It Will Never Work in Theory

http://neverworkintheory.org

March 24, 2023

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, Jun 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, Aug 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.

[**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, May 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, Jun 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, May 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, Feb 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, Nov 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.

[**AlencarDaCosta2023**] Daniel Alencar da Costa, Natalie Grattan, Nigel Stanger, and Sherlock A. Licorish. Studying the characteristics of SQL-related development tasks: An empirical study, 2023.
Abstract: A key function of a software system is its ability to facilitate the manipulation of data, which is often implemented using a flavour of the Structured Query Language (SQL). To develop the data operations of software (i.e, creating, retrieving, updating, and deleting data), developers are required to excel in writing and combining both SQL and application code. The problem is that writing SQL code in itself is already challenging (e.g., SQL anti-patterns are commonplace) and combining SQL with application code (i.e., for SQL development tasks) is even more demanding. Meanwhile, we have little empirical understanding regarding the characteristics of SQL development tasks. Do SQL development tasks typically need more code changes? Do they typically have a longer time-to-completion? Answers to such questions would prepare the community for the potential challenges associated with such tasks. Our results obtained from 20 Apache projects reveal that SQL development tasks have a significantly longer time-to-completion than SQL-unrelated tasks and require significantly more code changes. Through our qualitative analyses, we observe that SQL development tasks require more spread out changes, effort in reviews and documentation. Our results also corroborate previous research highlighting the prevalence of SQL anti-patterns. The software engineering community should make provision for the peculiarities of SQL coding, in the delivery of safe and secure interactive software.

[**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, Jun 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 re-

leases, 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, Jun 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, May 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.

[**Alsuhaibani2022**] Reem S. Alsuhaibani, Christian D. Newman, Michael J. Decker, Michael L. Collard, and Jonathan I. Maletic. An approach to automatically assess method names. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. ACM, May 2022, DOI 10.1145/3524610.3527780.

Abstract: An approach is presented to automatically assess the quality of method names by providing a score and feedback. The approach implements ten method naming standards to evaluate the names. The naming standards are taken from work that validated the standards via a large survey of software professionals. Natural language processing techniques such as part-of-speech tagging, identifier splitting, and dictionary lookup are required to implement the standards. The approach is evaluated by first manually constructing a large golden set of method names. Each method name is rated by several developers and labeled as conforming to each standard or not. These ratings allow for comparing the results of the approach against expert assessment. Additionally, the approach is applied to several systems and the results are manually inspected for accuracy.

[**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, Feb 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, Sep 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, Nov 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, May 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 re-

vision 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, May 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, May 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.

[**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, Nov 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, Jun 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, Sep 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, May 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, Oct 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, Dec 2019, DOI `10.7717/peerj-cs.241`.
Abstract: Background One of the twelve agile principles is to build projects around motivated individuals and trust them to get the job done. Such agile teams must self-organize, but this involves conflict, making self-organization difficult. One area of difficulty is agreeing on everybody's role. Background What dynamics arise in a self-organizing team from the negotiation of everybody's role? Method We conceptualize observations from five agile teams (work observations, interviews) by Charmazian Grounded Theory Methodology. Results We define role as something transient and implicit, not fixed and named. The roles are characterized by the responsibilities and expectations of each team member. Every team member must understand and accept their own roles (Local role clarity) and everbody else's roles (Team-wide role clarity). Role clarity allows a team to work smoothly and effectively and to develop its members' skills fast. Lack of role clarity creates friction that not only hampers the day-to-day work, but also appears to lead to high employee turnover. Agile coaches are critical to create and maintain role clarity. Conclusions Agile teams should pay close attention to the levels of Local role clarity of each member and Team-wide role clarity overall, because role clarity deficits are highly detrimental.

[**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, Jun 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, Dec 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, May 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, Oct 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, Aug 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, Mar 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, Oct 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, May 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 information 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 sys-

tems, which should focus on engaging bug reporters, better tool support, and improved handling of bug duplicates.

[**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.

[**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, Jul 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, Aug 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, Nov 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, Aug 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.

[**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.

[**Businge2022**] John Businge, Moses Openja, Sarah Nadi, and Thorsten Berger. Reuse and maintenance practices among divergent forks in three software ecosystems. *Empirical Software Engineering*, 27(2), Mar 2022, DOI 10.1007/s10664-021-10078-2.

Abstract: With the rise of social coding platforms that rely on distributed version control systems, software reuse is also on the rise. Many software developers leverage this reuse by creating variants through forking, to account for different customer needs, markets, or environments. Forked variants then form a so-called software family; they share a common code base and are maintained in parallel by same or different developers. As such, software families can easily arise within software ecosystems, which are large collections of interdependent software components maintained by communities of collaborating contributors. However, little is known about the existence and characteristics of such families within ecosystems, especially about their maintenance practices. Improving our empirical understanding of such families will help build better tools for maintaining and evolving such families. We empirically explore maintenance practices in such fork-based software families within ecosystems of open-source software. Our focus is on three of the largest software ecosystems existence today: Android, .NET, and JavaScript. We identify and analyze software families that are maintained together and that exist both on the official distribution platform (Google play, nuget, and npm) as well as on GitHub , allowing us to analyze reuse practices in depth. We mine and identify 38 software families, 526 software families, and 8,837 software families from the ecosystems of Android, .NET, and JavaScript, to study their characteristics and code-propagation practices. We provide scripts for analyzing code integration within our families. Interestingly, our results show that there is little code integration across the studied software families from the three ecosystems. Our studied families also show that techniques of direct integration using git outside of GitHub is more commonly used than GitHub pull requests. Overall, we hope to raise awareness about the existence of software families within larger ecosystems of software, calling for further research and better tools support to effectively maintain and evolve them.

[**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.

[**Campos2017**] Eduardo Cunha Campos and Marcelo de Almeida Maia. Common bug-fix patterns: a large-scale observational study. In *Proc. International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, Nov 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.

[**CardosoPereira2023**] Isadora Cardoso-Pereira, Geraldo Gomes, Danilo Monteiro Ribeiro, Alberto de Souza, Danilo Lucena, and Gustavo Pinto. Supporting the careers of developers with disabilities: Lessons from Zup Innovation, 2023.
Abstract: Software developers with disabilities have a hard time to join the software development market. Due to the lack of diversity that developers with disabilities could hinder innovation. In this work, we explore the Catalisa program envisioned by Zup Innovation, a Brazilian tech company, aimed to hire and train software developers with disabilities. We found that the program was able to accelerate the participants careers, although some shortcomings are still present.

[**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, May 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, Apr 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, May 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.

[**Codoban2015**] Mihai Codoban, Sruti Srinivasa Ragavan, Danny Dig, and Brian Bailey. Software history under the lens: A study on why and how developers examine it. In *Proc. International Conference on Software Maintenance (ICSM)*. IEEE, Sep 2015, DOI 10.1109/icsm.2015.7332446.
Abstract: Despite software history being indispensable for developers, there is little empirical knowledge about how they examine software history. Without such knowledge, researchers and tool builders are in danger of making wrong assumptions and building inadequate tools. In this paper we present an in-depth empirical study about the motivations developers have for examining software history, the strategies they use, and the challenges they encounter. To learn these, we interviewed 14 experienced developers from industry, and then extended our findings by surveying 217 developers. We found that history does not begin with the latest commit but with uncommitted changes. Moreover, we found that developers had different motivations for examining recent and old history. Based on these findings we propose 3-LENS HISTORY, a novel unified model for reasoning about software history.

