It Will Never Work in Theory: To Do

http://neverworkintheory.org

May 10, 2023

References

[**Abate2020**] Pietro Abate, Roberto Di Cosmo, Georgios Gousios, and Stefano Zacchiroli. Dependency solving is still hard, but we are getting better at it. 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), Feb 2020, London, Canada. pp.547-551, 2020, DOI 10.1109/SANER48275.2020.9054837.

Abstract: Dependency solving is a hard (NP-complete) problem in all non-trivial component models due to either mutually incompatible versions of the same packages or explicitly declared package conflicts. As such, software upgrade planning needs to rely on highly specialized dependency solvers, lest falling into pitfalls such as incompleteness—a combination of package versions that satisfy dependency constraints does exist, but the package manager is unable to find it. In this paper we look back at proposals from dependency solving research dating back a few years. Specifically, we review the idea of treating dependency solving as a separate concern in package manager implementations, relying on generic dependency solvers based on tried and tested techniques such as SAT solving, PBO, MILP, etc. By conducting a census of dependency solving capabilities in state-of-the-art package managers we conclude that some proposals are starting to take off (e.g., SAT-based dependency solving) while—with few exceptions—others have not (e.g., outsourcing dependency solving to reusable components). We reflect on why that has been the case and look at novel challenges for dependency solving that have emerged since.

[**AbouKhalil2022**] Zeinab Abou Khalil and Stefano Zacchiroli. The general index of software engineering papers. In *Proc. International Conference on Mining Software Repositories (MSR)*. ACM, May 2022, DOI 10.1145/3524842.3528494.

Abstract: We introduce the General Index of Software Engineering Papers, a dataset of fulltext-indexed papers from the most prominent scientific venues in the field of Software Engineering. The dataset includes both complete bibliographic information and indexed n-grams (sequence of contiguous words after removal of stopwords and non-words, for a total of 577 276 382 unique n-grams in this release) with length 1 to 5 for 44 581 papers retrieved

from 34 venues over the 1971–2020 period. The dataset serves use cases in the field of meta-research, allowing to introspect the output of software engineering research even when access to papers or scholarly search engines is not possible (e.g., due to contractual reasons). The dataset also contributes to making such analyses reproducible and independently verifiable, as opposed to what happens when they are conducted using 3rd-party and non-open scholarly indexing services. The dataset is available as a portable Postgres database dump and released as open data.

[**Abreu2022**] Rui Abreu. The bumpy road of taking automated debugging to industry, 2022.
Abstract: Debugging is arguably among the most difficult and extremely time consuming tasks of the software development life cycle. Therefore, it comes as no surprise that researchers have invested a considerable amount of effort in developing automated techniques and tools to support developers excel in these tasks. Despite the significant advances, including demonstrations of usefulness, efficacy, and efficiency, these techniques are yet to find their way into industrial adoption. In this paper, we reflect upon the commercialization efforts of a particular automated debugging technique and lay down potential reasons for lack of success stories as well as ideas to move forward.

[**Ahmed2022**] Umair Z. Ahmed, Zhiyu Fan, Jooyong Yi, Omar I. Al-Bataineh, and Abhik Roychoudhury. Verifix: Verified repair of programming assignments, 2022.
Abstract: Automated feedback generation for introductory programming assignments is useful for programming education. Most works try to generate feedback to correct a student program by comparing its behavior with an instructor's reference program on selected tests. In this work, our aim is to generate verifiably correct program repairs as student feedback. The student assignment is aligned and composed with a reference solution in terms of control flow, and differences in data variables are automatically summarized via predicates to relate the variable names. Failed verification attempts for the equivalence of the two programs are exploited to obtain a collection of maxSMT queries, whose solutions point to repairs of the student assignment. We have conducted experiments on student assignments curated from a widely deployed intelligent tutoring system. Our results indicate that we can generate verified feedback in up to 58% of the assignments. More importantly, our system indicates when it is able to generate a verified feedback, which is then usable by novice students with high confidence.

[**Ait2022**] Adem Ait, Javier Luis Cánovas Izquierdo, and Jordi Cabot. An empirical study on the survival rate of GitHub projects. In *Proc. International Conference on Mining Software Repositories (MSR)*. ACM, May 2022, DOI `10.1145/3524842.3527941`.
Abstract: The number of Open Source projects hosted in social coding platforms such as GitHub is constantly growing. However, many of these

projects are not regularly maintained and some are even abandoned shortly after they were created. In this paper we analyze early project development dynamics in software projects hosted on GitHub, including their survival rate. To this aim, we collected all 1,127 GitHub repositories from four different ecosystems (i.e., NPM packages, R packages, WordPress plugins and Laravel packages) created in 2016. We stored their activity in a time series database and analyzed their activity evolution along their lifespan, from 2016 to now. Our results reveal that the prototypical development process consists of intensive coding-driven active periods followed by long periods of inactivity. More importantly, we have found that a significant number of projects die in the first year of existence with the survival rate decreasing year after year. In fact, the probability of surviving longer than five years is less than 50% though some types of projects have better chances of survival.

[**AlOmar2022**] Eman Abdullah AlOmar, Moataz Chouchen, Mohamed Wiem Mkaouer, and Ali Ouni. Code review practices for refactoring changes: An empirical study on openstack, 2022.
Abstract: Modern code review is a widely used technique employed in both industrial and open-source projects to improve software quality, share knowledge, and ensure adherence to coding standards and guidelines. During code review, developers may discuss refactoring activities before merging code changes in the code base. To date, code review has been extensively studied to explore its general challenges, best practices and outcomes, and socio-technical aspects. However, little is known about how refactoring is being reviewed and what developers care about when they review refactored code. Hence, in this work, we present a quantitative and qualitative study to understand what are the main criteria developers rely on to develop a decision about accepting or rejecting a submitted refactored code, and what makes this process challenging. Through a case study of 11,010 refactoring and non-refactoring reviews spread across OpenStack open-source projects, we find that refactoring-related code reviews take significantly longer to be resolved in terms of code review efforts. Moreover, upon performing a thematic analysis on a significant sample of the refactoring code review discussions, we built a comprehensive taxonomy consisting of 28 refactoring review criteria. We envision our findings reaffirming the necessity of developing accurate and efficient tools and techniques that can assist developers in the review process in the presence of refactorings.

[**Amjad2023**] Abdul Haddi Amjad, Zubair Shafiq, and Muhammad Ali Gulzar. Blocking javascript without breaking the web: An empirical investigation, 2023.
Abstract: Modern websites heavily rely on JavaScript (JS) to implement legitimate functionality as well as privacy-invasive advertising and tracking. Browser extensions such as NoScript block any script not loaded by a trusted list of endpoints, thus hoping to block privacy-invasive scripts while avoiding breaking legitimate website functionality. In this paper, we investigate whether blocking JS on the web is feasible without breaking legitimate

functionality. To this end, we conduct a large-scale measurement study of JS blocking on 100K websites. We evaluate the effectiveness of different JS blocking strategies in tracking prevention and functionality breakage. Our evaluation relies on quantitative analysis of network requests, and resource loads as well as manual qualitative analysis of visual breakage. First, we show that while blocking all scripts is quite effective at reducing tracking, it significantly degrades functionality on approximately two-thirds of the tested websites. Second, we show that selective blocking of a subset of scripts based on a curated list achieves a better tradeoff. However, there remain approximately 15% "mixed" scripts, which essentially merge tracking and legitimate functionality and thus cannot be blocked without causing website breakage. Finally, we show that fine-grained blocking of a subset of JS methods, instead of scripts, reduces major breakage by 3.7× while providing the same level of tracking prevention. Our work highlights the promise and open challenges in fine-grained JS blocking for tracking prevention without breaking the web.

[**Andersen2020**] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. Adding interactive visual syntax to textual code. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, Nov 2020, DOI `10.1145/3428290`.

Abstract: Many programming problems call for turning geometrical thoughts into code: tables, hierarchical structures, nests of objects, trees, forests, graphs, and so on. Linear text does not do justice to such thoughts. But, it has been the dominant programming medium for the past and will remain so for the foreseeable future. This paper proposes a novel mechanism for conveniently extending textual programming languages with problem-specific visual syntax. It argues the necessity of this language feature, demonstrates the feasibility with a robust prototype, and sketches a design plan for adapting the idea to other languages.

[**Apple2022**] Jim Apple. Stretching your data with taffy filters, 2022.

Abstract: Popular approximate membership query structures such as Bloom filters and cuckoo filters are widely used in databases, security, and networking. These structures represent sets approximately, and support at least two operations - insert and lookup; lookup always returns true on elements inserted into the structure; it also returns true with some probability 0¡ϵ¡1 on elements not inserted into the structure. These latter elements are called false positives. Compensatory for these false positives, filters can be much smaller than hash tables that represent the same set. However, unlike hash tables, cuckoo filters and Bloom filters must be initialized with the intended number of inserts to be performed, and cannot grow larger - inserts beyond this number fail or significantly increase the false positive probability. This paper presents designs and implementations of filters than can grow without inserts failing and without meaningfully increasing the false positive probability, even if the filters are created with a small initial size.

The resulting code is available on GitHub under a permissive open source license.

[**Arora2023**] Chetan Arora, Laura Tubino, Andrew Cain, Kevin Lee, and Vasudha Malhotra. Persona-based assessment of software engineering student research projects: An experience report, 2023.
Abstract: Students enrolled in software engineering degrees are generally required to undertake a research project in their final year through which they demonstrate the ability to conduct research, communicate outcomes, and build in-depth expertise in an area. Assessment in these projects typically involves evaluating the product of their research via a thesis or a similar artifact. However, this misses a range of other factors that go into producing successful software engineers and researchers. Incorporating aspects such as process, attitudes, project complexity, and supervision support into the assessment can provide a more holistic evaluation of the performance likely to better align with the intended learning outcomes. In this paper, we present on our experience of adopting an innovative assessment approach to enhance learning outcomes and research performance in our software engineering research projects. Our approach adopted a task-oriented approach to portfolio assessment that incorporates student personas, frequent formative feedback, delayed summative grading, and standards-aligned outcomes-based assessment. We report upon our continuous improvement journey in adapting tasks and criteria to address the challenges of assessing student research projects. Our lessons learnt demonstrate the value of personas to guide the development of holistic rubrics, giving meaning to grades and focusing staff and student attention on attitudes and skills rather than a product only.

[**Arteca2022**] Ellen Arteca, Sebastian Harner, Michael Pradel, and Frank Tip. Nessie: automatically testing javascript apis with asynchronous callbacks. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, May 2022, DOI 10.1145/3510003.3510106.
Abstract: Previous algorithms for feedback-directed unit test generation iteratively create sequences of API calls by executing partial tests and by adding new API calls at the end of the test. These algorithms are challenged by a popular class of APIs: higher-order functions that receive callback arguments, which often are invoked asyn-chronously. Existing test generators cannot effectively test such APIs because they only sequence API calls, but do not nest one call into the callback function of another. This paper presents Nessie, the first feedback-directed unit test generator that supports nesting of API calls and that tests asynchronous callbacks. Nesting API calls enables a test to use values produced by an API that are available only once a callback has been invoked, and is often necessary to ensure that methods are invoked in a specific order. The core contributions of our approach are a tree-based representation of unit tests with callbacks and a novel algorithm to iteratively generate such tests in a feedback-directed manner. We evaluate our approach on ten popular JavaScript libraries with both asynchronous and synchronous callbacks. The results show that, in a comparison with LambdaTester, a

state of the art test generation technique that only considers sequencing of method calls, Nessie finds more behavioral differences and achieves slightly higher coverage. Notably, Nessie needs to generate significantly fewer tests to achieve and exceed the coverage achieved by the state of the art.

[**Asare2022**] Owura Asare, Meiyappan Nagappan, and N. Asokan. Is github's copilot as bad as humans at introducing vulnerabilities in code?, 2022.
Abstract: Several advances in deep learning have been successfully applied to the software development process. Of recent interest is the use of neural language models to build tools, such as Copilot, that assist in writing code. In this paper we perform a comparative empirical analysis of Copilot-generated code from a security perspective. The aim of this study is to determine if Copilot is as bad as human developers - we investigate whether Copilot is just as likely to introduce the same software vulnerabilities that human developers did. Using a dataset of C/C++ vulnerabilities, we prompt Copilot to generate suggestions in scenarios that previously led to the introduction of vulnerabilities by human developers. The suggestions are inspected and categorized in a 2-stage process based on whether the original vulnerability or the fix is reintroduced. We find that Copilot replicates the original vulnerable code $\tilde{3}3\%$ of the time while replicating the fixed code at a $\tilde{2}5\%$ rate. However this behavior is not consistent: Copilot is more susceptible to introducing some types of vulnerability than others and is more likely to generate vulnerable code in response to prompts that correspond to older vulnerabilities than newer ones. Overall, given that in a substantial proportion of instances Copilot did not generate code with the same vulnerabilities that human developers had introduced previously, we conclude that Copilot is not as bad as human developers at introducing vulnerabilities in code.

