform the commenting. For the purpose of long-term knowledge maintenance and effective information retrieval for developers, we also provide actionable suggestions to encourage Stack Overflow users/engineers/moderators to leverage our insights for enhancing the current Stack Overflow commenting system for improving the maintenance and organization of the crowdsourced knowledge.

[**Zhang2021a**] Jingxuan Zhang, He Jiang, Zhilei Ren, Tao Zhang, and Zhiqiu Huang. Enriching API documentation with code samples and usage scenarios from crowd knowledge. *IEEE Transactions on Software Engineering*, 47(6):1299–1314, 6 2021, DOI `10.1109/tse.2019.2919304`.

Abstract: As one key resource to learn Application Programming Interfaces (APIs), a lot of API reference documentation lacks code samples with usage scenarios, thus heavily hindering developers from programming with APIs. Although researchers have investigated how to enrich API documentation with code samples from general code search engines, two main challenges remain to be resolved, including the quality challenge of acquiring high-quality code samples and the mapping challenge of matching code samples to usage scenarios. In this study, we propose a novel approach named ADECK towards enriching API documentation with code samples and corresponding usage scenarios by leveraging crowd knowledge from Stack Overflow, a popular technical Question and Answer (Q&A) website attracting millions of developers. Given an API related Q&A pair, a code sample in the answer is extensively evaluated by developers and targeted towards resolving the question under the specified usage scenario. Hence, ADECK can obtain high-quality code samples and map them to corresponding usage scenarios to address the above challenges. Extensive experiments on the Java SE and Android API documentation show that the number of code-sample-illustrated API types in the ADECK-enriched API documentation is 3.35 and 5.76 times as many as that in the raw API documentation. Meanwhile, the quality of code samples obtained by ADECK is better than that of code samples by the baseline approach eXoaDocs in terms of correctness, conciseness, and usability, e.g., the average correctness values of representative code samples obtained by ADECK and eXoaDocs are 4.26 and 3.28 on a 5-point scale in the enriched Java SE API documentation. In addition, an empirical study investigating the impacts of different types of API documentation on the productivity of developers shows that, compared against the raw and the eXoaDocs-enriched API documentation, the ADECK-enriched API documentation can help developers complete 23.81 and 14.29 percent more programming tasks and reduce the average completion time by 9.43 and 11.03 percent.

[**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.

[**Zhu2021**] Wenhan Zhu and Michael W. Godfrey. Mea culpa: how developers fix their own simple bugs differently from other developers. In *Proc. International Conference on Mining Software Repositories (MSR)*. IEEE, 5 2021, DOI `10.1109/msr52588.2021.00065`.
Abstract: In this work, we study how the authorship of code affects bug-fixing commits using the SStuBs dataset, a collection of single-statement bug fix changes in popular Java Maven projects. More specifically, we study the differences in characteristics between simple bug fixes by the original author—that is, the developer who submitted the bug-inducing commit—and by different developers (i.e., non-authors). Our study shows that nearly half (i.e., 44.3%) of simple bugs are fixed by a different developer. We found that bug fixes by the original author and by different developers differed qualitatively and quantitatively. We observed that bug-fixing time by authors is much shorter than that of other developers. We also found that bug-fixing commits by authors tended to be larger in size and scope, and address multiple issues, whereas bug-fixing commits by other developers tended to be smaller and more focused on the bug itself. Future research can further study the different patterns in bug-fixing and create more tailored tools based on the developer's needs.

[**Zieris2020**] Franz Zieris and Lutz Prechelt. Explaining pair programming session dynamics from knowledge gaps. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 6 2020, DOI `10.1145/3377811.3380925`.
Abstract: Background: Despite a lot of research on the effectiveness of Pair Programming (PP), the question when it is useful or less useful remains unsettled. Method: We analyze recordings of many industrial PP sessions with Grounded Theory Methodology and build on prior work that identified various phenomena related to within-session knowledge build-up and transfer. We validate our findings with practitioners. Result: We identify two fundamentally different types of required knowledge and explain how different constellations of knowledge gaps in these two respects lead to different session dynamics. Gaps in project-specific systems knowledge are more hampering than gaps in general programming knowledge and are dealt with first and foremost in a PP session. Conclusion: Partner constellations with complementary knowledge make PP a particularly effective practice. In PP sessions, differences in system understanding are more important than differences in general software development knowledge.