[**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, Aug 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.

[**Costa2019**] Diego Elias 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.

[**Costa2022**] Keila Costa, Ronivaldo Ferreira, Gustavo Pinto, Marcelo d'Amorim, and Breno Miranda. Test flakiness across programming languages. *IEEE Transactions on Software Engineering*, pages 1–14, 2022, DOI 10.1109/tse.2022.3208864.

Abstract: Regression Testing (RT) is a quality-assurance practice commonly adopted in the software industry to check if functionality remains intact after code changes. Test flakiness is a serious problem for RT. A test is said to be flaky when it non-deterministically passes or fails on a fixed environment. Prior work studied test flakiness primarily on Java programs. It is unclear, however, how problematic is test flakiness for software written in other programming languages. This paper reports on a study focusing on three central aspects of test flakiness: concentration, similarity, and cost. Considering concentration, our results show that, for any given programming language that we studied (C, Go, Java, JS, and Python), most issues could be explained by a small fraction of root causes (5/13 root causes cover 78.07% of the issues) and could be fixed by a relatively small fraction of fix strategies (10/23 fix strategies cover 85.20% of the issues). Considering similarity, although there were commonalities in root causes and fixes across languages (e.g., concurrency and async wait are common causes of flakiness in most languages), we also found important differences (e.g., flakiness due to improper release of resources are more common in C), suggesting that

there is opportunity to fine tuning analysis tools. Considering cost, we found that issues related to flaky tests are resolved either very early once they are posted (¡10 days), suggesting relevance, or very late (¿100 days), suggesting irrelevance.

[**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, Feb 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, Jun 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 permission. 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, May 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, Aug 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, Dec

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.

[**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, May 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.

[**Decan2021a**] Alexandre Decan and Tom Mens. Lost in zero space –

an empirical comparison of 0.y.z releases in software package distributions. *Science of Computer Programming*, 208:102656, Aug 2021, DOI 10.1016/j.scico.2021.102656.
Abstract: Distributions of open source software packages dedicated to specific programming languages facilitate software development by allowing software projects to depend on the functionality provided by such reusable packages. The health of a software project can be affected by the maturity of the packages on which it depends. The version numbers of the used package releases provide an indication of their maturity. Packages with a 0.y.z version number are commonly assumed to be under initial development, suggesting that they are likely to be less stable, and depending on them may be considered as less healthy. In this paper, we empirically study, for four open source package distributions (Cargo, npm, Packagist and RubyGems) to which extent 0.y.z package releases and ≥1.0.0 package releases behave differently. We quantify the prevalence of 0.y.z releases, we explore how long packages remain in the initial development stage, we compare the update frequency of 0.y.z and ≥1.0.0 package releases, we study how often 0.y.z releases are required by other packages, we assess whether semantic versioning is respected for dependencies towards them, and we compare some characteristics of 0.y.z and ≥1.0.0 package repositories hosted on GitHub. Among others, we observe that package distributions are more permissive than what semantic versioning dictates for 0.y.z releases, and that many of the 0.y.z releases can actually be regarded as mature packages. As a consequence, the version number does not provide a good indication of the maturity of a package release.

[**Decan2021b**] Alexandre Decan and Tom Mens. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*, 47(6):1226–1240, Jun 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.

[**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, Jun 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.

[**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, May 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).

[**Eghbal2020**] Nadia Eghbal. *Working in Public: The Making and Maintenance of Open Source Software*. Stripe Press, 2020.

Abstract: An inside look at modern open source software developers and their applications to, and influence on, our online social world.

[**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, Aug 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, Jul 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.

[**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 Dy-*

namic Languages (DL). ACM, Oct 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.

[**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, Nov 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, Thomas Zimmermann, Christian Bird, and Nachiappan Nagappan. Characterizing software engineering work with personas based on knowledge worker actions. In *Proc. International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, Nov 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, May 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.

[**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.

[**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, May 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.

[**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, Sep 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, Jan 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.

[**Gallagher2017**] Kevin Gallagher, Sameer Patil, and Nasir D. Memon. New me: Understanding expert and non-expert perceptions and usage of the tor anonymity network. In *Proc. Symposium on Usable Privacy and Security (SOUPS)*, pages 385–398. USENIX Association, 2017.

Abstract: Proper use of an anonymity system requires adequate understanding of how it functions. Yet, there is surprisingly little research that looks into user understanding and usage of anonymity software. Improper use stemming from a lack of sufficient knowledge of the system has the potential to lead to deanonymization, which may hold severe personal consequences for the user. We report on the understanding and the use of the Tor anonymity system. Via semi-structured interviews with 17 individuals (6 experts and 11 non-experts) we found that experts and non-experts view, understand, and use Tor in notably different ways. Moreover, both groups exhibit behavior as well as gaps in understanding that could potentially compromise anonymity. Based on these findings, we provide several suggestions for im-

proving the user experience of Tor to facilitate better user understanding of its operation, threat model, and limitations.

[**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, May 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, Aug 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.

[**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, May 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.

[**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, Aug 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, Oct 2020, DOI 10.1145/3320269.3384745.

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

[**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, Aug 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, May 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, Mar 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 psychological 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, Jun 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.).

[**Gu2018**] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, May 2018, DOI `10.1145/3180155.3180167`.
Abstract: To implement a program functionality, developers can reuse previously written code snippets by searching through a large-scale codebase. Over the years, many code search tools have been proposed to help developers. The existing approaches often treat source code as textual documents and utilize information retrieval models to retrieve relevant code snippets that match a given query. These approaches mainly rely on the textual similarity between source code and natural language query. They lack a deep understanding of the semantics of queries and source code. In this paper, we propose a novel deep neural network named CODEnn (Code-Description

Embedding Neural Network). Instead of matching text similarity, CODEnn jointly embeds code snippets and natural language descriptions into a high-dimensional vector space, in such a way that code snippet and its corresponding description have similar vectors. Using the unified vector representation, code snippets related to a natural language query can be retrieved according to their vectors. Semantically related words can also be recognized and irrelevant/noisy keywords in queries can be handled. As a proof-of-concept application, we implement a code search tool named DeepCS using the proposed CODEnn model. We empirically evaluate DeepCS on a large scale codebase collected from GitHub. The experimental results show that our approach can effectively retrieve relevant code snippets and outperforms previous techniques.

[**Guler2019**] Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz. AntiFuzz: Impeding fuzzing audits of binary executables. In *Proc. USENIX Conference on Security Symposium*, pages 1931–1947, Aug 2019, DOI 10.5555/3361338.3361472.