[**Barbosa2022**] Leonardo Barbosa, Victor Hugo Santiago, Alberto Luiz Oliveira Tavares de Souza, and Gustavo Pinto. To what extent cognitive-driven development improves code readability?, 2022.
Abstract: Cognitive-Driven Development (CDD) is a coding design technique that aims to reduce the cognitive effort that developers place in understanding a given code unit (e.g., a class). By following CDD design practices, it is expected that the coding units to be smaller, and, thus, easier to maintain and evolve. However, it is so far unknown whether these smaller code units coded using CDD standards are, indeed, easier to understand. In this work we aim to assess to what CDD improves code readability. To achieve this goal, we conducted a two-phase study. We start by inviting professional software developers to vote (and justify their rationale) on the most readable pair of code snippets (from a set of 10 pairs); one of the pairs was coded using CDD practices. We received 133 answers. In the second phase, we applied the state-of-the art readability model on the 10-pairs of CDD-guided refactorings. We observed some conflicting results. On the one hand, developers perceived that seven (out of 10) CDD-guided refactorings were more readable than their counterparts; for two other CDD-guided refactorings, developers were undecided, while only in one of the CDD-guided refactorings,

developers preferred the original code snippet. On the other hand, we noticed that only one CDD-guided refactorings have better performance readability, assessed by state-of-the-art readability models. Our results provide initial evidence that CDD could be an interesting approach for software design.

[**Barrak2021**] Amine Barrak, Ellis E. Eghan, and Bram Adams. On the co-evolution of ML pipelines and source code - empirical study of DVC projects. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Mar 2021, DOI 10.1109/saner50967.2021.00046.

Abstract: The growing popularity of machine learning (ML) applications has led to the introduction of software engineering tools such as Data Versioning Control (DVC), MLFlow and Pachyderm that enable versioning ML data, models, pipelines and model evaluation metrics. Since these versioned ML artifacts need to be synchronized not only with each other, but also with the source and test code of the software applications into which the models are integrated, prior findings on co-evolution and coupling between software artifacts might need to be revisited. Hence, in order to understand the degree of coupling between ML-related and other software artifacts, as well as the adoption of ML versioning features, this paper empirically studies the usage of DVC in 391 Github projects, 25 of which in detail. Our results show that more than half of the DVC files in a project are changed at least once every one-tenth of the project's lifetime. Furthermore, we observe a tight coupling between DVC files and other artifacts, with 1/4 pull requests changing source code and 1/2 pull requests changing tests requiring a change to DVC files. As additional evidence of the observed complexity associated with adopting ML-related software engineering tools like DVC, an average of 78% of the studied projects showed a non-constant trend in pipeline complexity.

[**Beasley2022**] Zachariah J. Beasley and Ayesha R. Johnson. The impact of remote pair programming in an upper-level CS course. In *Proc. Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, Jul 2022, DOI 10.1145/3502718.3524772.

Abstract: Pair programming is an active learning technique with several benefits to students, including increasing participation and improving outcomes, particularly for female computer science students. However, most of the literature highlights the effects of pair programming in introductory courses, where students have different prior programming experience and thus may experience group issues. This work analyzes the effect of pair programming in an upper-level computer science course, where students have a more consistent background education, particularly in languages learned and coding best practices. Secondly, the effect of remote pair programming on student outcomes is still an open question of increasing importance with the advent of Covid-19. This work utilized split sections with a control and treatment group in a large, public university. In addition to comparing pair programming to individual programming, results were analyzed by modality (remote vs. in person) and by gender, focusing on how pair programming

benefits female computer science students in confidence, persistence in the major, and outcomes. We found that pair programming groups scored higher on assignments and exams, that remote pair programming groups performed as well as in person groups, and that female students increased their confidence in asking questions in class and scored 12% higher in the course when utilizing pair programming.

[**Becker2022**] Brett A. Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. Programming is hard – or at least it used to be: Educational opportunities and challenges of ai code generation, 2022.
Abstract: The introductory programming sequence has been the focus of much research in computing education. The recent advent of several viable and freely-available AI-driven code generation tools present several immediate opportunities and challenges in this domain. In this position paper we argue that the community needs to act quickly in deciding what possible opportunities can and should be leveraged and how, while also working on how to overcome or otherwise mitigate the possible challenges. Assuming that the effectiveness and proliferation of these tools will continue to progress rapidly, without quick, deliberate, and concerted efforts, educators will lose advantage in helping shape what opportunities come to be, and what challenges will endure. With this paper we aim to seed this discussion within the computing education community.

[**Bendrissou2022**] Bachir Bendrissou, Rahul Gopinath, and Andreas Zeller. "synthesizing input grammars": a replication study. In *Proc. Conference on Programming Language Design and Implementation (PLDI)*. ACM, Jun 2022, DOI 10.1145/3519939.3523716.
Abstract: When producing test inputs for a program, test generators ("fuzzers") can greatly profit from grammars that formally describe the language of expected inputs. In recent years, researchers thus have studied means to recover input grammars from programs and their executions. The GLADE algorithm by Bastani et al., published at PLDI 2017, was the first black-box approach to claim context-free approximation of input specification for non-trivial languages such as XML, Lisp, URLs, and more. Prompted by recent observations that the GLADE algorithm may show lower performance than reported in the original paper, we have reimplemented the GLADE algorithm from scratch. Our evaluation confirms that the effectiveness score (F1) reported in the GLADE paper is overly optimistic, and in some cases, based on the wrong language. Furthermore, GLADE fares poorly in several real-world languages evaluated, producing grammars that spend megabytes to enumerate inputs.

[**Bi2021**] Tingting Bi, Wei Ding, Peng Liang, and Antony Tang. Architecture information communication in two OSS projects: The why, who, when, and what. *Journal of Systems and Software*, 181:111035, Nov 2021, DOI 10.1016/j.jss.2021.111035.

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

[**Bijlsma2022**] Lex A. Bijlsma, Arjan J. F. Kok, Harrie J. M. Passier, Harold J. Pootjes, and Sylvia Stuurman. Evaluation of design pattern alternatives in java. *Softw. Pract. Exp.*, 52(5):1305–1315, May 2022, DOI `10.1002/spe.3061`.

Abstract: Design patterns are standard solutions to common design problems. The famous Gang of Four book describes more than twenty design patterns for the object-oriented paradigm. These patterns were developed more than twenty-five years ago, using the programming language concepts available at that time. Patterns do not always fit underlying domain concepts. For example, even when a concrete strategy is a pure function, the classical strategy pattern represents this as a separate subclass and as such obscures the intent of this pattern with extra complexities due to the inheritance-based implementation. Due to the ongoing development of oo-languages, a relevant question is whether the implementation of these patterns can be improved using new language features, such that they fit more closely with the intent. An additional question is then how we can decide which implementation is to be preferred. In this article, we investigate both questions, using the strategy pattern as an example. Our main contribution is that we show how to reason about different implementations, using both the description of a design pattern and design principles as guidance.

[**Biswas2022**] Sumon Biswas, Mohammad Wardat, and Hridesh Rajan. The art and practice of data science pipelines: A comprehensive study of data science pipelines in theory, in-the-small, and in-the-large. ICSE 2022: The 44th International Conference on Software Engineering, 2022, DOI `10.1145/3510003.3510057`.

Abstract: Increasingly larger number of software systems today are including data science components for descriptive, predictive, and prescriptive

analytics. The collection of data science stages from acquisition, to cleaning/curation, to modeling, and so on are referred to as data science pipelines. To facilitate research and practice on data science pipelines, it is essential to understand their nature. What are the typical stages of a data science pipeline? How are they connected? Do the pipelines differ in the theoretical representations and that in the practice? Today we do not fully understand these architectural characteristics of data science pipelines. In this work, we present a three-pronged comprehensive study to answer this for the state-of-the-art, data science in-the-small, and data science in-the-large. Our study analyzes three datasets: a collection of 71 proposals for data science pipelines and related concepts in theory, a collection of over 105 implementations of curated data science pipelines from Kaggle competitions to understand data science in-the-small, and a collection of 21 mature data science projects from GitHub to understand data science in-the-large. Our study has led to three representations of data science pipelines that capture the essence of our subjects in theory, in-the-small, and in-the-large.

[**Bittner2022**] Paul Maximilian Bittner, Christof Tinnes, Alexander Schultheiß, Sören Viegener, Timo Kehrer, and Thomas Thüm. Classifying edits to variability in source code. In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, Nov 2022, DOI 10.1145/3540250.3549108.
Abstract: For highly configurable software systems, such as the Linux kernel, maintaining and evolving variability information along changes to source code poses a major challenge. While source code itself may be edited, also feature-to-code mappings may be introduced, removed, or changed. In practice, such edits are often conducted ad-hoc and without proper documentation. To support the maintenance and evolution of variability, it is desirable to understand the impact of each edit on the variability. We propose the first complete and unambiguous classification of edits to variability in source code by means of a catalog of edit classes. This catalog is based on a scheme that can be used to build classifications that are complete and unambiguous by construction. To this end, we introduce a complete and sound model for edits to variability. In about 21.5ms per commit, we validate the correctness and suitability of our classification by classifying each edit in 1.7 million commits in the change histories of 44 open-source software systems automatically. We are able to classify all edits with syntactically correct feature-to-code mappings and find that all our edit classes occur in practice.

[**Blacher2022**] Mark Blacher, Joachim Giesen, Peter Sanders, and Jan Wassenberg. Vectorized and performance-portable quicksort, 2022.
Abstract: Recent works showed that implementations of Quicksort using vector CPU instructions can outperform the non-vectorized algorithms in widespread use. However, these implementations are typically single-threaded, implemented for a particular instruction set, and restricted to a small set of key types. We lift these three restrictions: our proposed 'vqsort'

algorithm integrates into the state-of-the-art parallel sorter 'ips4o', with a geometric mean speedup of 1.59. The same implementation works on seven instruction sets (including SVE and RISC-V V) across four platforms. It also supports floating-point and 16-128 bit integer keys. To the best of our knowledge, this is the fastest sort for non-tuple keys on CPUs, up to 20 times as fast as the sorting algorithms implemented in standard libraries. This paper focuses on the practical engineering aspects enabling the speed and portability, which we have not yet seen demonstrated for a Quicksort implementation. Furthermore, we introduce compact and transpose-free sorting networks for in-register sorting of small arrays, and a vector-friendly pivot sampling strategy that is robust against adversarial input.

[**Blackwell2019**] Alan F. Blackwell, Marian Petre, and Luke Church. Fifty years of the psychology of programming. *International Journal of Human-Computer Studies*, 131:52–63, Nov 2019, DOI 10.1016/j.ijhcs.2019.06.009.

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

[**Boag2022**] William Boag, Harini Suresh, Bianca Lepe, and Catherine D'Ignazio. Tech worker organizing for power and accountability. In *2022 ACM Conference on Fairness, Accountability, and Transparency*. ACM, Jun 2022, DOI 10.1145/3531146.3533111.

Abstract: In recent years, there has been a growing interest in the field of "AI Ethics" and related areas. This field is purposefully broad, allowing for the intersection of numerous subfields and disciplines. However, a lot of work in this area thus far has centered computational methods, leading to a narrow lens where technical tools are framed as solutions for broader sociotechnical problems. In this work, we discuss a less-explored mode of what it can mean to "do" AI Ethics: tech worker collective action. Through collective action, the employees of powerful tech companies can act as a countervailing force against strong corporate impulses to grow or make a profit to the detriment of other values. In this work, we ground these efforts

in existing scholarship of social movements and labor organizing. We characterize 150 documented collective actions, and explore several case studies of successful campaigns. Looking forward, we also identify under-explored types of actions, and provide conceptual frameworks and inspiration for how to utilize worker organizing as an effective lever for change.

[**Bogner2022**] Justus Bogner and Manuel Merkel. To type or not to type? a systematic comparison of the software quality of javascript and typescript applications on github, 2022.
Abstract: JavaScript (JS) is one of the most popular programming languages, and widely used for web apps and even backend development. Due to its dynamic nature, however, JS applications often have a reputation for poor software quality. As a type-safe superset of JavaScript, TypeScript (TS) offers features to address this. However, there is currently insufficient empirical evidence to broadly support the claim that TS apps exhibit better software quality than JS apps. We therefore conducted a repository mining study based on 604 GitHub projects (299 for JS, 305 for TS) with over 16M LoC and collected four facets of software quality: a) code quality (# of code smells per LoC), b) code understandability (cognitive complexity per LoC), c) bug proneness (bug fix commit ratio), and d) bug resolution time (mean time a bug issue is open). For TS, we also collected how frequently the type-safety ignoring 'any' type was used. The analysis indicates that TS apps exhibit significantly better code quality and understandability than JS apps. Contrary to expectations, however, bug proneness and bug resolution time of our TS sample were not significantly lower than for JS: mean bug fix commit ratio was more than 60% larger (0.126 vs. 0.206), and TS projects needed on average more than an additional day to fix bugs (31.86 vs. 33.04 days). Furthermore, reducing the usage of the 'any' type in TS apps was significantly correlated with all metrics except bug proneness (Spearman's rho between 0.17 and 0.26). Our results indicate that the perceived positive influence of TypeScript for avoiding bugs in comparison to JavaScript may be more complicated than assumed. While using TS seems to have benefits, it does not automatically lead to less and easier to fix bugs. However, more research is needed in this area, especially concerning the potential influence of project complexity and developer experience.