Abstract: A general defense strategy in computer security is to increase the cost of successful attacks in both computational resources as well as human time. In the area of binary security, this is commonly done by using obfuscation methods to hinder reverse engineering and the search for software vulnerabilities. However, recent trends in automated bug finding changed the modus operandi. Nowadays it is very common for bugs to be found by various fuzzing tools. Due to ever-increasing amounts of automation and research on better fuzzing strategies, large-scale, dragnet-style fuzzing of many hundreds of targets becomes viable. As we show, current obfuscation techniques are aimed at increasing the cost of human understanding and do little to slow down fuzzing. In this paper, we introduce several techniques to protect a binary executable against an analysis with automated bug f inding approaches that are based on fuzzing, symbolic/concolic execution, and taint-assisted fuzzing (commonly known as hybrid fuzzing). More specifically, we perform a systematic analysis of the fundamental assumptions of bug finding tools and develop general countermeasures for each assumption. Note that these techniques are not designed to target specific implementations of fuzzing tools, but address general assumptions that bug finding tools necessarily depend on. Our evaluation demonstrates that these techniques effectively impede fuzzing audits, while introducing a negligible performance overhead. Just as obfuscation techniques increase the amount of human labor needed to find a vulnerability, our techniques render automated fuzzing-based approaches futile.

[**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, May 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, May 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 in-

ternationalization 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, Dec 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, Jan 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, Aug 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, May 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, May 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.

[**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, May 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, May 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, May 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, May 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, Sep 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, Apr 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.

[**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, Feb 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.

[**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, May 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, May 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.

[**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.

[**Inozemtseva2014**] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, May 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.

[**Jesse2022**] Kevin Jesse, Premkumar Devanbu, and Anand Ashok Sawant. Learning to predict user-defined types. *IEEE Transactions on Software Engineering*, pages 1–1, 2022, DOI 10.1109/tse.2022.3178945.
Abstract: TypeScript is a widely adopted gradual typed language where developers can optionally type variables, functions, parameters and more. Probabilistic type inference approaches with ML (machine learning) work well especially for commonly occurring types such as boolean, number, and string. TypeScript permits a wide range of types including developer defined class names and type interfaces. These developer defined types, termed user-defined types, can be written within the realm of language naming conventions. The set of user-defined types is boundless and existing bounded type guessing approaches are an imperfect solution. Existing works either under perform in user-defined types or ignore user-defined types altogether. This work leverages a BERT-style pre-trained model, with multi-task learning objectives, to learn how to type user-defined classes and interfaces. Thus we present DIVERSETYPER, a solution that explores the diverse set of user-defined types by uniquely aligning classes and interfaces declarations to the places in which they are used. DIVERSETYPER surpasses all existing works including those that model user-defined types.

[**Jiang2022**] Yuan Jiang, Christian Kastner, and Shurui Zhou. Elevating Jupyter notebook maintenance tooling by identifying and extracting notebook structures. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Oct 2022, DOI 10.1109/icsme55016.2022.00047.

Abstract: Data analysis is an exploratory, interactive, and often collaborative process. Computational notebooks have become a popular tool to support this process, among others because of their ability to interleave code, narrative text, and results. However, notebooks in practice are often criticized as hard to maintain and being of low code quality, including problems such as unused or duplicated code and out-of-order code execution. Data scientists can benefit from better tool support when maintaining and evolving notebooks. We argue that central to such tool support is identifying the structure of notebooks. We present a lightweight and accurate approach to extract notebook structure and outline several ways such structure can be used to improve maintenance tooling for notebooks, including navigation and finding alternatives.

[**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, May 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.

[**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, Sep 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.

[**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, Sep 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, May 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.

[**Jovanovic2022**] Ana Jovanovic and Allison Sullivan. Towards automated input generation for sketching alloy models. In *Proceedings of the IEEE/ACM 10th International Conference on Formal Methods in Software Engineering*. ACM, May 2022, DOI `10.1145/3524482.3527651`.
Abstract: Writing declarative models has numerous benefits, ranging from automated reasoning and correction of design-level properties before systems are built, to automated testing and debugging of their implementations after they are built. Alloy is a declarative modeling language that is well suited for verifying system designs. While Alloy comes deployed in the Analyzer, an automated scenario-finding tool set, writing correct models remains a difficult and error-prone task. ASketch is a synthesis framework that helps users build their Alloy models. ASketch takes as an input a partial Alloy models with holes and an AUnit test suite. As output, ASketch returns a completed model that passes all tests. ASketch's initial evaluation reveals ASketch to be a promising approach to synthesize Alloy models. In this paper, we present and explore SketchGen2, an approach that looks to broaden the adoption of ASketch by increasing the automation of the inputs needed for the sketching process. Experimental results show SketchGen2 is effective at producing both expressions and test suites for synthesis.

[**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, May 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, Jul 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, May 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 con-

flict 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, May 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, Jun 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, Oct 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.

[**Kim2013a**] 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, May 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 approach 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.

[**Kim2013b**] Jinhan Kim, Sanghoon Lee, Seung-Won Hwang, and Sunghun Kim. Enriching documents with examples. *ACM Transactions on Information Systems*, 31(1):1–27, Jan 2013, DOI 10.1145/2414782.2414783.

Abstract: Software developers increasingly rely on information from the Web, such as documents or code examples on application programming interfaces (APIs), to facilitate their development processes. However, API documents often do not include enough information for developers to fully understand how to use the APIs, and searching for good code examples requires considerable effort. To address this problem, we propose a novel code example recommendation system that combines the strength of browsing documents and searching for code examples and returns API documents embedded with high-quality code example summaries mined from the Web. Our evaluation results show that our approach provides code examples with high precision and boosts programmer productivity.

[**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, Oct 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.

[**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, Jul 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, May 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.

[**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, Nov 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 $9\times10 = 90$ 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, Aug 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, Feb 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, Apr 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.

[**Kula2022a**] Elvan Kula, Eric Greuter, Arie van Deursen, and Georgios Gousios. Factors affecting on-time delivery in large-scale agile software development. *IEEE Transactions on Software Engineering*, 48(9):3573–3592, Sep 2022, DOI 10.1109/tse.2021.3101192.
Abstract: Late delivery of software projects and cost overruns have been common problems in the software industry for decades. Both problems are manifestations of deficiencies in effort estimation during project planning. With software projects being complex socio-technical systems, a large pool of factors can affect effort estimation and on-time delivery. To identify the most relevant factors and their interactions affecting schedule deviations in large-scale agile software development, we conducted a mixed-methods case study at ING: two rounds of surveys revealed a multitude of organizational, people, process, project and technical factors which were then quantified and statistically modeled using software repository data from 185 teams. We find that factors such as requirements refinement, task dependencies, organizational alignment and organizational politics are perceived to have the greatest impact on on-time delivery, whereas proxy measures such as project size, number of dependencies, historical delivery performance and team familiarity can help explain a large degree of schedule deviations. We also discover hierarchical interactions among factors: organizational factors are perceived to interact with people factors, which in turn impact technical factors. We compose our findings in the form of a conceptual framework representing influential factors and their relationships to on-time delivery. Our results can help practitioners identify and manage delay risks in agile settings, can inform the design of automated tools to predict schedule overruns and can contribute towards the development of a relational theory of software project management.

[**Kula2022b**] Raula Gaikovina Kula and Christoph Treude. In war and peace: The impact of world politics on software ecosystems, 2022.
Abstract: Reliance on third-party libraries is now commonplace in contemporary software engineering. Being open source in nature, these libraries should advocate for a world where the freedoms and opportunities of open source software can be enjoyed by all. Yet, there is a growing concern related to maintainers using their influence to make political stances (i.e., referred to as protestware). In this paper, we reflect on the impact of world politics on software ecosystems, especially in the context of the ongoing War in Ukraine. We show three cases where world politics has had an impact on a software ecosystem, and how these incidents may result in either benign or malignant consequences. We further point to specific opportunities for research, and conclude with a research agenda with ten research questions to guide future research directions.

[**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, May 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.

[**Leitao2019**] Roxanne Leit ao. Technology-facilitated intimate partner abuse: a qualitative analysis of data from online domestic abuse forums. *Human–Computer Interaction*, 36(3):203–242, Dec 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, May 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), Jan 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, May 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, May 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, Aug 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.

[**Licorish2022**] Sherlock A. Licorish and Markus Wagner. Combining GIN and PMD for code improvements. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM, Jul 2022, DOI 10.1145/3520304.3528772.
Abstract: Software developers are increasingly dependent on question and answer portals and blogs for coding solutions. While such interfaces provide useful information, there are concerns that code hosted here is often incorrect, insecure or incomplete. Previous work indeed detected a range of faults in code provided on Stack Overflow through the use of static analysis. Static analysis may go a far way towards quickly establishing the health of software

code available online. In addition, mechanisms that enable rapid automated program improvement may then enhance such code. Accordingly, we present this proof of concept. We use the PMD static analysis tool to detect performance faults for a sample of Stack Overflow Java code snippets, before performing mutations on these snippets using GIN. We then re-analyse the performance faults in these snippets after the GIN mutations. GIN's RandomSampler was used to perform 17,986 unique line and statement patches on 3,034 snippets where PMD violations were removed from 770 patched versions. Our outcomes indicate that static analysis techniques may be combined with automated program improvement methods to enhance publicly available code with very little resource requirements. We discuss our planned research agenda in this regard.

[**Lima2021**] 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), Jun 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.

[**Lin2020**] Sarah Lin, Ibraheem Ali, and Greg Wilson. Ten quick tips for making things findable. *PLOS Computational Biology*, 16(12):e1008469, Dec 2020, DOI `10.1371/journal.pcbi.1008469`.

Abstract: The distribution of scholarly content today happens in the context of an immense deluge of information found on the internet. As a result, researchers face serious challenges when archiving and finding information that relates to their work. Library science principles provide a framework for navigating information ecosystems in order to help researchers improve findability of their professional output. Here, we describe the information

ecosystem which consists of users, context, and content, all 3 of which must be addressed to make information findable and usable. We provide a set of tips that can help researchers evaluate who their users are, how to archive their research outputs to encourage findability, and how to leverage structural elements of software to make it easier to find information within and beyond their publications. As scholars evaluate their research communication strategies, they can use these steps to improve how their research is discovered and reused.

[**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, Aug 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.

[**Lopez2018**] Fernando López de la Mora and Sarah Nadi. An empirical study of metric-based comparisons of software libraries. In *Proc. International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*. ACM, Oct 2018, DOI 10.1145/3273934.3273937.
Abstract: Software libraries provide a set of reusable functionality, which helps developers write code in a systematic and timely manner. However, selecting the appropriate library to use is often not a trivial task. AIMS: In this paper, we investigate the usefulness of software metrics in helping developers choose libraries.Different developers care about different aspects of a library and two developers looking for a library in a given domain may not necessarily choose the same library. Thus, instead of directly recommending a library to use, we provide developers with a metric-based comparison of libraries in the same domain to empower them with the information they need to make an informed decision. METHOD:We use software data analytics from several sources of information to create quantifiable metric-based comparisons of software libraries. For evaluation, we select 34 open-source

Java libraries from 10 popular domains and extract nine metrics related to these libraries. We then conduct a survey of 61 developers to evaluate whether our proposed metric-based comparison is useful, and to understand which metrics developers care about. RESULTS: Our results show that developers find that the proposed technique provides useful information when selecting libraries. We observe that developers care the most about metrics related to the popularity, security, and performance of libraries. We also find that the usefulness of some metrics may vary according to the domain. CONCLUSIONS:Our survey results showed that our proposed technique is useful.We are currently building a public website for metric-based library comparisons, while incorporating the feedback we obtained from our survey participants.

[**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, Jun 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/.

[**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, Sep 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.

[**Madampe2022**] Kashumi Madampe, Rashina Hoda, and John Grundy. The emotional roller coaster of responding to requirements changes in software engineering. *IEEE Transactions on Software Engineering*, pages 1–1, 2022, DOI `10.1109/tse.2022.3172925`.

Abstract: Background: A preliminary study we conducted showed that software practitioners respond to requirements changes (RCs) with different emotions, and that their emotions vary at stages of the RC handling life cycle, such as receiving, developing, and delivering RCs. Furthermore, such developer emotions have direct linkages to cognition, productivity, and decision making. Therefore, it is important to gain a comprehensive understanding the role of emotions in a critical scenarios like handling RCs. Objective: We wanted to study how practitioners emotionally respond to RCs. Method: We conducted a world-wide survey with the participation of 201 software practitioners. In our survey, we used the Job-related Affective Well-being Scale (JAWS) and open-ended questions to capture participants emotions when handling RCs in their work and query about the different circumstances when they feel these emotions. We used a combined approach of statistical analysis, JAWS, and Socio-Technical Grounded Theory (STGT) for Data Analysis to analyse our survey data. Findings: We identified (1) emotional responses to RCs, i.e., the most common emotions felt by practitioners when handling RCs; (2) different stimuli – such as the RC, the practitioner, team, manager, customer – that trigger these emotions through their own different characteristics; (3) emotion dynamics, i.e., the changes in emotions during the RC handling life cycle; (4) RC stages where particular emotions are triggered; and (5) time related aspects that regulate the emotion dynamics. Conclusion: Practitioners are not pleased with receiving RCs all the time. Last minute RCs introduced closer to a deadline especially violate emotional well-being of practitioners. We present some practical recommendations for practitioners to follow, including a dual-purpose emotion-centric decision guide to help decide when to introduce or accept an RC, and some future key research directions.

[**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), Aug 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.

[**Mahmoudi2019**] Mehran Mahmoudi, Sarah Nadi, and Nikolaos Tsantalis. Are refactorings to blame? an empirical study of refactorings in merge conflicts. In *Proc. International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Feb 2019, DOI 10.1109/saner.2019.8668012.
Abstract: With the rise of distributed software development, branching has become a popular approach that facilitates collaboration between software developers. One of the biggest challenges that developers face when using multiple development branches is dealing with merge conflicts. Conflicts occur when inconsistent changes happen to the code. Resolving these conflicts can be a cumbersome task as it requires prior knowledge about the changes in each of the development branches. A type of change that could potentially lead to complex conflicts is code refactoring. Previous studies have proposed techniques for facilitating conflict resolution in the presence of refactorings. However, the magnitude of the impact that refactorings have on merge conflicts has never been empirically evaluated. In this paper, we perform an empirical study on almost 3,000 wellengineered open-source Java software repositories and investigate the relation between merge conflicts and 15 popular refactoring types. Our results show that refactoring operations are involved in 22% of merge conflicts, which is remarkable taking into account that we investigated a relatively small subset of all possible refactoring types. Furthermore, certain refactoring types, such as EXTRACT METHOD, tend to be more problematic for merge conflicts. Our results also suggest that conflicts that involve refactored code are usually more complex, compared to conflicts with no refactoring changes.