[**Borg2022**] Markus Borg, Leif Jonsson, Emelie Engström, Béla Bartalos, and Attila Szabó. Adopting automated bug assignment in practice: A longitudinal case study at ericsson, 2022.
Abstract: The continuous inflow of bug reports is a considerable challenge in large development projects. Inspired by contemporary work on mining software repositories, we designed a prototype bug assignment solution based on machine learning in 2011-2016. The prototype evolved into an internal Ericsson product, TRR, in 2017-2018. TRR's first bug assignment without human intervention happened in April 2019. Our study evaluates the adoption of TRR within its industrial context at Ericsson. Moreover,

we investigate 1) how TRR performs in the field, 2) what value TRR provides to Ericsson, and 3) how TRR has influenced the ways of working. We conduct an industrial case study combining interviews with TRR stakeholders, minutes from sprint planning meetings, and bug tracking data. The data analysis includes thematic analysis, descriptive statistics, and Bayesian causal analysis. TRR is now an incorporated part of the bug assignment process. Considering the abstraction levels of the telecommunications stack, high-level modules are more positive while low-level modules experienced some drawbacks. On average, TRR automatically assigns 30% of the incoming bug reports with an accuracy of 75%. Auto-routed TRs are resolved around 21% faster within Ericsson, and TRR has saved highly seasoned engineers many hours of work. Indirect effects of adopting TRR include process improvements, process awareness, increased communication, and higher job satisfaction. TRR has saved time at Ericsson, but the adoption of automated bug assignment was more intricate compared to similar endeavors reported from other companies. We primarily attribute the difference to the very large size of the organization and the complex products. Key facilitators in the successful adoption include a gradual introduction, product champions, and careful stakeholder analysis.

[**BoumaSims2023**] Elijah Bouma-Sims and Yasemin Acar. Beyond the boolean: How programmers ask about, use, and discuss gender, 2023, DOI `10.1145/3579461`.

Abstract: Categorization via gender is omnipresent throughout society, and thus also computing; gender identity is often requested of users before they use software or web services. Despite this fact, no research has explored how software developers approach requesting gender disclosure from users. To understand how developers think about gender in software, we present an interview study with 15 software developers recruited from the freelancing platform Upwork as well as Twitter. We also collected and categorized 917 threads that contained keywords relevant to gender from programming-related sub-forums on the social media service Reddit. 16 posts that discussed approaches to gender disclosure were further analyzed. We found that while some developers have an understanding of inclusive gender options, programmers rarely consider when gender data is necessary or the way in which they request gender disclosure from users. Our findings have implications for programmers, software engineering educators, and the broader community concerned with inclusivity.

[**Brodley2022**] Carla E. Brodley, Benjamin J. Hescott, Jessica Biron, Ali Ressing, Melissa Peiken, Sarah Maravetz, and Alan Mislove. Broadening participation in computing via ubiquitous combined majors (CS+X). In *Proc. Technical Symposium on Computer Science Education (SIGCSE)*. ACM, Feb 2022, DOI `10.1145/3478431.3499352`.

Abstract: In 2001, Khoury College of Computer Sciences at Northeastern University created their first combined majors with Cognitive Psychology, Mathematics and Physics. This type of degree has often been referred to as

C̈S+Xïn the literature and is increasingly relevant as the need for interdisciplinary computer scientists grows. As of 2021, students at Northeastern can choose among three computing majors (Computer Science, Data Science or Cybersecurity) and 42 combined majors, which combine one of the three computing degrees with one of 29 distinct majors in other fields. Prior to 2014, combined majors were with the sciences, business and design. Over the last seven years, we created 29 new combined majors, explicitly creating combinations with fields where there has traditionally been greater gender diversity. The resulting increase in student interest and gender diversity over the last seven years is compelling. As of Fall 2020, 44.6% of the 2,800+ computing majors at Northeastern are pursuing combined majors, 39% of whom are women. This is substantially higher than the 21.5% reported in IPEDS for 2019 women computing graduates in the U.S. We did not observe any significant differences in racial and ethnic diversity between combined and computing only degrees. In this experience paper, we describe how we create and manage combined majors, and we present results on enrollments, admissions, graduation, internship placements, and how students discover combined majors.

[**Brun2022**] Yuriy Brun, Tian Lin, Jessie Elise Somerville, Elisha M. Myers, and Natalie C. Ebner. Blindspots in python and java APIs result in vulnerable code. *ACM Transactions on Software Engineering and Methodology*, Nov 2022, DOI `10.1145/3571850`.

Abstract: Blindspots in APIs can cause software engineers to introduce vulnerabilities, but such blindspots are, unfortunately, common. We study the efect APIs with blindspots have on developers in two languages by replicating a 109-developer, 24-Java-API controlled experiment. Our replication applies to Python and involves 129 new developers and 22 new APIs. We ind that using APIs with blindspots statistically signiicantly reduces the developers' ability to correctly reason about the APIs in both languages, but that the efect is more pronounced for Python. Interestingly, for Java, the efect increased with complexity of the code relying on the API, whereas for Python, the opposite was true. This suggests that Python developers are less likely to notice potential for vulnerabilities in complex code than in simple code, whereas Java developers are more likely to recognize the extra complexity and apply more care, but are more careless with simple code. Whether the developers considered API uses to be more diicult, less clear, and less familiar did not have an efect on their ability to correctly reason about them. Developers with better long-term memory recall were more likely to correctly reason about APIs with blindspots, but short-term memory, processing speed, episodic memory, and memory span had no efect. Surprisingly, professional experience and expertise did not improve the developers' ability to reason about APIs with blindspots across both languages, with long-term professionals with many years of experience making mistakes as often as relative novices. Finally, personality traits did not signiicantly afect the Python developers' ability to reason about APIs with blindspots, but less extraverted

and more open developers were better at reasoning about Java APIs with blindspots. Overall, our indings suggest that blindspots in APIs are a serious problem across languages, and that experience and education alone do not overcome that problem, suggesting that tools are needed to help developers recognize blindspots in APIs as they write code that uses those APIs.

[**Buffardi2020**] Kevin Buffardi. Assessing individual contributions to software engineering projects with Git logs and user stories. In *Proc. Technical Symposium on Computer Science Education (SIGCSE)*. ACM, Feb 2020, DOI 10.1145/3328778.3366948.

Abstract: Software Engineering courses often incorporate large-scale projects with collaboration between students working in teams. However, it is difficult to objectively assess individual students when their projects are a product of collaborative efforts. This study explores measurements of individuals' contributions to their respective teams. I analyzed ten Software Engineering team projects (n=42) and evaluations of individual contributions using automated evaluation of the version control system history (Git logs) and user stories completed on their project management (Kanban) boards. Unique insights from meta-data within the Git history and Kanban board user stories reveal complicated relationships between these measurements and traditional assessments, such as peer review and subjective instructor evaluation. From the results, I suggest supplementing and validating traditional assessments with insights from individuals' commit history and user story contributions.

[**CanovasIzquierdo2022**] Javier Luis Cánovas Izquierdo and Jordi Cabot. On the analysis of non-coding roles in open source development. *Empir. Softw. Eng.*, 27(1), Jan 2022, DOI 10.1007/s10664-021-10061-x.

Abstract: The role of non-coding contributors in Open Source Software (OSS) is poorly understood. Most of current research around OSS development focuses on the coding aspects of the project (e.g., commits, pull requests or code reviews) while ignoring the potential of other types of contributions. Often, due to the assumption that these other contributions are not significant in number and that, in any case, they are handled by the same people that are also part of the "coding team". This paper aims to investigate whether this is actually the case by analyzing the frequency and diversity of non-coding contributions in OSS development. As a sample of projects for our study we have taken the 100 most popular projects in the ecosystem of NPM, a package manager for JavaScript. Our results validate the importance of dedicated non-coding contributors in OSS and the diversity of OSS communities as, typically, a contributor specializes in a specific subset of roles. We foresee that projects adopting explicit policies to attract and onboard them could see a positive impact in their long-term sustainability providing they also put in place the right governance strategies to facilitate the migration and collaboration among the different roles. As part of this work, we also provide a replicability package to facilitate further quantitative role-based analysis by other researchers.

[**Chaparro2017**] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, Aug 2017, DOI 10.1145/3106237.3106285.
Abstract: Bug reports document unexpected software behaviors experienced by users. To be effective, they should allow bug triagers to easily understand and reproduce the potential reported bugs, by clearly describing the Observed Behavior (OB), the Steps to Reproduce (S2R), and the Expected Behavior (EB). Unfortunately, while considered extremely useful, reporters often miss such pieces of information in bug reports and, to date, there is no effective way to automatically check and enforce their presence. We manually analyzed nearly 3k bug reports to understand to what extent OB, EB, and S2R are reported in bug reports and what discourse patterns reporters use to describe such information. We found that (i) while most reports contain OB (i.e., 93.5%), only 35.2% and 51.4% explicitly describe EB and S2R, respectively; and (ii) reporters recurrently use 154 discourse patterns to describe such content. Based on these findings, we designed and evaluated an automated approach to detect the absence (or presence) of EB and S2R in bug descriptions. With its best setting, our approach is able to detect missing EB (S2R) with 85.9% (69.2%) average precision and 93.2% (83%) average recall. Our approach intends to improve bug descriptions quality by alerting reporters about missing EB and S2R at reporting time.

[**Chowdhury2022a**] Md Atique Reza Chowdhury, Rabe Abdalkareem, Emad Shihab, and Bram Adams. On the untriviality of trivial packages: An empirical study of npm JavaScript packages. *IEEE Transactions on Software Engineering*, 48(8):2695–2708, Aug 2022, DOI 10.1109/tse.2021.3068901.
Abstract: Nowadays, developing software would be unthinkable without the use of third-party packages. Although such code reuse helps to achieve rapid continuous delivery of software to end-users, blindly reusing code has its pitfalls. For example, prior work has investigated the rationale for using packages that implement simple functionalities, known as trivial packages (i.e., in terms of the code size and complexity). This prior work showed that although these trivial packages were simple, they were popular and prevalent in the npm ecosystem. This popularity and prevalence of trivial packages peaked our interest in questioning the 'triviality of trivial packages'. To better understand and examine the triviality of trivial packages, we mine a large set of JavaScript projects that use trivial npm packages and evaluate their relative centrality. Specifically, we evaluate the triviality from two complementary points of view: based on project usage and ecosystem usage of these trivial packages. Our result shows that trivial packages are being used in central JavaScript files of a software project. Additionally, by analyzing all external package API calls in these JavaScript files, we found that a high percentage of these API calls are attributed to trivial packages.

Therefore, these packages play a significant role in JavaScript files. Furthermore, in the package dependency network, we observed that 16.8 percent packages are trivial and in some cases removing a trivial package can impact approximately 29 percent of the ecosystem. Overall, our finding indicates that although smaller in size and complexity, trivial packages are highly depended on packages by JavaScript projects. Additionally, our study shows that although they might be called trivial, nothing about trivial packages is trivial.

[**Chowdhury2022b**] Partha Das Chowdhury, Mohammad Tahaei, and Awais Rashid. Better call saltzer & schroeder: A retrospective security analysis of solarwinds & log4j, 2022.
Abstract: Saltzer & Schroeder's principles aim to bring security to the design of computer systems. We investigate SolarWinds Orion update and Log4j to unpack the intersections where observance of these principles could have mitigated the embedded vulnerabilities. The common principles that were not observed include *fail safe defaults*, *economy of mechanism*, *complete mediation* and *least privilege*. Then we explore the literature on secure software development interventions for developers to identify usable analysis tools and frameworks that can contribute towards improved observance of these principles. We focus on a system wide view of access of codes, checking access paths and aiding application developers with safe libraries along with an appropriate security task list for functionalities.

[**Coleman2022**] Cora Coleman, William G. Griswold, and Nick Mitchell. Do cloud developers prefer clis or web consoles? clis mostly, though it varies by task, 2022.
Abstract: Despite the increased importance of Cloud tooling, and many large-scale studies of Cloud users, research has yet to answer what tool modalities (e.g. CLI or web console) developers prefer. In formulating our studies, we quickly found that preference varies heavily based on the programming task at hand. To address this gap, we conducted a two-part research study that quantifies modality preference as a function of programming task. Part one surveys how preference for three tool modalities (CLI, IDE, web console) varies across three classes of task (CRUD, debugging, monitoring). The survey shows, among 60 respondents, developers most prefer the CLI modality, especially for CRUD tasks. Monitoring tasks are the exception for which developers prefer the web console. Part two observes how four participants complete a task using the kubectl CLI and the OpenShift web console. All four participants prefer using the CLI to accomplish the task.

[**Colicev2022**] Anatoli Colicev, Tuuli Hakkarainen, and Torben Pedersen. Multi-project work and project performance: Friends or foes? *Strategic Management Journal*, Aug 2022, DOI `10.1002/smj.3443`.

[**Collaris2022**] Dennis Collaris, Hilde J. P. Weerts, Daphne Miedema, Jarke J. van Wijk, and Mykola Pechenizkiy. Characterizing data scientists' mental

models of local feature importance. In *Nordic Human-Computer Interaction Conference*. ACM, Oct 2022, DOI 10.1145/3546155.3546670.
Abstract: Feature importance is an approach that helps to explain machine learning model predictions. It works through assigning importance scores to input features of a particular model. Different techniques exist to derive these scores, with widely varying underlying assumptions of what importance means. Little research has been done to verify whether these assumptions match the expectations of the target user, which is imperative to ensure that feature importance values are not misinterpreted. In this work, we explore data scientists' mental models of (local) feature importance and compare these with the conceptual models of the techniques. We first identify several properties of local feature importance techniques that could potentially lead to misinterpretations. Subsequently, we explore the expectations data scientists have about local feature importance through an exploratory (qualitative and quantitative) survey of 34 data scientists in industry. We compare the identified expectations to the theory and assumptions behind the techniques and find that the two are not (always) in agreement.

[**Cosden2022**] Ian A. Cosden. An rse group model: Operational and organizational approaches from princeton university's central research software engineering group, 2022.
Abstract: The Princeton Research Software Engineering Group has grown rapidly since its inception in late 2016. The group, housed in the central Research Computing Department, comprised of professional Research Software Engineers (RSEs), works directly with researchers to create high quality research software to enable new scientific advances. As the group has matured so has the need for formalizing operational details and procedures. The RSE group uses an RSE partnership model, where Research Software Engineers work long-term with a designated academic department, institute, center, consortium, or individual principal investigator (PI). This article describes the operation of the central Princeton RSE group including funding, partner & project selection, and best practices for defining expectations for a successful partnership with researchers.

[**DalSasso2016**] Tommaso Dal Sasso, Andrea Mocci, and Michele Lanza. What makes a satisficing bug report? In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, Aug 2016, DOI 10.1109/QRS.2016.28.
Abstract: To ensure quality of software systems, developers use bug reports to track defects. It is in the interest of users and developers that bug reports provide the necessary information to ease the fixing process. Past research found that users do not provide the information that developers deem ideally useful to fix a bug. This raises an interesting question: What is the satisficing information to speed up the bug fixing process? We conducted an observational study on the relation between provided report information and its lifetime, considering more than 650,000 reports from open-source

systems using popular bug trackers. We distilled a meta-model for a minimal bug report, establishing a basic layer of core features. We found that few fields influence the resolution time and that customized fields have little impact on it. We performed a survey to investigate what users deem easy to provide in a bug report.

[**DeAlmeida2022**] Eduardo Santana de Almeida, Iftekhar Ahmed, and Andre van der Hoek. Let's go to the whiteboard (again):perceptions from software architects on whiteboard architecture meetings, 2022.
Abstract: The whiteboard plays a crucial role in the day-to-day lives of software architects, as they frequently will organize meetings at the whiteboard to discuss a new architecture, some proposed changes to the architecture, a mismatch between the architecture and the code, and more. While much has been studied about software architects, the architectures they produce, and how they produce them, a detailed understanding of these whiteboards meetings is still lacking. In this paper, we contribute a mixed-methods study involving semi-structured interviews and a subsequent survey to understand the perceptions of software architects on whiteboard architecture meetings. We focus on five aspects: (1) why do they hold these meetings, what is the impact of the experience levels of the participants in these meetings, how do the architects document the meetings, what kinds of changes are made after the meetings have concluded and their results are moved to implementation, and what role do digital whiteboards plays? In studying these aspects, we identify 12 observations related to both technical aspects and social aspects of the meetings. These insights have implications for further research, offer concrete advice to practitioners, provide guidance for future tool design, and suggest ways of educating future software architects.

[**DeSantana2022**] Taijara Loiola de Santana, Paulo Anselmo da Mota Silveira Neto, Eduardo Santana de Almeida, and Iftekhar Ahmed. Bug analysis in jupyter notebook projects: An empirical study, 2022.
Abstract: Computational notebooks, such as Jupyter, have been widely adopted by data scientists to write code for analyzing and visualizing data. Despite their growing adoption and popularity, there has been no thorough study to understand Jupyter development challenges from the practitioners' point of view. This paper presents a systematic study of bugs and challenges that Jupyter practitioners face through a large-scale empirical investigation. We mined 14,740 commits from 105 GitHub open-source projects with Jupyter notebook code. Next, we analyzed 30,416 Stack Overflow posts which gave us insights into bugs that practitioners face when developing Jupyter notebook projects. Finally, we conducted nineteen interviews with data scientists to uncover more details about Jupyter bugs and to gain insights into Jupyter developers' challenges. We propose a bug taxonomy for Jupyter projects based on our results. We also highlight bug categories, their root causes, and the challenges that Jupyter practitioners face.

[**DeSouzaSantos2022**] Ronnie E. de Souza Santos and Paul Ralph. A grounded theory of coordination in remote-first and hybrid software teams, 2022.

Abstract: While the long-term effects of the COVID-19 pandemic on software professionals and organizations are difficult to predict, it seems likely that working from home, remote-first teams, distributed teams, and hybrid (part-remote/part-office) teams will be more common. It is therefore important to investigate the challenges that software teams and organizations face with new remote and hybrid work. Consequently, this paper reports a year-long, participant-observation, constructivist grounded theory study investigating the impact of working from home on software development. This study resulted in a theory of software team coordination. Briefly, shifting from in-office to at-home work fundamentally altered coordination within software teams. While group cohesion and more effective communication appear protective, coordination is undermined by distrust, parenting and communication bricolage. Poor coordination leads to numerous problems including misunderstandings, help requests, lower job satisfaction among team members, and more ill-defined tasks. These problems, in turn, reduce overall project success and prompt professionals to alter their software development processes (in this case, from Scrum to Kanban). Our findings suggest that software organizations with many remote employees can improve performance by encouraging greater engagement within teams and supporting employees with family and childcare responsibilities.

[**Demirag2022**] Didem Demirag and Jeremy Clark. Opening sentences in academic writing. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education*. ACM, Feb 2022, DOI `10.1145/3478431.3499378`.

Abstract: Traditionally, education in computer science focuses on stakeholders like teachers, undergraduate students, and employers. However researchers also educate themselves about recent results and new subject matters. An important vehicle in this informal, self-education process is reading peer-reviewed academic papers—papers that are also used in the curriculum of graduate-level research courses. Technical writing skills are important in this domain, as well as engaging the reader with interesting text. This paper is a study of academic writing. We study in depth the first sentence used by researchers in opening their academic papers and how this sentence operates to draw the reader in. We use a corpus of 379 papers from a top-tier cybersecurity conference and use qualitative analysis (coding from grounded theory) to create a taxonomy of 5 general types and 14 sub-types of opening sentences. In this paper, we define and illustrate each type through examples, and reflect on what we learned about writing after examining all of these sentences.

[**DiGrazia2022**] Luca Di Grazia and Michael Pradel. The evolution of type annotations in Python: an empirical study. In *Proc. European*

Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE). ACM, Nov 2022, DOI 10.1145/3540250.3549114.
Abstract: Type annotations and gradual type checkers attempt to reveal errors and facilitate maintenance in dynamically typed programming languages. Despite the availability of these features and tools, it is currently unclear how quickly developers are adopting them, what strategies they follow when doing so, and whether adding type annotations reveals more type errors. This paper presents the first large-scale empirical study of the evolution of type annotations and type errors in Python. The study is based on an analysis of 1,414,936 type annotation changes, which we extract from 1,123,393 commits among 9,655 projects. Our results show that (i) type annotations are getting more popular, and once added, often remain unchanged in the projects for a long time, (ii) projects follow three evolution patterns for type annotation usage – regular annotation, type sprints, and occasional uses – and that the used pattern correlates with the number of contributors, (iii) more type annotations help find more type errors (0.704 correlation), but nevertheless, many commits (78.3%) are committed despite having such errors. Our findings show that better developer training and automated techniques for adding type annotations are needed, as most code still remains unannotated, and they call for a better integration of gradual type checking into the development process.

[**Dias2021**] Edson Dias, Paulo Meirelles, Fernando Castor, Igor Steinmacher, Igor Wiese, and Gustavo Pinto. What makes a great maintainer of open source projects? In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, May 2021, DOI 10.1109/icse43902.2021.00093.
Abstract: Although Open Source Software (OSS) maintainers devote a significant proportion of their work to coding tasks, great maintainers must excel in many other activities beyond coding. Maintainers should care about fostering a community, helping new members to find their place, while also saying "no" to patches that although are well-coded and well-tested, do not contribute to the goal of the project. To perform all these activities masterfully, maintainers should exercise attributes that software engineers (working on closed source projects) do not always need to master. This paper aims to uncover, relate, and prioritize the unique attributes that great OSS maintainers might have. To achieve this goal, we conducted 33 semi-structured interviews with well-experienced maintainers that are the gatekeepers of notable projects such as the Linux Kernel, the Debian operating system, and the GitLab coding platform. After we analyzed the interviews and curated a list of attributes, we created a conceptual framework to explain how these attributes are connected. We then conducted a rating survey with 90 OSS contributors. We noted that "technical excellence" and "communication" are the most recurring attributes. When grouped, these attributes fit into four broad categories: management, social, technical, and personality. While we noted that "sustain a long term vision of the project" and being "extremely

careful" seem to form the basis of our framework, we noted through our survey that the communication attribute was perceived as the most essential one.

[**Dickson2022**] Paul E. Dickson, Tim Richards, and Brett A. Becker. Experiences implementing and utilizing a notional machine in the classroom. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education*. ACM, Feb 2022, DOI 10.1145/3478431.3499320.
Abstract: In the computing education community, discussion is growing about the benefits of teaching programming by explicitly using notional machines to help students. To-date most work is largely theoretical and little work addresses actually using them in a classroom. This paper documents our experience of creating a notional machine for a specific course and using it in that classroom. A key point we learned while creating this notional machine is that many of the difficulties encountered were due to the concept of a notional machine being tightly coupled to students' mental models. Although not surprising, the numerous complications this brings are important to overcome. The potential amount of detail included in the notional machine is enormously influenced by the students' mental models, which are likely specific to a course, and also change throughout a semester – and certainly across several semesters. We present lessons learned from this experience, among them that implementing a notional machine and using it in class is a non-trivial yet possibly beneficial exercise.

[**Diercks2022**] Philipp Diercks, Dennis Gläser, Ontje Lünsdorf, Michael Selzer, Bernd Flemisch, and Jörg F. Unger. Evaluation of tools for describing, reproducing and reusing scientific workflows, 2022.
Abstract: In the field of computational science and engineering, workflows often entail the application of various software, for instance, for simulation or pre- and postprocessing. Typically, these components have to be combined in arbitrarily complex workflows to address a specific research question. In order for peer researchers to understand, reproduce and (re)use the findings of a scientific publication, several challenges have to be addressed. For instance, the employed workflow has to be automated and information on all used software must be available for a reproduction of the results. Moreover, the results must be traceable and the workflow documented and readable to allow for external verification and greater trust. In this paper, existing workflow management systems (WfMSs) are discussed regarding their suitability for describing, reproducing and reusing scientific workflows. To this end, a set of general requirements for WfMSs were deduced from user stories that we deem relevant in the domain of computational science and engineering. On the basis of an exemplary workflow implementation, publicly hosted at GitHub, a selection of different WfMSs is compared with respect to these requirements, to support fellow scientists in identifying the WfMSs that best suit their requirements.

[**Do2022**] Lisa Nguyen Quang Do, James R. Wright, and Karim Ali. Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. *IEEE Transactions on Software Engineering*, 48(3):835–847, Mar 2022, DOI 10.1109/tse.2020.3004525.

Abstract: As increasingly complex software is developed every day, a growing number of companies use static analysis tools to reason about program properties ranging from simple coding style rules to more advanced software bugs, to multi-tier security vulnerabilities. While increasingly complex analyses are created, developer support must also be updated to ensure that the tools are used to their best potential. Past research in the usability of static analysis tools has primarily focused on usability issues encountered by software developers, and the causes of those issues in analysis tools. In this article, we adopt a more user-centered approach, and aim at understanding why software developers use analysis tools, which decisions they make when using those tools, what they look for when making those decisions, and the motivation behind their strategies. This approach allows us to derive new tool requirements that closely support software developers (e.g., systems for recommending warnings to fix that take developer knowledge into account), and also open novel avenues for further static-analysis research such as collaborative user interfaces for analysis warnings.

[**Dogan2022**] Emre Doğan and Eray Tüzün. Towards a taxonomy of code review smells. *Inf. Softw. Technol.*, 142(106737):106737, Feb 2022, DOI 10.1016/j.infsof.2021.106737.

Abstract: Context: Code review is a crucial step of the software development life cycle in order to detect possible problems in source code before merging the changeset to the codebase. Although there is no consensus on a formally defined life cycle of the code review process, many companies and open source software (OSS) communities converge on common rules and best practices. In spite of minor differences in different platforms, the primary purpose of all these rules and practices leads to a faster and more effective code review process. Non-conformance of developers to this process does not only reduce the advantages of the code review but can also introduce waste in later stages of the software development. Objectives: The aim of this study is to provide an empirical understanding of the bad practices followed in the code review process, that are code review (CR) smells. Methods: We first conduct a multivocal literature review in order to gather code review bad practices discussed in white and gray literature. Then, we conduct a targeted survey with 32 experienced software practitioners and perform follow-up interviews in order to get their expert opinion. Based on this process, a taxonomy of code review smells is introduced. To quantitatively demonstrate the existence of these smells, we analyze 226,292 code reviews collected from eight OSS projects. Results: We observe that a considerable number of code review smells exist in all projects with varying degrees of ratios. The empirical results illustrate that 72.2% of the code reviews among eight projects are affected by at least one code review smell.

Conclusion: The empirical analysis shows that the OSS projects are substantially affected by the code review smells. The provided taxonomy could provide a foundation for best practices and tool support to detect and avoid code review smells in practice.

[**Dong2023**] Yiwen Dong, Zheyang Li, Yongqiang Tian, Chengnian Sun, Michael W. Godfrey, and Meiyappan Nagappan. Bash in the wild: Language usage, code smells, and bugs. *ACM Transactions on Software Engineering and Methodology*, 32(1):1–22, Jan 2023, DOI 10.1145/3517193.
Abstract: The Bourne-again shell (Bash) is a prevalent scripting language for orchestrating shell commands and managing resources in Unix-like environments. It is one of the mainstream shell dialects that is available on most GNU Linux systems. However, the unique syntax and semantics of Bash could easily lead to unintended behaviors if carelessly used. Prior studies primarily focused on improving reliability of Bash scripts or facilitating writing Bash scripts; there is yet no empirical study on the characteristics of Bash programs written in reality, e.g., frequently used language features, common code smells and bugs. In this paper, we perform a large-scale empirical study of Bash usage, based on analyses over one million open-source Bash scripts found in Github repositories. We identify and discuss which features and utilities of Bash are most often used. Using static analysis, we ind that Bash scripts are often error prone, and the error-proneness has a moderately positive correlation with the size of the scripts. We also ind that the most common problem areas concern quoting, resource management, command options, permissions, and error handling. We envision that these indings can be beneicial for learning Bash and future research that aims to improve shell and command-line productivity and reliability.