[**Majumder2019**] Suvodeep Majumder, Joymallya Chakraborty, Amritanshu Agrawal, and Tim Menzies. Communication and code dependency effects on software code quality: An empirical analysis of herbsleb hypothesis, 2019.

[**Majumder2021**] Suvodeep Majumder, Joymallya Chakraborty, Gina R. Bai, Kathryn T. Stolee, and Tim Menzies. Fair enough: Searching for sufficient measures of fairness, 2021.
Abstract: Testing machine learning software for ethical bias has become a pressing current concern. In response, recent research has proposed a plethora of new fairness metrics, for example, the dozens of fairness metrics in the IBM AIF360 toolkit. This raises the question: How can any fairness tool satisfy such a diverse range of goals? While we cannot completely simplify the task of fairness testing, we can certainly reduce the problem. This paper shows that many of those fairness metrics effectively measure the same

thing. Based on experiments using seven real-world datasets, we find that (a) 26 classification metrics can be clustered into seven groups, and (b) four dataset metrics can be clustered into three groups. Further, each reduced set may actually predict different things. Hence, it is no longer necessary (or even possible) to satisfy all fairness metrics. In summary, to simplify the fairness testing problem, we recommend the following steps: (1) determine what type of fairness is desirable (and we offer a handful of such types); then (2) lookup those types in our clusters; then (3) just test for one item per cluster.

[**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, Dec 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, Jul 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 André van der Hoek. How software designers interact with sketches at the whiteboard. *IEEE Transactions on Software Engineering*, 41(2):135–156, Feb 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.

[**Margulieux2020**] Lauren E. Margulieux, Briana B. Morrison, and Adrienne Decker. Reducing withdrawal and failure rates in introductory programming with subgoal labeled worked examples. *International Journal of STEM Education*, 7(1), May 2020, DOI 10.1186/s40594-020-00222-7.
Abstract: Background: Programming a computer is an increasingly valuable skill, but dropout and failure rates in introductory programming courses are regularly as high as 50%. Like many fields, programming requires students to learn complex problem-solving procedures from instructors who tend to have tacit knowledge about low-level procedures that they have automatized. The subgoal learning framework has been used in programming and other fields to breakdown procedural problem solving into smaller

pieces that novices can grasp more easily, but it has only been used in short-term interventions. In this study, the subgoal learning framework was implemented throughout a semester-long introductory programming course to explore its longitudinal effects. Of 265 students in multiple sections of the course, half received subgoal-oriented instruction while the other half received typical instruction. Results: Learning subgoals consistently improved performance on quizzes, which were formative and given within a week of learning a new procedure, but not on exams, which were summative. While exam performance was not statistically better, the subgoal group had lower variance in exam scores and fewer students dropped or failed the course than in the control group. To better understand the learning process, we examined students' responses to open-ended questions that asked them to explain the problem-solving process. Furthermore, we explored characteristics of learners to determine how subgoal learning affected students at risk of dropout or failure. Conclusions: Students in an introductory programming course performed better on initial assessments when they received instructions that used our intervention, subgoal labels. Though the students did not perform better than the control group on exams on average, they were less likely to get failing grades or to drop the course. Overall, subgoal labels seemed especially effective for students who might otherwise struggle to pass or complete the course.

[**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, Sep 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.

[**Mattmann2015**] Chris A. Mattmann, Joshua Garcia, Ivo Krka, Daniel Popescu, and Nenad Medvidović. Revisiting the anatomy and physiology of the grid. *Journal of Grid Computing*, 13(1):19–34, Jan 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.

[**McGee2011**] Sharon McGee and Des Greer. Software requirements change taxonomy: evaluation by case study. In *Proc. International Requirements Engineering Conference (RE)*. IEEE, Aug 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, May 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, Mar 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, Oct 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.

[**McNamara2018**] Andrew McNamara, Justin Smith, and Emerson Murphy-Hill. Does ACM's code of ethics change ethical decision making in software development? In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, Oct 2018, DOI `10.1145/3236024.3264833`.
Abstract: Ethical decisions in software development can substantially impact end-users, organizations, and our environment, as is evidenced by recent ethics scandals in the news. Organizations, like the ACM, publish codes of ethics to guide software-related ethical decisions. In fact, the ACM has recently demonstrated renewed interest in its code of ethics and made updates for the first time since 1992. To better understand how the ACM code of ethics changes software-related decisions, we replicated a prior behavioral

ethics study with 63 software engineering students and 105 professional software developers, measuring their responses to 11 ethical vignettes. We found that explicitly instructing participants to consider the ACM code of ethics in their decision making had no observed effect when compared with a control group. Our findings suggest a challenge to the research community: if not a code of ethics, what techniques can improve ethical decision making in software engineering?

[**Mehrpour2022**] Sahar Mehrpour and Thomas D. LaToza. Can static analysis tools find more defects? *Empirical Software Engineering*, 28(1), Nov 2022, DOI 10.1007/s10664-022-10232-4.

Abstract: Static analysis tools find defects in code, checking code against rules to reveal potential defects. Many studies have evaluated these tools by measuring their ability to detect known defects in code. But these studies measure the current state of tools rather than their future potential to find more defects. To investigate the prospects for tools to find more defects, we conducted a study where we formulated each issue raised by a code reviewer as a violation of a rule, which we then compared to what static analysis tools might potentially check. We first gathered a corpus of 1323 defects found through code review. Through a qualitative analysis process, for each defect we identified a violated rule and the type of Static Analysis Tool (SAT) which might check this rule. We found that SATs might, in principle, be used to detect as many as 76% of code review defects, considerably more than current tools have been demonstrated to successfully detect. Among a variety of types of SATs, Style Checkers and AST Pattern Checkers had the broadest coverage of defects, each with the potential to detect 25% of all code review defects. We found that static analysis tools might be able to detect more code review defects by better supporting the creation of project-specific rules. We also investigated the characteristics of code review defects not detectable by traditional static analysis techniques, which to detect might require tools which simulate human judgements about code.

[**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, May 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, Nov 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, Nov 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, May 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, Aug 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.

[**Miedema2022**] Daphne Miedema, George Fletcher, and Efthimia Aivaloglou. So many brackets!: an analysis of how SQL learners (mis)manage complexity during query formulation. In *Proc. International Conference on Program Comprehension (ICPC)*. ACM, May 2022, DOI 10.1145/3524610.3529158.
Abstract: The Structured Query Language (SQL) is a widely taught

database query language in computer science, data science, and software engineering programs. While highly expressive, SQL is challenging to learn for novices. Various research has explored the errors and mistakes that SQL users make. Specific attributes of SQL code, such as the number of tables and the degree of nesting, have been found to impact its understandability and maintainability. Furthermore, prior studies have shown that novices have significant issues using SQL correctly, due to factors such as expressive ease, existing knowledge and misconceptions, and the impact of cognitive load. In this paper we identify another factor: self-inflicted query complexity, where users hinder their own problem solving process. We analyse 8K intermediate and final student attempts to six SQL exer-cises, approaching complexity from four perspective: correctness, execution order, edit distance and query intricacy. Through our analyses, we find that our students are hindered in their query formulation process by mismanaging complexity through writing overly elaborate queries containing unnecessary elements, overusing brackets and nesting, and incrementally building queries with persistent errors.

[**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, Aug 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, May 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, May 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 organization 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, May 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, Jul 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, May 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.