[**Drage2022**] Eleanor Drage and Kerry Mackereth. Does AI debias recruitment? race, gender, and AI's "eradication of difference". *Philos. Technol.*, 35(4):89, Oct 2022, DOI 10.1007/s13347-022-00543-1.
Abstract: In this paper, we analyze two key claims offered by recruitment AI companies in relation to the development and deployment of AI-powered HR tools: (1) recruitment AI can objectively assess candidates by removing gender and race from their systems, and (2) this removal of gender and race will make recruitment fairer, help customers attain their DEI goals, and lay the foundations for a truly meritocratic culture to thrive within an organization. We argue that these claims are misleading for four reasons: First, attempts to "strip" gender and race from AI systems often misunderstand what gender and race are, casting them as isolatable attributes rather than broader systems of power. Second, the attempted outsourcing of "diversity work" to AI-powered hiring tools may unintentionally entrench cultures of problems within organizations. Third, AI hiring tools' supposedly neutral assessment of candidates' traits belie the power relationship between the observer and the observed. Specifically, the racialized history of character analysis and its associated processes of classification and categorization play into longer histories of taxonomical sorting and reflect the current demands

and desires of the job market, even when not explicitly conducted along the lines of gender and race. Fourth, recruitment AI tools help produce the "ideal candidate" that they supposedly identify through by constructing associations between words and people's bodies. From these four conclusions outlined above, we offer three key recommendations to AI HR firms, their customers, and policy makers going forward.

[**Dyer2022**] Robert Dyer and Jigyasa Chauhan. An exploratory study on the predominant programming paradigms in Python code. In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, Nov 2022, DOI 10.1145/3540250.3549158.

Abstract: Python is a multi-paradigm programming language that fully supports object-oriented (OO) programming. The language allows writing code in a non-procedural imperative manner, using procedures, using classes, or in a functional style. To date, no one has studied what paradigm(s), if any, are predominant in Python code and projects. In this work, we first define a technique to classify Python files into predominant paradigm(s). We then automate our approach and evaluate it against human judgements, showing over 80% agreement. We then analyze over 100k open-source Python projects, automatically classifying each source file and investigating the paradigm distributions. The results indicate Python developers tend to heavily favor OO features. We also observed a positive correlation between OO and procedural paradigms and the size of the project. And despite few files or projects being predominantly functional, we still found many functional feature uses.

[**Dykstra2022**] Josiah Dykstra, Kelly Shortridge, Jamie Met, and Douglas Hough. Sludge for good: Slowing and imposing costs on cyber attackers, 2022.

Abstract: Choice architecture describes the design by which choices are presented to people. Nudges are an aspect intended to make "good" outcomes easy, such as using password meters to encourage strong passwords. Sludge, on the contrary, is friction that raises the transaction cost and is often seen as a negative to users. Turning this concept around, we propose applying sludge for positive cybersecurity outcomes by using it offensively to consume attackers' time and other resources. To date, most cyber defenses have been designed to be optimally strong and effective and prohibit or eliminate attackers as quickly as possible. Our complimentary approach is to also deploy defenses that seek to maximize the consumption of the attackers' time and other resources while causing as little damage as possible to the victim. This is consistent with zero trust and similar mindsets which assume breach. The Sludge Strategy introduces cost-imposing cyber defense by strategically deploying friction for attackers before, during, and after an attack using deception and authentic design features. We present the characteristics of effective sludge, and show a continuum from light to heavy sludge. We describe the quantitative and qualitative costs to attackers

and offer practical considerations for deploying sludge in practice. Finally, we examine real-world examples of U.S. government operations to frustrate and impose cost on cyber adversaries.

[**Eid2023**] Elias Eid and Nancy A. Day. Static profiling of alloy models. *IEEE Transactions on Software Engineering*, 49(2):743–759, Feb 2023, DOI 10.1109/tse.2022.3162985.
Abstract: Modeling of software-intensive systems using formal declarative modeling languages offers a means of managing software complexity through the use of abstraction and early identification of correctness issues by formal analysis. Alloy is one such language used for modeling systems early in the development process. Little work has been done to study the styles and techniques commonly used in Alloy models. We present the first static analysis study of Alloy models. We investigate research questions that examine a large corpus of 1,652 Alloy models. To evaluate these research questions, we create a methodology that leverages the power of ANTLR pattern matching and the query language XPath. Our research questions are split into two categories depending on their purpose. The Model Characteristics category aims to identify what language constructs are used commonly. Modeling Practices questions are considerably more complex and identify how modelers are using Alloy's constructs. We also evaluate our research questions on a subset of models from our corpus written by expert modelers. We compare the results of the expert corpus to the results obtained from the general corpus to gain insight into how expert modelers use the Alloy language. We draw conclusions from the findings of our research questions and present actionable items for educators, language and environment designers, and tool developers. Actionable items for educators are intended to highlight underutilized language constructs and features, and help student modelers avoid discouraged practices. Actionable items aimed at language designers present ways to improve the Alloy language by adding constructs or removing unused ones based on trends identified in our corpus of models. The actionable items aimed at environment designers address features to facilitate model creation. Actionable items for tool developers provide suggestions for backend optimizations.

[**Etemadi2022**] Khashayar Etemadi, Aman Sharma, Fernanda Madeiral, and Martin Monperrus. Augmenting diffs with runtime information, 2022.

[**Faria2023**] João Pascoal Faria and Rui Abreu. Case studies of development of verified programs with dafny for accessibility assessment, 2023.
Abstract: Formal verification techniques aim at formally proving the correctness of a computer program with respect to a formal specification, but the expertise and effort required for applying formal specification and verification techniques and scalability issues have limited their practical application. In recent years, the tremendous progress with SAT and SMT solvers enabled the construction of a new generation of tools that promise to make formal verification more accessible for software engineers, by automating

most if not all of the verification process. The Dafny system is a prominent example of that trend. However, little evidence exists yet about its accessibility. To help fill this gap, we conducted a set of 10 case studies of developing verified implementations in Dafny of some real-world algorithms and data structures, to determine its accessibility for software engineers. We found that, on average, the amount of code written for specification and verification purposes is of the same order of magnitude as the traditional code written for implementation and testing purposes (ratio of 1.14)—an "overhead" that certainly pays off for high-integrity software. The performance of the Dafny verifier was impressive, with 2.4 proof obligations generated per line of code written, and 24 ms spent per proof obligation generated and verified, on average. However, we also found that the manual work needed in writing auxiliary verification code may be significant and difficult to predict and master. Hence, further automation and systematization of verification tasks are possible directions for future advances in the field.

[**Farzat2021**] Fabio de A. Farzat, Marcio de O. Barros, and Guilherme H. Travassos. Evolving JavaScript code to reduce load time. *IEEE Transactions on Software Engineering*, 47(8):1544–1558, Aug 2021, DOI 10.1109/tse.2019.2928293.

Abstract: JavaScript is one of the most used programming languages for front-end development of Web applications. The increase in complexity of front-end features brings concerns about performance, especially the load and execution time of JavaScript code. In this paper, we propose an evolutionary program improvement technique to reduce the size of JavaScript programs and, therefore, the time required to load and execute them in Web applications. To guide the development of this technique, we performed an experimental study to characterize the patches applied to JavaScript programs to reduce their size while keeping the functionality required to pass all test cases in their test suites. We applied this technique to 19 JavaScript programs varying from 92 to 15,602 LOC and observed reductions from 0.2 to 73.8 percent of the original code, as well as a relationship between the quality of a program's test suite and the ability to reduce the size of its source code.

[**Feal2020**] Álvaro Feal, Paolo Calciati, Narseo Vallina-Rodriguez, Carmela Troncoso, and Alessandra Gorla. Angel or devil? a privacy study of mobile parental control apps. *Proceedings on Privacy Enhancing Technologies*, 2020(2):314–335, Apr 2020, DOI 10.2478/popets-2020-0029.

Abstract: Android parental control applications are used by parents to monitor and limit their children's mobile behaviour (e.g., mobile apps usage, web browsing, calling, and texting). In order to offer this service, parental control apps require privileged access to sys-tem resources and access to sensitive data. This may significantly reduce the dangers associated with kids' online activities, but it raises important privacy con-cerns. These concerns have so far been overlooked by organizations providing recommendations regarding the use of parental control applications to the public. We conduct

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

[**Feld2022**] Jan Feld, Edwin Ip, Andreas Leibbrandt, and Joseph Vecci. Identifying and overcoming gender barriers in tech: A field experiment on inaccurate statistical discrimination. *SSRN Electronic Journal*, 2022, DOI `10.2139/ssrn.4238277`.

Abstract: Women are significantly underrepresented in the technology sector. We design a field experiment to identify statistical discrimination in job applicant assessments and test treatments to help improve hiring of the best applicants. In our experiment, we measure the programming skills of job applicants for a programming job. Then, we recruit a sample of employers consisting of human resource and tech professionals and incentivize them to assess the performance of these applicants based on their resumes. We find evidence consistent with inaccurate statistical discrimination: while there are no significant gender differences in performance, employers believe that female programmers perform worse than male programmers. This belief is strongest among female employers, who are more prone to selection neglect than male employers. We also find experimental evidence that statistical discrimination can be mitigated. In two treatments, in which we provide assessors with additional information on the applicants' aptitude or personality, we find no gender differences in the perceived applicant performance. Together, these findings show the malleability of statistical discrimination and provide levers to improve hiring and reduce gender imbalance.

[**Feng2022**] Runhan Feng, Ziyang Yan, Shiyan Peng, and Yuanyuan Zhang. Automated detection of password leakage from public GitHub repositories. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, May 2022, DOI `10.1145/3510003.3510150`.

Abstract: The prosperity of the GitHub community has raised new concerns about data security in public repositories. Practitioners who manage authentication secrets such as textual passwords and API keys in the source code may accidentally leave these texts in the public repositories, resulting in secret leakage. If such leakage in the source code can be automatically detected in time, potential damage would be avoided. With existing approaches

focusing on detecting secrets with distinctive formats (e.g., API keys, cryptographic keys in PEM format), textual passwords, which are ubiquitously used for authentication, fall through the crack. Given that textual passwords could be virtually any strings, a naive detection scheme based on regular expression performs poorly. This paper presents PassFinder, an automated approach to effectively detecting password leakage from public repositories that involve various programming languages on a large scale. PassFinder utilizes deep neural networks to unveil the intrinsic characteristics of textual passwords and understand the semantics of the code snippets that use textual passwords for authentication, i.e., the contextual information of the passwords in the source code. Using this new technique, we performed the first large-scale and longitudinal analysis of password leakage on GitHub. We inspected newly uploaded public code files on GitHub for 75 days and found that password leakage is pervasive, affecting over sixty thousand repositories. Our work contributes to a better understanding of password leakage on GitHub, and we believe our technique could promote the security of the open-source ecosystem.

[**Ferreira2021**] Fabio Ferreira, Hudson Silva Borges, and Marco Tulio Valente. On the (un-)adoption of JavaScript front-end frameworks. *Software: Practice and Experience*, 52(4):947–966, Oct 2021, DOI `10.1002/spe.3044`.
Abstract: JavaScript is characterized by a rich ecosystem of libraries and frameworks. A key element in this ecosystem are frameworks used for implementing the front-end of web-based applications, such as Vue and React. However, despite their relevance, we have few works investigating the factors that drive the adoption—and un-adoption—of front-end-based JavaScript frameworks. Therefore, in this paper, we first report the results of a survey with 49 developers where we asked them to describe the factors they consider when selecting a front-end framework. In the second part of the work, we focus on projects that migrate from one framework to another since JavaScript's ecosystem is also very dynamic. Finally, we provide a quantitative characterization of the migration effort and reveal the main barriers faced by the developers during this effort. Although not completely generalizable, our central findings are as follows: (a) popularity and learnability are the key factors that motivate the choice of front-end frameworks in JavaScript; (b) from the 49 surveyed developers, one out of four have plans to migrate to another framework in the future; (c) the time spent performing the migration is greater than or equal to the time spent using the old framework in all studied projects. We conclude with a list of implications for practitioners, framework developers, tool builders, and researchers.

[**Ferreira2022**] Fabio Ferreira, Hudson Silva Borges, and Marco Tulio Valente. On the (un-)adoption of JavaScript front-end frameworks. *Software: Practice and Experience*, 52(4):947–966, Oct 2021, DOI `10.1002/spe.3044`.
Abstract: JavaScript is characterized by a rich ecosystem of libraries and frameworks. A key element in this ecosystem are frameworks used for implementing the front-end of web-based applications, such as Vue and React.

However, despite their relevance, we have few works investigating the factors that drive the adoption—and un-adoption—of front-end-based JavaScript frameworks. Therefore, in this article, we first report the results of a survey with 49 developers where we asked them to describe the factors they consider when selecting a front-end framework. In the second part of the work, we focus on projects that migrate from one framework to another since JavaScript's ecosystem is also very dynamic. Finally, we provide a quantitative characterization of the migration effort and reveal the main barriers faced by the developers during this effort. Although not completely generalizable, our central findings are as follows: (a) popularity and learnability are the key factors that motivate the choice of front-end frameworks in JavaScript; (b) from the 49 surveyed developers, one out of four have plans to migrate to another framework in the future; (c) the time spent performing the migration is greater than or equal to the time spent using the old framework in all studied projects. We conclude with a list of implications for practitioners, framework developers, tool builders, and researchers.

[**FinnieAnsley2022**] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. The robots are coming: Exploring the implications of OpenAI codex on introductory programming. In *Australasian Computing Education Conference*. ACM, Feb 2022, DOI 10.1145/3511861.3511863.

Abstract: Recent advances in artificial intelligence have been driven by an exponential growth in digitised data. Natural language processing, in particular, has been transformed by machine learning models such as OpenAI's GPT-3 which generates human-like text so realistic that its developers have warned of the dangers of its misuse. In recent months OpenAI released Codex, a new deep learning model trained on Python code from more than 50 million GitHub repositories. Provided with a natural language description of a programming problem as input, Codex generates solution code as output. It can also explain (in English) input code, translate code between programming languages, and more. In this work, we explore how Codex performs on typical introductory programming problems. We report its performance on real questions taken from introductory programming exams and compare it to results from students who took these same exams under normal conditions, demonstrating that Codex outscores most students. We then explore how Codex handles subtle variations in problem wording using several published variants of the well-known "Rainfall Problem" along with one unpublished variant we have used in our teaching. We find the model passes many test cases for all variants. We also explore how much variation there is in the Codex generated solutions, observing that an identical input prompt frequently leads to very different solutions in terms of algorithmic approach and code length. Finally, we discuss the implications that such technology will have for computing education as it continues to evolve, including both challenges and opportunities.

[**Flyvbjerg2022**] Bent Flyvbjerg, Alexander Budzier, Jong Seok Lee, Mark

Keil, Daniel Lunn, and Dirk W Bester. The empirical reality of IT project cost overruns: Discovering a power-law distribution. *J. Manag. Inf. Syst.*, 39(3):607–639, Jul 2022, DOI 10.1080/07421222.2022.2096544.
Abstract: If managers assume a normal or near-normal distribution of Information Technology (IT) project cost overruns, as is common, and cost overruns can be shown to follow a power-law distribution, managers may be unwittingly exposing their organizations to extreme risk by severely underestimating the probability of large cost overruns. In this research, we collect and analyze a large sample comprised of 5,392 IT projects to empirically examine the probability distribution of IT project cost overruns. Further, we propose and examine a mechanism that can explain such a distribution. Our results reveal that IT projects are far riskier in terms of cost than normally assumed by decision makers and scholars. Specifically, we found that IT project cost overruns follow a power-law distribution in which there are a large number of projects with relatively small overruns and a fat tail that includes a smaller number of projects with extreme overruns. A possible generative mechanism for the identified power-law distribution is found in interdependencies among technological components in IT systems. We propose and demonstrate, through computer simulation, that a problem in a single technological component can lead to chain reactions in which other interdependent components are affected, causing substantial overruns. What the power law tells us is that extreme IT project cost overruns will occur and that the prevalence of these will be grossly underestimated if managers assume that overruns follow a normal or near-normal distribution. This underscores the importance of realistically assessing and mitigating the cost risk of new IT projects up front.

[**Foidl2022**] Harald Foidl, Michael Felderer, and Rudolf Ramler. Data smells. In *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI*. ACM, May 2022, DOI 10.1145/3522664.3528590.
Abstract: High data quality is fundamental for today's AI-based systems. However, although data quality has been an object of research for decades, there is a clear lack of research on potential data quality issues (e.g., ambiguous, extraneous values). These kinds of issues are latent in nature and thus often not obvious. Nevertheless, they can be associated with an increased risk of future problems in AI-based systems (e.g., technical debt, data-induced faults). As a counterpart to code smells in software engineering, we refer to such issues as Data Smells. This article conceptualizes data smells and elaborates on their causes, consequences, detection, and use in the context of AI-based systems. In addition, a catalogue of 36 data smells divided into three categories (i.e., Believability Smells, Understandability Smells, Consistency Smells) is presented. Moreover, the article outlines tool support for detecting data smells and presents the result of an initial smell detection on more than 240 real-world datasets.

[**Fregnan2023**] Enrico Fregnan, Josua Fröhlich, Davide Spadini, and Alberto Bacchelli. Graph-based visualization of merge requests for code

review. *Journal of Systems and Software*, 195:111506, Jan 2023, DOI 10.1016/j.jss.2022.111506.

[**Friend2022**] Michelle Friend, Monica Mcgill, and Anni Reinking. Solve this! K-12 CS education teachers' problems of practice. In *Koli Calling '22: 22nd Koli Calling International Conference on Computing Education Research*. ACM, Nov 2022, DOI 10.1145/3564721.3564738.

Abstract: Problem. Educational research identifies answerable questions, but often does not address the problems K-12 teachers identify as important. Further, academic research findings can be difficult for teachers to apply to their practices and unique contexts. Currently, little research exists on the lived experiences of primary and secondary instructors who teach computer science (CS) or computational thinking (CT) and also on the specific problems of practice teachers face when teaching CS. Research Question. What problems of practice do K-12 teachers face when teaching CS/CT? Method. Data for this qualitative study was collected using an online questionnaire distributed to teachers internationally. CS/CT teachers responded to an open-ended prompt asking for problems related to teaching CS. The data was analyzed using descriptive first-round coding and focused second-round coding. Validity was established through collaborative coding. Analysis was theorized using locus of control. Findings. Problems with students encompassed behavioral, cognitive, and attitudinal issues, as well as lack of home support or resources. Teachers identified many problems of policy notably stemming from lack of resources or support from administrators. A smaller number of challenges, such as lack of content knowledge, were situated within teachers themselves. While some problems such as student motivation are general, a number of responses identified unique challenges in CS education compared to other disciplines. Implications. Identifying problems faced by teachers can guide professional development offerings, help researchers develop studies that would result in meaningful improvement to CS education, and suggest policy decisions which would result in better outcomes for students.

[**Furia2022**] Carlo A. Furia, Richard Torkar, and Robert Feldt. Applying bayesian analysis guidelines to empirical software engineering data: The case of programming languages and code quality. *ACM Transactions on Software Engineering and Methodology*, 31(3):1–38, Mar 2022, DOI 10.1145/3490953.

Abstract: Statistical analysis is the tool of choice to turn data into information, and then information into empirical knowledge. The process that goes from data to knowledge is, however, long, uncertain, and riddled with pitfalls. To be valid, it should be supported by detailed, rigorous guidelines, which help ferret out issues with the data or model, and lead to qualified results that strike a reasonable balance between generality and practical relevance. Such guidelines are being developed by statisticians to support the latest techniques for Bayesian data analysis. In this article, we frame these

guidelines in a way that is apt to empirical research in software engineering. To demonstrate the guidelines in practice, we apply them to reanalyze a GitHub dataset about code quality in different programming languages. The dataset's original analysis (Ray et al., 2014) and a critical reanalysis (Berger et al., 2019) have attracted considerable attention—in no small part because they target a topic (the impact of different programming languages) on which strong opinions abound. The goals of our reanalysis are largely orthogonal to this previous work, as we are concerned with demonstrating, on data in an interesting domain, how to build a principled Bayesian data analysis and to showcase its benefits. In the process, we will also shed light on some critical aspects of the analyzed data and of the relationship between programming languages and code quality—such as the impact of project-specific characteristics other than the used programming language. The high-level conclusions of our exercise will be that Bayesian statistical techniques can be applied to analyze software engineering data in a way that is principled, flexible, and leads to convincing results that inform the state of the art while highlighting the boundaries of its validity. The guidelines can support building solid statistical analyses and connecting their results, and hence help buttress continued progress in empirical software engineering research.

[**Furia2023**] Carlo A. Furia, Richard Torkar, and Robert Feldt. Towards causal analysis of empirical software engineering data: The impact of programming languages on coding competitions, 2023.
Abstract: There is abundant observational data in the software engineering domain, whereas running large-scale controlled experiments is often practically impossible. Thus, most empirical studies can only report statistical correlations—instead of potentially more insightful and robust causal relations. This paper discusses some novel techniques that support analyzing purely observational data for causal relations. Using fundamental causal models such as directed acyclic graphs, one can rigorously express, and partially validate, causal hypotheses; and then use the causal information to guide the construction of a statistical model that captures genuine causal relations—such that correlation does imply causation. We apply these ideas to analyzing public data about programmer performance in Code Jam, a large world-wide coding contest organized by Google every year. Specifically, we look at the impact of different programming languages on a participant's performance in the contest. While the overall effect associated with programming languages is weak compared to other variables—regardless of whether we consider correlational or causal links—we found considerable differences between a purely statistical and a causal analysis of the very same data. The takeaway message is that even an imperfect causal analysis of observational data can help answer the salient research questions more precisely and more robustly than with just purely statistical techniques.

[**Gaffney2022**] Kevin P Gaffney, Martin Prammer, Larry Brasfield, D Richard Hipp, Dan Kennedy, and Jignesh M Patel. SQLite. *Proceedings VLDB En-*

dowment, 15(12):3535–3547, Aug 2022, DOI 10.14778/3554821.3554842.
Abstract: In the two decades following its initial release, SQLite has become the most widely deployed database engine in existence. Today, SQLite is found in nearly every smartphone, computer, web browser, television, and automobile. Several factors are likely responsible for its ubiquity, including its in-process design, standalone codebase, extensive test suite, and cross-platform file format. While it supports complex analytical queries, SQLite is primarily designed for fast online transaction processing (OLTP), employing row-oriented execution and a B-tree storage format. However, fueled by the rise of edge computing and data science, there is a growing need for efficient in-process online analytical processing (OLAP). DuckDB, a database engine nicknamed "the SQLite for analytics", has recently emerged to meet this demand. While DuckDB has shown strong performance on OLAP benchmarks, it is unclear how SQLite compares. Furthermore, we are aware of no work that attempts to identify root causes for SQLite's performance behavior on OLAP workloads. In this paper, we discuss SQLite in the context of this changing workload landscape. We describe how SQLite evolved from its humble beginnings to the full-featured database engine it is today. We evaluate the performance of modern SQLite on three benchmarks, each representing a different flavor of in-process data management, including transactional, analytical, and blob processing. We delve into analytical data processing on SQLite, identifying key bottlenecks and weighing potential solutions. As a result of our optimizations, SQLite is now up to 4.2X faster on SSB. Finally, we discuss the future of SQLite, envisioning how it will evolve to meet new demands and challenges.

[**Galappaththi2022**] Akalanka Galappaththi, Sarah Nadi, and Christoph Treude. Does this apply to me? In *Proc. International Conference on Mining Software Repositories (MSR)*. ACM, May 2022, DOI 10.1145/3524842.3528435.
Abstract: Stack Overflow has become an essential technical resource for developers. However, given the vast amount of knowledge available on Stack Overflow, finding the right information that is relevant for a given task is still challenging, especially when a developer is looking for a solution that applies to their specific requirements or technology stack. Clearly marking answers with their technical context, i.e., the information that characterizes the technologies and assumptions needed for this answer, is potentially one way to improve navigation. However, there is no information about how often such context is mentioned, and what kind of information it might offer. In this paper, we conduct an empirical study to understand the occurrence of technical context in Stack Overflow answers and comments, using tags as a proxy for technical context. We specifically focus on additional context, where answers/comments mention information that is not already discussed in the question. Our results show that nearly half of our studied threads contain at least one additional context. We find that almost 50% of the additional context are either a library/framework, a programming language,

a tool/application, an API, or a database. Overall, our findings show the promise of using additional context as navigational cues.

[**Gamblin2022**] Todd Gamblin, Massimiliano Culpo, Gregory Becker, and Sergei Shudler. Using answer set programming for hpc dependency solving, 2022.

[**Georgiou2022**] Stefanos Georgiou, Maria Kechagia, Tushar Sharma, Federica Sarro, and Ying Zou. Green AI. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, May 2022, DOI 10.1145/3510003.3510221.
Abstract: The use of Artificial Intelligence (AI), and more specifically of Deep Learning (DL), in modern software systems, is nowadays widespread and continues to grow. At the same time, its usage is energy de-manding and contributes to the increased CO2 emissions, and has a great financial cost as well. Even though there are many studies that examine the capabilities of DL, only a few focus on its green aspects, such as energy consumption. This paper aims at raising awareness of the costs incurred when using different DL frameworks. To this end, we perform a thorough empirical study to measure and compare the energy consumption and run-time performance of six different DL models written in the two most popular DL frameworks, namely PYTORCH and TENSORFLOW. We use a well-known benchmark of DL models, Deep LEARNINGEXAMPLES, created by NVIDIA, to compare both the training and inference costs of DL. Finally, we manually investigate the functions of these frameworks that took most of the time to execute in our experiments. The results of our empirical study reveal that there is a statistically significant difference between the cost incurred by the two DL frameworks in 94% of the cases studied. While Tensorflow achieves significantly better energy and run-time performance than PYTORCH, and with large effect sizes in 100% of the cases for the training phase, PYTORCH instead exhibits significantly better energy and run-time performance than TENSORFLOW in the inference phase for 66% of the cases, always, with large effect sizes. Such a large difference in performance costs does not, however, seem to affect the accuracy of the models produced, as both frameworks achieve comparable scores under the same configurations. Our manual analysis, of the documentation and source code of the functions examined, reveals that such a difference in performance costs is under-documented, in these frameworks. This suggests that developers need to improve the documentation of their DL frameworks, the source code of the functions used in these frameworks, as well as to enhance existing DL algorithms.