[**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 Op-*

erating Systems (PLOS). ACM, Oct 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.

[**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, Feb 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, Aug 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 representative 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±5%) and cleaning up resources at the end of a procedure (40.36±5%); and (2) quantitatively analyze the commit history from the release branches of six OSS projects and find that no goto statement was removed/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.

[**Nakajo1991**] Takeshi Nakajo and Hitoshi Kume. A case history analysis of software error cause-effect relationships. *IEEE Transactions on Software Engineering*, 17(8):830–838, 1991, DOI 10.1109/32.83917.
Abstract: Approximately 700 errors in four commercial measuring-control software products were analyzed, and the cause-effect relationships of errors occurring during software development were identified. The analysis method used defined appropriate observation points along the path leading from cause to effect of a software error and gathered the corresponding data by analyzing each error using fault tree analysis. Each observation point's data were categorized, and the relationships between two adjoining points were summarized using a cross-indexing table. Four major error-occurrence mechanisms were identified; two are related to hardware and software interface specification misunderstandings, while the other two are related to system and module function misunderstandings. The effects of structured analysis and structured design methods on software errors were evaluated.

[**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, May 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 subclasses 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 unnoticed. Additionally, it is observed that classes higher in the exception class hierarchy are more frequently used as compared to specific exception subclasses.

[**Nassif2021**] Mathieu Nassif, Alexa Hernandez, Ashvitha Sridharan, and Martin P. Robillard. Generating unit tests for documentation. *IEEE Transactions on Software Engineering*, 2021, DOI 10.1109/tse.2021.3087087.
Abstract: Software projects capture redundant information in various kinds of artifacts, as specifications from the source code are also tested and documented. Such redundancy provides an opportunity to reduce development effort by supporting the joint generation of different types of artifact. We

introduce a tool-supported technique, called DScribe, that allows developers to combine unit tests and documentation templates, and to invoke those templates to generate documentation and unit tests. DScribe supports the detection and replacement of outdated documentation, and the use of templates can encourage extensive test suites with a consistent style. Our evaluation of 835 specifications revealed that 85% were not tested or correctly documented, and DScribe could be used to automatically generate 97% of the tests and documentation. An additional study revealed that tests generated by DScribe are more focused and readable than those written by human testers or generated by state-of-the-art automated techniques.

[**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, May 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.

[**Ni2021**] Ansong Ni, Daniel Ramos, Aidan Z. H. Yang, Ines Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. SOAR: A synthesis approach for data science API refactoring. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, May 2021, DOI 10.1109/icse43902.2021.00023.

Abstract: With the growth of the open-source data science community, both the number of data science libraries and the number of versions for the same library are increasing rapidly. To match the evolving APIs from those libraries, open-source organizations often have to exert manual effort to refactor the APIs used in the code base. Moreover, due to the abundance of similar open-source libraries, data scientists working on a certain application may have an abundance of libraries to choose, maintain and migrate between. The manual refactoring between APIs is a tedious and error-prone task. Although recent research efforts were made on performing automatic API refactoring between different languages, previous work relies on statistical learning with collected pairwise training data for the API matching and migration. Using large statistical data for refactoring is not ideal because such training data will not be available for a new library or a new version of the same library. We introduce Synthesis for OpenSource API Refactoring (SOAR), a novel technique that requires no training data to achieve API migration and refactoring. SOAR relies only on the documentation that is

readily available at the release of the library to learn API representations and mapping between libraries. Using program synthesis, SOAR automatically computes the correct configuration of arguments to the APIs and any glue code required to invoke those APIs. SOAR also uses the interpreter's error messages when running refactored code to generate logical constraints that can be used to prune the search space. Our empirical evaluation shows that SOAR can successfully refactor 80% of our benchmarks corresponding to deep learning models with up to 44 layers with an average run time of 97.23 seconds, and 90% of the data wrangling benchmarks with an average run time of 17.31 seconds.

[**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, Nov 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.

[**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, Apr 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.

[**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, Jun 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.

[**Pan2008**] Kai Pan, Sunghun Kim, and E. James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, Aug 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, Jun 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, Jun 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.

[**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, Aug 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.

[**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, Mar 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 (OOPSLA)*. ACM, Oct 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, May 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 cre-

ated 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, Sep 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, May 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.

[**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, Oct 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, Oct 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, Oct 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.

[**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, Sep 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, Mar 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 be-

ing 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, May 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, May 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.

[**Rahman2019**] Akond Rahman, Chris Parnin, and Laurie Williams. The seven sins: Security smells in infrastructure as code scripts. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, May 2019, DOI 10.1109/icse.2019.00033.
Abstract: Practitioners use infrastructure as code (IaC) scripts to provision servers and development environments. While developing IaC scripts, practitioners may inadvertently introduce security smells. Security smells are recurring coding patterns that are indicative of security weakness and can potentially lead to security breaches. The goal of this paper is to help practitioners avoid insecure coding practices while developing infrastructure as code (IaC) scripts through an empirical study of security smells in IaC scripts. We apply qualitative analysis on 1,726 IaC scripts to identify seven security smells. Next, we implement and validate a static analysis tool called Security Linter for Infrastructure as Code scripts (SLIC) to identify the occurrence of each smell in 15,232 IaC scripts collected from 293 open source repositories. We identify 21,201 occurrences of security smells that include 1,326 occurrences of hard-coded passwords. We submitted bug reports for 1,000 randomly-selected security smell occurrences. We obtain 212 responses to these bug reports, of which 148 occurrences were accepted by the development teams to be fixed. We observe security smells can have a long lifetime, e.g., a hard-coded secret can persist for as long as 98 months, with a median lifetime of 20 months.

[**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, Jun 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.

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

[**Riccomini2021**] Chris Riccomini and Dmitriy Ryaboy. *The Missing README: A Guide for the New Software Engineer*. No Starch Press, 2021.
Abstract: For new software engineers, knowing how to program is only

half the battle. You'll quickly find that many of the skills and processes key to your success are not taught in any school or bootcamp. The Missing README fills in that gap—a distillation of workplace lessons, best practices, and engineering fundamentals that the authors have taught rookie developers at top companies for more than a decade.

[**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, May 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, May 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, Nov 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, Aug 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, Dec 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, May 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, Jul 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 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.

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

[**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.

[**Sajadi2023**] Amirali Sajadi, Kostadin Damevski, and Preetha Chatterjee. Interpersonal Trust in OSS: Exploring Dimensions of Trust in GitHub Pull Requests. In *Proceedings of the 45th International Conference on Software Engineering (NIER Track)*, ICSE '23, 2023.
Abstract: Interpersonal trust plays a crucial role in facilitating collaborative tasks, such as software development. While previous research recognizes the significance of trust in an organizational setting, there is a lack of understanding in how trust is exhibited in OSS distributed teams, where there is an absence of direct, in-person communications. To foster trust and collaboration in OSS teams, we need to understand what trust is and how it is exhibited in written developer communications (e.g., pull requests, chats). In this paper, we first investigate various dimensions of trust to identify the ways trusting behavior can be observed in OSS. Next, we sample a set of 100 GitHub pull requests from Apache Software Foundation (ASF) projects, to analyze and demonstrate how each dimension of trust can be exhibited. Our findings provide preliminary insights into cues that might be helpful to automatically assess team dynamics and establish interpersonal trust in OSS teams, leading to successful and sustainable OSS.