[**Getseva2022**] Vanesa Getseva and Amruth N Kumar. An empirical analysis of code-tracing concepts. In *Proc. Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, Jul 2022, DOI 10.1145/3502718.3524794.
Abstract: Which code-tracing concepts are introductory programming students likely to learn from classroom instruction and which ones need addi-

tional problem-solving practice to master? Are there relationships among programming concepts that can be used to build adaptive assessment instruments? To answer these questions, we analyzed the data collected over several semesters by a suite of code-tracing tutors called problets, that administered pre-test, practice, post-test protocol. Each tutor covered a single programming topic, which consisted of 9-25 concepts. For each concept, we used the pretest data to calculate the probability that students knew the concept before using the tutor. Using a weighted average of the concept probabilities, we found that students had learned some topics more than others: if/if-else (0.85), function behavior (0.76), arrays (0.73), while (0.7), for (0.69), switch (0.67), and debugging functions (0.55). Some of the concepts on which students needed additional practice included bugs, nested loops and back-to-back loops. Expressions, even when used in novel contexts, were not challenging for students. We built a Bayesian network for each topic based on conditional probabilities to discover the concepts that must be covered, and those whose coverage is redundant in the presence of other concepts. A strength of this empirical study is that it uses a large dataset collected from multiple institutions over multiple semesters. We also list threats to the validity of the study.

[**Ghorbani2023**] Amir Ghorbani, Nathan Cassee, Derek Robinson, Adam Alami, Neil A. Ernst, Alexander Serebrenik, and Andrzej Wasowski. Autonomy is an acquired taste: Exploring developer preferences for github bots, 2023.

Abstract: Software bots fulfill an important role in collective software development, and their adoption by developers promises increased productivity. Past research has identified that bots that communicate too often can irritate developers, which affects the utility of the bot. However, it is not clear what other properties of human-bot collaboration affect developers' preferences, or what impact these properties might have. The main idea of this paper is to explore characteristics affecting developer preferences for interactions between humans and bots, in the context of GitHub pull requests. We carried out an exploratory sequential study with interviews and a subsequent vignette-based survey. We find developers generally prefer bots that are personable but show little autonomy, however, more experienced developers tend to prefer more autonomous bots. Based on this empirical evidence, we recommend bot developers increase configuration options for bots so that individual developers and projects can configure bots to best align with their own preferences and project cultures.

[**Gorz2022**] Philipp Görz, Björn Mathis, Keno Hassler, Emre Güler, Thorsten Holz, Andreas Zeller, and Rahul Gopinath. How to compare fuzzers, 2022.

[**Gote2022**] Christoph Gote, Pavlin Mavrodiev, Frank Schweitzer, and Ingo Scholtes. Big data = big insights? operationalising brooks' law in a massive github data set, 2022.

[**Graziotin2022**] Daniel Graziotin, Per Lenberg, Robert Feldt, and Stefan Wagner. Psychometrics in behavioral software engineering: A methodological introduction with guidelines. *ACM Trans. Softw. Eng. Methodol.*, 31(1):1–36, Jan 2022, DOI 10.1145/3469888.

Abstract: A meaningful and deep understanding of the human aspects of software engineering (SE) requires psychological constructs to be considered. Psychology theory can facilitate the systematic and sound development as well as the adoption of instruments (e.g., psychological tests, questionnaires) to assess these constructs. In particular, to ensure high quality, the psychometric properties of instruments need evaluation. In this article, we provide an introduction to psychometric theory for the evaluation of measurement instruments for SE researchers. We present guidelines that enable using existing instruments and developing new ones adequately. We conducted a comprehensive review of the psychology literature framed by the Standards for Educational and Psychological Testing. We detail activities used when operationalizing new psychological constructs, such as item pooling, item review, pilot testing, item analysis, factor analysis, statistical property of items, reliability, validity, and fairness in testing and test bias. We provide an openly available example of a psychometric evaluation based on our guideline. We hope to encourage a culture change in SE research towards the adoption of established methods from psychology. To improve the quality of behavioral research in SE, studies focusing on introducing, validating, and then using psychometric instruments need to be more common.

[**Greiler2022**] Michaela Greiler, Margaret-Anne Storey, and Abi Noda. An actionable framework for understanding and improving developer experience, 2022.

Abstract: Developer experience is an important concern for software organizations as enhancing developer experience improves productivity, satisfaction, engagement and retention. We set out to understand what affects developer experience through semi-structured interviews with 21 developers from industry, which we transcribed and iteratively coded. Our findings elucidate factors that affect developer experience and characteristics that influence their respective importance to individual developers. We also identify strategies employed by individuals and teams to improve developer experience and the barriers that stand in their way. Lastly, we describe the coping mechanisms of developers when developer experience cannot be sufficiently improved. Our findings result in the DX Framework, an actionable conceptual framework for understanding and improving developer experience. The DX Framework provides a go-to reference for organizations that want to enable more productive and effective work environments for their developers.

[**Grotov2022**] Konstantin Grotov, Sergey Titov, Vladimir Sotnikov, Yaroslav Golubev, and Timofey Bryksin. A large-scale comparison of python code in jupyter notebooks and scripts, 2022.

[**Guizani2022a**] Mariam Guizani, Thomas Zimmermann, Anita Sarma, and Denae Ford. Attracting and retaining oss contributors with a maintainer dashboard, 2022.

[**Guizani2022b**] Mariam Guizani, Igor Steinmacher, Jillian Emard, Abrar Fallatah, Margaret Burnett, and Anita Sarma. How to debug inclusivity bugs? a debugging process with information architecture, 2022, DOI 10.1145/3510458.3513009.

[**Haduong2019**] Paulina Haduong. "i like computers. I hate coding": a portrait of two teens' experiences. *Inf. Learn. Sci.*, 120(5/6):349–365, May 2019, DOI 10.1108/ILS-05-2018-0037.

Abstract: Purpose Some empirical evidence suggests that historically marginalized young people may enter introductory programming experiences with skepticism or reluctance, because of negative perceptions of the computing field. This paper aims to explore how learner identity and motivation can affect their experiences in an introductory computer science (CS) experience, particularly for young people who have some prior experience with computing. In this program, learners were asked to develop digital media artifacts about civic issues using Scratch, a block-based programming language. Design/methodology/approach Through participant observation as a teacher and designer of the course, artifact analysis of student-generated computer programs and design journals, as well as with two follow-up 1-h interviews, the author used the qualitative method of portraiture to examine how two reluctant learners experienced a six-week introductory CS program. Findings These learners' experiences illuminate the ways in which identity, community and competence can play a role in supporting learner motivation in CS education experiences. Research limitations/implications As more students have multiple introductory computing encounters, educators need to take into account not only their perceptions of the computing field more broadly but also specific prior encounters with programming. Because of the chosen research approach, the research results may lack generalizability. Researchers are encouraged to explore other contexts and examples further. Practical implications This portrait highlights the need for researchers and educators to take into account student motivation in the design of learning environments. Originality/value This portrait offers a novel examination of novice programmer experiences through the choice in method, as well as new examples of how learner identity can affect student motivation.

[**Hartel2022**] Johannes Härtel and Ralf Lämmel. Operationalizing threats to MSR studies by simulation-based testing. In *Proc. International Conference on Mining Software Repositories (MSR)*. ACM, May 2022, DOI 10.1145/3524842.3527960.

Abstract: Quantitative studies on the border between Mining Software Repository (MSR) and Empirical Software Engineering (ESE) apply data analysis methods, like regression modeling, statistic tests or correlation analysis, to commits or pulls to better understand the software development

process. Such studies assure the validity of the reported results by following a sound methodology. However, with increasing complexity, parts of the methodology can still go wrong. This may result in MSR/ESE studies with undetected threats to validity. In this paper, we propose to systematically protect against threats by operationalizing their treatment using simulations. A simulation substitutes observed and unobserved data, related to an MSR/ESE scenario, with synthetic data, carefully defined according to plausible assumptions on the scenario. Within a simulation, unobserved data becomes transparent, which is the key difference to a real study, necessary to detect threats to an analysis methodology. Running an analysis methodology on synthetic data may detect basic technical bugs and misinterpretations, but it also improves the trust in the methodology. The contribution of a simulation is to operationalize testing the impact of important assumptions. Assumptions still need to be rated for plausibility. We evaluate simulation-based testing by operationalizing undetected threats in the context of four published MSR/ESE studies. We recommend that future research uses such more systematic treatment of threats, as a contribution against the reproducibility crisis.

[**Head2020**] Andrew Head, Jason Jiang, James Smith, Marti A Hearst, and Björn Hartmann. Composing flexibly-organized step-by-step tutorials from linked source code, snippets, and outputs. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. ACM, Apr 2020, DOI 10.1145/3313831.3376798.

Abstract: Programming tutorials are a pervasive, versatile medium for teaching programming. In this paper, we report on the content and structure of programming tutorials, the pain points authors experience in writing them, and a design for a tool to help improve this process. An interview study with 12 experienced tutorial authors found that they construct documents by interleaving code snippets with text and illustrative outputs. It also revealed that authors must often keep related artifacts of source programs, snippets, and outputs consistent as a program evolves. A content analysis of 200 frequently-referenced tutorials on the web also found that most tutorials contain related artifacts—duplicate code and outputs generated from snippets—that an author would need to keep consistent with each other. To address these needs, we designed a tool called Torii with novel authoring capabilities. An in-lab study showed that tutorial authors can successfully use the tool for the unique affordances identified, and provides guidance for designing future tools for tutorial authoring.

[**Hellman2022**] Jazlyn Hellman, Jiahao Chen, Md Sami Uddin, Jinghui Cheng, and Jin L C Guo. Characterizing user behaviors in open-source software user forums. In *Proceedings of the 15th International Conference on Cooperative and Human Aspects of Software Engineering*. ACM, May 2022, DOI 10.1145/3528579.3529178.

Abstract: User forums of Open Source Software (OSS) enable end-users to collaboratively discuss problems concerning the OSS applications. Despite

decades of research on OSS, we know very little about how end-users engage with OSS communities on these forums, in particular, the challenges that hinder their continuous and meaningful participation in the OSS community. Many previous works are developer-centric and overlook the importance of end-user forums. As a result, end-users' expectations are seldom reflected in OSS development. To better understand user behaviors in OSS user forums, we carried out an empirical study analyzing about 1.3 million posts from user forums of four popular OSS applications: Zotero, Audacity, VLC, and RStudio. Through analyzing the contribution patterns of three common user types (end-users, developers, and organizers), we observed that end-users not only initiated most of the threads (above 96% of threads in three projects, 86% in the other), but also acted as the significant contributors for responding to other users' posts, even though they tended to lack confidence in their activities as indicated by psycho-linguistic analyses. Moreover, we found end-users more open, reflecting a more positive emotion in communication than organizers and developers in the forums. Our work contributes new knowledge about end-users' activities and behaviors in OSS user forums that the vital OSS stakeholders can leverage to improve end-user engagement in the OSS development process.

[**Hidellaarachchi2022**] Dulaji Hidellaarachchi, John Grundy, Rashina Hoda, and Ingo Mueller. Does personality impact requirements engineering activities?, 2022.

[**Hoda2021**] Rashina Hoda. Socio-technical grounded theory for software engineering. *IEEE Transactions on Software Engineering*, pages 1–1, 2021, DOI `10.1109/tse.2021.3106280`.

Abstract: Grounded Theory (GT), a sociological research method designed to study social phenomena, is increasingly being used to investigate the human and social aspects of software engineering (SE). However, being written by and for sociologists, GT is often challenging for a majority of SE researchers to understand and apply. Additionally, SE researchers attempting ad hoc adaptations of traditional GT guidelines for modern socio-technical (ST) contexts often struggle in the absence of clear and relevant guidelines to do so, resulting in poor quality studies. To overcome these research community challenges and leverage modern research opportunities, this paper presents Socio-Technical Grounded Theory (STGT) designed to ease application and achieve quality outcomes. It defines what exactly is meant by an ST research context and presents the STGT guidelines that expand GT's philosophical foundations, provide increased clarity and flexibility in its methodological steps and procedures, define possible scope and contexts of application, encourage frequent reporting of a variety of interim, preliminary, and mature outcomes, and introduce nuanced evaluation guidelines for different outcomes. It is hoped that the SE research community and related ST disciplines such as computer science, data science, artificial intelligence, information systems, human computer/robot/AI interaction, human-centered emerging technologies (and increasingly other disciplines being transformed

by rapid digitalisation and AI-based augmentation), will benefit from applying STGT to conduct quality research studies and systematically produce rich findings and mature theories with confidence.

[**Hundhausen2022**] C D Hundhausen, P T Conrad, A S Carter, and O Adesope. Assessing individual contributions to software engineering projects: a replication study. *Comput. Sci. Educ.*, 32(3):335–354, Jul 2022, DOI 10.1080/08993408.2022.2071543.
Abstract: ABSTRACT Background and Context Assessing team members' indivdiual contributions to software development projects poses a key problem for computing instructors. While instructors typically rely on subjective assessments, objective assessments could provide a more robust picture. To explore this possibility, In a 2020 paper, Buffardi presented a correlational analysis of objective metrics and subjective metrics in an advanced software engineering project course (n= 41 students and 10 teams), finding only two significant correlations. Objective To explore the robustness of Buffardi's findings and gain further insight, we conducted a larger scale replication of the Buffardi study (n = 118 students and 25 teams) in three courses at three institutions. Method We collected the same data as in the Buffardi study and computed the same measures from those data. We replicated Buffardi's exploratory, correlational and regression analyses of objective and subjective measures. Findings While replicating four of Buffardi's five significant correlational findings and partially replicating the findings of Buffardi's regression analyses, our results go beyond those of Buffardi by identifying eight additional significant correlations. Implications In contrast to Buffardi's study, our larger scale study suggests that subjective and objective measures of individual performance in team software development projects can be fruitfully combined to provide consistent and complementary assessments of individual performance.