[**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, Jun 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, Mar 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, Oct 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 et al. 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, Jul 2021, DOI `10.1016/j.obhdp.2021.02.003`.

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 devel-

opment waste. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, May 2017, DOI 10.1109/icse.2017.20.
Abstract: Context: Since software development is a complex socio-technical 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.

[**Sellitto2022**] Giulia Sellitto, Emanuele Iannone, Zadia Codabux, Valentina Lenarduzzi, Andrea De Lucia, Fabio Palomba, and Filomena Ferrucci. Toward understanding the impact of refactoring on program comprehension. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Mar 2022, DOI 10.1109/saner53432.2022.00090.
Abstract: Software refactoring is the activity associated with developers changing the internal structure of source code without modifying its external behavior. The literature argues that refactoring might have beneficial and harmful implications for software maintainability, primarily when performed without the support of automated tools. This paper continues the narrative on the effects of refactoring by exploring the dimension of program comprehension, namely the property that describes how easy it is for developers to understand source code. We start our investigation by assessing the basic unit of program comprehension, namely program readability. Next, we set up a large-scale empirical investigation—conducted on 156 open-source projects—to quantify the impact of refactoring on program readability. First, we mine refactoring data and, for each commit involving a refactoring, we compute (i) the amount and type(s) of refactoring actions performed and (ii) eight state-of-the-art program comprehension metrics. Afterwards, we build statistical models relating the various refactoring operations to each of the

readability metrics considered to quantify the extent to which each refactoring impacts the metrics in either a positive or negative manner. The key results are that refactoring has a notable impact on most of the readability metrics considered.

[**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, Sep 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, May 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, Aug 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, Sep 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.

[**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.

[**Soremekun2021**] Ezekiel Soremekun, Lukas Kirschner, Marcel Böhme, and Andreas Zeller. Locating faults with program slicing: an empirical analysis. *Empirical Software Engineering*, 26(3), Apr 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).

[**Spinellis2016**] Diomidis Spinellis, Panos Louridas, and Maria Kechagia. The evolution of c programming practices. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, May 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, Jun 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, May 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, Nov 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.

[**Stolee2011**] Kathryn T. Stolee and Sebastian Elbaum. Refactoring pipe-like mashups for end-user programmers. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, May 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.

[**Stylos2007**] Jeffrey Stylos and Steven Clarke. Usability implications of requiring parameters in objects' constructors. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, May 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.

[**Sven2019**] Amann Sven, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. Investigating next steps in static API-misuse detection. In *Proc. International Conference on Mining Software Repositories (MSR)*. IEEE, May 2019, DOI 10.1109/msr.2019.00053.
Abstract: Application Programming Interfaces (APIs) often impose constraints such as call order or preconditions. API misuses, i.e., usages violating these constraints, may cause software crashes, data-loss, and vulnerabilities. Researchers developed several approaches to detect API misuses, typically still resulting in low recall and precision. In this work, we investigate ways to improve API-misuse detection. We design MUDetect, an API-misuse detector that builds on the strengths of existing detectors and tries to mitigate their weaknesses. MUDetect uses a new graph representation of API usages that captures different types of API misuses and a systematically designed ranking strategy that effectively improves precision. Evaluation shows that MUDetect identifies real-world API misuses with twice the recall of previous detectors and 2.5x higher precision. It even achieves almost 4x higher precision and recall, when mining patterns across projects, rather than from only the target project.

[**Taipalus2018**] Toni Taipalus, Mikko Siponen, and Tero Vartiainen. Errors and complications in SQL query formulation. *ACM Transactions on Computing Education*, 18(3):1–29, Sep 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, Nov 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, Jul 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.

[**Tang2021**] Henry Tang and Sarah Nadi. On using stack overflow comment-edit pairs to recommend code maintenance changes. *Empirical Software Engineering*, 26(4), May 2021, DOI `10.1007/s10664-021-09954-8`.

Abstract: Code maintenance data sets typically consist of a before version of the code and an after version that contains the improvement or fix. Such data sets are important for various software engineering support tools related to code maintenance, such as program repair, code recommender systems, or Application Programming Interface (API) misuse detection. Most of the current data sets are typically constructed from mining commit history in versioncontrol systems or issues in issue-tracking systems. In this paper, we

investigate whether Stack Overflow can be used as an additional source for building code maintenance data sets. Comments on Stack Overflow provide an effective way for developers to point out problems with existing answers, alternative solutions, or pitfalls. Given its crowd-sourced nature, answers are then updated to incorporate these suggestions. In this paper, we mine commentedit pairs from Stack Overflow and investigate their potential usefulness for constructing the above data sets. These comment-edit pairs have the added benefit of having concrete descriptions/explanations of why the change is needed as well as potentially having less tangled changes to deal with. We first design a technique to extract related comment-edit pairs and then qualitatively and quantitatively investigate the nature of these pairs. We find that the majority of comment-edit pairs are not tangled, but find that only 27% of the studied pairs are potentially useful for the above applications. We categorize the types of mined pairs and find that the highest ratio of useful pairs come from those categorized as Correction, Obsolete, Flaw, and Extension. These categories can provide data for both corrective and preventative maintenance activities. To demonstrate the effectiveness of our extracted pairs, we submitted 15 pull requests to popular GitHub repositories, 10 of which have been accepted to widely used repositories such as Apache Beam (https://beam.apache.org/) and NLTK (https://www.nltk.org/). Our work is the first to investigate Stack Overflow commentedit pairs and opens the door for future work in this direction. Based on our findings and observations, we provide concrete suggestions on how to potentially identify a larger set of useful comment-edit pairs, which can also be facilitated by our shared data.

[**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, Aug 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 help-

ful 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, May 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, Aug 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 Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar 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, May 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, Feb 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, Mar 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, Jul 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.

[**Tshukudu2023**] Ethel Tshukudu, Sue Sentance, Oluwatoyin Adelakun-Adeyemo, Brenda Nyaringita, Keith Quille, and Ziling Zhong. Investigating K-12 computing education in four African countries (Botswana, Kenya, Nigeria, and Uganda). *ACM Transactions on Computing Education*, 23(1):1–29, Jan 2023, DOI 10.1145/3554924.
Abstract: Motivation. As K-12 computing education becomes more established throughout the world, there is an increasing focus on accessibility for all, whether in a particular country or setting or in areas of the world that may not yet have computing established. This is primarily articulated as an equity issue. The recently developed capacity for, access to, participation in, and experience of computer science education (CAPE) Framework is one way of demonstrating stages and dependencies and understanding relative equity, taking into consideration the disparities between sub-populations. While there is existing research that covers the state of computing education and equity issues, it is mostly in high-income countries; there is minimal research in the context of low-middle-income countries like the sub-Saharan African countries. Objectives. The objective of the article is therefore to report on a pilot study investigating the capacity (one of the equity issues), for delivering computing education in four sub-Saharan African countries:

Botswana, Kenya, Nigeria and Uganda, countries that are in different geographic regions as well as in different income brackets (low-middle income). Method. In addition to reviewing the capacity issues of curriculum and policy around computing education in each country, we surveyed 58 teachers about the infrastructure, resources, professional development, and curriculum for computing in their country. We used a localized version of the MEasuring TeacheR Enacted Computing Curriculum (METRECC) instrument for this purpose. Results. We analyzed the results through the lens of the CAPE framework at the capacity level. We identified similarities and differences in the data from teachers who completed the original METRECC survey, all of whom were from high-income countries and African teachers. The data revealed statistically significant differences between the two datasets in relation to access to resources and professional development opportunities in computer studies/computer science, with the African teachers experiencing more barriers. Results further showed that African teachers focus less on teaching algorithms and programming than teachers from high-income countries. In addition, we found differences between African countries in the study, reflecting their relative access to IT infrastructure and resources.

[**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, Nov 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

[**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 (OOPSLA)*. ACM, Oct 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.

[**Wang2020a**] 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, Jun 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.

[**Wang2021**] Jiawei Wang, Li Li, and Andreas Zeller. Restoring execution environments of Jupyter notebooks, 2021.
Abstract: More than ninety percent of published Jupyter notebooks do not state dependencies on external packages. This makes them non-executable and thus hinders reproducibility of scientific results. We present SnifferDog, an approach that 1) collects the APIs of Python packages and versions, creating a database of APIs; 2) analyzes notebooks to determine candidates for required packages and versions; and 3) checks which packages are required to make the notebook executable (and ideally, reproduce its stored results). In its evaluation, we show that SnifferDog precisely restores execution environments for the largest majority of notebooks, making them immediately executable for end users.

[**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, May 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, Mar 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.

[**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, Sep 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, Nov 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, May 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, Aug 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 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.

[**Yang2022**] Wenhua Yang, Chong Zhang, Minxue Pan, Chang Xu, Yu Zhou, and Zhiqiu Huang. Do developers really know how to use git commands? a large-scale study using stack overflow. *ACM Transactions on Software Engineering and Methodology*, 31(3):1–29, Jul 2022, DOI 10.1145/3494518.
Abstract: Git, a cross-platform and open-source distributed version control tool, provides strong support for non-linear development and is capable of handling everything from small to large projects with speed and efficiency. It has become an indispensable tool for millions of software developers and is the de facto standard of version control in software development nowadays. However, despite its widespread use, developers still frequently face difficulties when using various Git commands to manage projects and collaborate. To better help developers use Git, it is necessary to understand the issues and difficulties that they may encounter when using Git. Unfortunately, this problem has not yet been comprehensively studied. To fill this knowledge gap, in this paper, we conduct a large-scale study on Stack Overflow, a popular Q&A forum for developers. We extracted and analyzed 80,370 relevant questions from Stack Overflow, and reported the increasing popularity of the Git command questions. By analyzing the questions, we identified the Git commands that are frequently asked and those that are associated with difficult questions on Stack Overflow to help understand the difficulties developers may encounter when using Git commands. In addition, we conducted a survey to understand how developers learn Git commands in practice, showing that self-learning is the primary learning approach. These findings provide a range of actionable implications for researchers, educators, and developers.

[**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, Sep 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.

[**Yelam2021**] Anil Yelam, Shibani Subbareddy, Keerthana Ganesan, Stefan Savage, and Ariana Mirian. CoResident evil: Covert communication in the cloud with lambdas. In *Proceedings of the Web Conference 2021*. ACM, Apr 2021, DOI 10.1145/3442381.3450100.
Abstract: "Serverless" cloud services, such as AWS lambdas, are one of the fastest growing segments of the cloud services market. These services are popular in part due to their light-weight nature and flexibility in scheduling and cost, however the security issues associated with serverless computing are not well understood. In this work, we explore the feasibility of constructing a practical covert channel from lambdas. We establish that a fast co-residence detection for lambdas is key to enabling such a covert channel, and proceed to develop a reliable and scalable co-residence detector based on the memory bus hardware. Our technique enables dynamic discovery for co-resident lambdas and is incredibly fast, executing in a matter of seconds. We evaluate our approach for correctness and scalability, and use it to establish covert channels and perform data transfer on AWS lambdas. We show that we can establish hundreds of individual covert channels for every 1000 lambdas deployed, and each of those channels can send data at a rate of 200 bits per second, thus demonstrating that covert communication via lambdas is entirely feasible.

[**Yin2011**] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proc. International Symposium on Foundations of Software Engineering/International Symposium on the Foundations of Software Engineering (SIGSOFT/FSE)*. ACM Press, 2011, DOI 10.1145/2025113.2025121.
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.

[**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, Jan 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.

[**Zampetti2021**] Fiorella Zampetti, Gianmarco Fucci, Alexander Serebrenik, and Massimiliano Di Penta. Self-admitted technical debt practices: a comparison between industry and open-source. *Empirical Software Engineering*, 26(6), Sep 2021, DOI `10.1007/s10664-021-10031-3`.
Abstract: Self-admitted technical debt (SATD) consists of annotations, left by developers as comments in the source code or elsewhere, as a reminder about pieces of software manifesting technical debt (TD), i.e., "not being ready yet". While previous studies have investigated SATD management and its relationship with software quality, there is little understanding of the extent and circumstances to which developers admit TD. This paper reports the results of a study in which we asked developers from industry and opensource about their practices in annotating source code and other artifacts for self-admitting TD. The study consists of two phases. First, we conducted 10 interviews to gather a first understanding of the phenomenon and to prepare a survey questionnaire. Then, we surveyed 52 industrial developers as well as 49 contributors to open-source projects. Results of the study show how the TD annotation practices, as well as the typical content of SATD comments, are very similar between open-source and industry. At the same time, our results highlight how, while open-source code is spread of comments admitting the need for improvements, SATD in industry may be dictated by organizational guidelines but, at the same time, implicitly discouraged by the fear of admitting responsibilities. Results also highlight the need for tools helping developers to achieve a better TD awareness.

[**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, May 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.

[**Zeller2009**] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, second edition, 2009.
Abstract: Proof that debugging has graduated from a black art to a systematic discipline. It demystifies one of the toughest aspects of software programming, showing clearly how to discover what caused software failures, and fix them with minimal muss and fuss.

[**Zeller2021**] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2021.
Abstract: A live book explaining how to automate software testing by generating tests automatically.

[**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, Jun 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 overflow. *IEEE Transactions on Software Engineering*, 47(4):850–862, Apr 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.

[**Zhang2022a**] Haiyin Zhang, Luís Cruz, and Arie van Deursen. Code smells for machine learning applications. In *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI*. ACM, May 2022, DOI `10.1145/3522664.3528620`.

Abstract: The popularity of machine learning has wildly expanded in recent years. Machine learning techniques have been heatedly studied in academia and applied in the industry to create business value. However, there is a lack of guidelines for code quality in machine learning applications. In particular, code smells have rarely been studied in this domain. Although machine learning code is usually integrated as a small part of an overarching system, it usually plays an important role in its core functionality. Hence ensuring code quality is quintessential to avoid issues in the long run. This paper proposes and identifies a list of 22 machine learning-specific code smells collected from various sources, including papers, grey literature, GitHub commits, and Stack Overflow posts. We pinpoint each smell with a description of its context, potential issues in the long run, and proposed solutions. In addition, we link them to their respective pipeline stage and the evidence from both academic and grey literature. The code smell catalog helps data scientists and

developers produce and maintain high-quality machine learning application code.

[**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, May 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.