[**Huszar2022**] Ferenc Huszár, Sofia Ira Ktena, Conor O'Brien, Luca Belli, Andrew Schlaikjer, and Moritz Hardt. Algorithmic amplification of politics on twitter. *Proc. Natl. Acad. Sci. U. S. A.*, 119(1):e2025334119, Jan 2022, DOI 10.1073/pnas.2025334119.
Abstract: Significance The role of social media in political discourse has been the topic of intense scholarly and public debate. Politicians and commentators from all sides allege that Twitter's algorithms amplify their opponents' voices, or silence theirs. Policy makers and researchers have thus called for increased transparency on how algorithms influence exposure to political content on the platform. Based on a massive-scale experiment involving millions of Twitter users, a fine-grained analysis of political parties in seven countries, and 6.2 million news articles shared in the United States, this study carries out the most comprehensive audit of an algorithmic recommender system and its effects on political content. Results unveil that the political right enjoys higher amplification compared to the political left. Content on Twitter's home timeline is selected and ordered by personalization algorithms. By consistently ranking certain content higher, these algorithms

may amplify some messages while reducing the visibility of others. There's been intense public and scholarly debate about the possibility that some political groups benefit more from algorithmic amplification than others. We provide quantitative evidence from a long-running, massive-scale randomized experiment on the Twitter platform that committed a randomized control group including nearly 2 million daily active accounts to a reverse-chronological content feed free of algorithmic personalization. We present two sets of findings. First, we studied tweets by elected legislators from major political parties in seven countries. Our results reveal a remarkably consistent trend: In six out of seven countries studied, the mainstream political right enjoys higher algorithmic amplification than the mainstream political left. Consistent with this overall trend, our second set of findings studying the US media landscape revealed that algorithmic amplification favors right-leaning news sources. We further looked at whether algorithms amplify far-left and far-right political groups more than moderate ones; contrary to prevailing public belief, we did not find evidence to support this hypothesis. We hope our findings will contribute to an evidence-based debate on the role personalization algorithms play in shaping political content consumption.

[**Idowu2022**] Samuel Idowu, Daniel Strüber, and Thorsten Berger. Asset management in machine learning: State-of-research and state-of-practice. *ACM Comput. Surv.*, Jun 2022, DOI 10.1145/3543847.
Abstract: Machine learning components are essential for today's software systems, causing a need to adapt traditional software engineering practices when developing machine-learning-based systems. This need is pronounced due to many development-related challenges of machine learning components such as asset, experiment, and dependency management. Recently, many asset management tools addressing these challenges have become available. It is essential to understand the support such tools offer to facilitate research and practice on building new management tools with native supports for machine learning and software engineering assets. This article positions machine learning asset management as a discipline that provides improved methods and tools for performing operations on machine learning assets. We present a feature-based survey of 18 state-of-practice and 12 state-of-research tools supporting machine-learning asset management. We overview their features for managing the types of assets used in machine learning experiments. Most state-of-research tools focus on tracking, exploring, and retrieving assets to address development concerns such as reproducibility, while the state-of-practice tools also offer collaboration and workflow-execution-related operations. In addition, assets are primarily tracked intrusively from the source code through APIs and managed via web dashboards or command-line interfaces. We identify asynchronous collaboration and asset reusability as directions for new tools and techniques.

[**Imam2021**] Ahmed Imam and Tapajit Dey. Tracking hackathon code creation and reuse. In *Proc. International Conference on Mining Software Repositories (MSR)*. IEEE, May 2021, DOI 10.1109/msr52588.2021.00085.

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

[**Imtiaz2022**] Nasif Imtiaz, Aniqa Khanom, and Laurie Williams. Open or sneaky? fast or slow? light or heavy?: Investigating security releases of open source packages. *IEEE Transactions on Software Engineering*, pages 1–21, 2022, DOI `10.1109/tse.2022.3181010`.
Abstract: Vulnerabilities in open source packages can be a security risk for the downstream client projects. When a new vulnerability is discovered, a package should quickly release a fix in a new version, referred to as a security release in this study. The security release should be well-documented and require minimal migration effort to facilitate fast adoption by the clients. However, to what extent the open source packages follow these recommendations is not known. In this paper, we study (1) the time lag between fix and release; (2) how security fixes are documented in the release notes; (3) code change characteristics (size and semantic versioning) of the release; and (4) the time lag between the release and an advisory publication for security releases over a dataset of 4,377 security advisories across seven package ecosystems. We find that the median security release becomes available within 4 days of the corresponding fix and contains 131 lines of code (LOC) change. However, one-fourth of the releases in our data set still came at least 20 days after the fix was made. Further, we find that 61.5% of the security releases come with a release note that documents the corresponding security fix. Still, Snyk and NVD, two popular databases, take a median of 17 days (from the release) to publish a security advisory, possibly resulting in de-

layed notifications to the client projects. We also find that security releases may contain breaking change(s) as 13.2% indicated backward incompatibility through semantic versioning, while 6.4% mentioned breaking change(s) in the release notes. Based on our findings, we point out areas for future work, such as private fork for security fixes and standardized practice for announcing security releases.

[**Jeffries2022**] Bryn Jeffries, Jung A Lee, and Irena Koprinska. 115 ways not to say hello, world! In *Proc. Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, Jul 2022, DOI 10.1145/3502718.3524809.

Abstract: Online programming courses can provide detailed automatic feedback for code that fails to meet various test conditions, but novice students often struggle with syntax errors and are unable to write valid testable code. Even for very simple exercises, the range of incorrect code can be surprising to educators with mastery of a programming language. This research paper presents an analysis of the error messages from code run by students in an introductory Python 3 programming course, participated in by 8680 primary and high-school students from 680 institutions. The invalid programs demonstrate a wide diversity of mistakes: even for a one-line "Hello World!" exercise there were 115 unique invalid programs. The most common errors are identified and compared to the topics introduced in the course. The most generic errors in selected exercises are investigated in greater detail to understand the underlying causes. While the majority of students attempting an exercise reach a successful outcome, many students encounter at least one error in their code. Of these, many such errors indicate basic mistakes, such as unquoted string literals, even in exercises late in the course for which some proficiency of earlier concepts is assumed. These observations suggest there is significant scope to provide greater reinforcement of students' understanding of earlier concepts.

[**Joblin2022**] Mitchell Joblin and Sven Apel. How do successful and failed projects differ? a socio-technical analysis. *ACM Trans. Softw. Eng. Methodol.*, Feb 2022, DOI 10.1145/3504003.

Abstract: Software development is at the intersection of the social realm , involving people who develop the software, and the technical realm , involving artifacts (code, docs, etc.) that are being produced. It has been shown that a socio-technical perspective provides rich information about the state of a software project. In particular, we are interested in socio-technical factors that are associated with project success. For this purpose, we frame the task as a network classification problem. We show how a set of heterogeneous networks composed of social and technical entities can be jointly embedded in a single vector space enabling mathematically sound comparisons between distinct software projects. Our approach is specifically designed using intuitive metrics stemming from network analysis and statistics to ease the interpretation of results in the context of software engineering wisdom. Based on a selection of 32 open-source projects, we perform an empirical study to validate

our approach considering three prediction scenarios to test the classification model's ability generalizing to: (1) randomly held-out project snapshots, (2) future project states, and (3) entirely new projects. Our results provide evidence that a socio-technical perspective is superior to a pure social or technical perspective when it comes to early indicators of future project success. To our surprise, the methodology proposed here even shows evidence of being able to generalize to entirely novel (project hold-out set) software projects reaching predication accuracies of 80%, which is a further testament to the efficacy of our approach and beyond what has been possible so far. In addition, we identify key features that are strongly associated with project success. Our results indicate that even relatively simple socio-technical networks capture highly relevant and interpretable information about the early indicators of future project success.

[**Johnson2016**] Philip Johnson, Dan Port, and Emily Hill. An athletic approach to software engineering education. In *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*. IEEE, Apr 2016, DOI `10.1109/cseet.2016.29`.

Abstract: We present our findings after two years of experience involving three instructors using an "athletic" approach to software engineering education (AthSE). Co-author Johnson developed AthSE in 2013 to address issues he experienced teaching graduate and undergraduate software engineering. Co-authors Port and Hill subsequently adapted the original approach to their own software courses. AthSE is a pedagogy in which the course is organized into a series of skills to be mastered. For each skill, students are given practice "Workouts" along with videos showing the instructor performing the Workout both correctly and quickly. Unlike traditional home-work assignments, students are advised to repeat the Workout not only until they can complete it correctly, but also as quickly as the instructor. In this experience report we investigate the following question: how can software engineering education be redesigned as an athletic endeavor, and will this provide more efficient and effective learning among students and more rapidly lead them to greater competency and confidence?

[**Kacsmar2022**] Bailey Kacsmar. Improving interactive instruction: Faculty engagement requires starting small and telling all. In *Koli Calling '22: 22nd Koli Calling International Conference on Computing Education Research*. ACM, Nov 2022, DOI `10.1145/3564721.3564739`.

Abstract: Interactive instruction, such as student-centered learning or active learning, is known to benefit student success as well as diversity in computer science. However, there is a persistent and substantial dissonance between research and practice of computer science education techniques. Current research on computer science education, while extensive, sees limited adoption beyond the original researchers. The developed educational technologies can lack sufficient detail for replication or be too specific and require extensive reworking to be employable by other instructors. Furthermore, instructors face barriers to adopting interactive techniques within their

classroom due to student reception, resources, and awareness. We argue that the advancement of computer science education, in terms of propagation and sustainability of student-centered teaching, requires guided approaches for incremental instructional changes as opposed to revolutionary pedagogy. This requires the prioritization of lightweight techniques that can fit within existing lecture formats to enable instructors to overcome barriers hindering the adoption of interactive techniques. Furthermore, such techniques and innovations must be documented in the form of computing education research artifacts, building upon the practices of software artifacts.

[**Khan2022**] Faizan Khan, Boqi Chen, Daniel Varro, and Shane McIntosh. An empirical study of type-related defects in python projects. *IEEE Transactions on Software Engineering*, 48(8):3145–3158, Aug 2022, DOI 10.1109/tse.2021.3082068.

Abstract: In recent years, Python has experienced an explosive growth in adoption, particularly among open source projects. While Python's dynamically-typed nature provides developers with powerful programming abstractions, that same dynamic type system allows for type-related defects to accumulate in code bases. To aid in the early detection of type-related defects, type annotations were introduced into the Python ecosystem (i.e., PEP-484) and static type checkers like mypy have appeared on the market. While applying a type checker like mypy can in theory help to catch type-related defects before they impact users, little is known about the real impact of adopting a type checker to reveal defects in Python projects. In this paper, we study the extent to which Python projects benefit from such type checking features. For this purpose, we mine the issue tracking and version control repositories of 210 Python projects on GitHub. Inspired by the work of Gao et al. on type-related defects in JavaScript, we add type annotations to test whether mypy detects an error that would have helped developers to avoid real defects. We observe that 15 percent of the defects could have been prevented by mypy. Moreover, we find that there is no significant difference between the experience level of developers committing type-related defects and the experience of developers committing defects that are not type-related. In addition, a manual analysis of the anti-patterns that most commonly lead to type-checking faults reveals that the redefinition of Python references, dynamic attribute initialization and incorrectly handled Null objects are the most common causes of type-related faults. Since our study is conducted on fixed public defects that have gone through code reviews and multiple test cycles, these results represent a lower bound on the benefits of adopting a type checker. Therefore, we recommend incorporating a static type checker like mypy into the development workflow, as not only will it prevent type-related defects but also mitigate certain anti-patterns during development.

[**Knutas2023**] Antti Knutas, Dominik Siemon, Natasha Tylosky, and Giovanni Maccani. Contradicting motivations in civic tech software development: Analysis of a grassroots project, 2023.

Abstract: Grassroots civic tech, or software for social change, is an emerging practice where people create and then use software to create positive change in their community. In this interpretive case study, we apply Engeström's expanded activity theory as a theoretical lens to analyze motivations, how they relate to for example group goals or development tool supported processes, and what contradictions emerge. Participants agreed on big picture motivations, such as learning new skills or improving the community. The main contradictions occurred inside activity systems on details of implementation or between system motives, instead of big picture motivations. Two most significant contradictions involved planning, and converging on design and technical approaches. These findings demonstrate the value of examining civic tech development processes as evolving activity systems.

[**Kokinda2023**] Ella Kokinda and Paige Rodeghero. Streaming software development: Accountability, community, and learning, 2023.
Abstract: People use the Internet to learn new skills, stay connected with friends, and find new communities to engage with. Live streaming platforms like Twitch.tv, YouTube Live, and Facebook Gaming provide a place where all three of these activities intersect and enable users to live-stream themselves playing a video game or live-coding software and game development, as well as the ability to participate in chat while watching someone else engage in an activity. Through fifteen interviews with software and game development streamers, we investigate why people choose to stream themselves programming and if they perceive themselves improving their programming skills by live streaming. We found that the motivations to stream included accountability, self-education, community, and visibility of the streamers' work, and streamers perceived a positive influence on their ability to write source code. Our findings implicate that alternative learning methods like live streaming programming are a beneficial tool in the age of the virtual classroom. This work also contributes to and extends research efforts surrounding educational live streaming and collaboration in developer communities.

[**Kotti2022**] Zoe Kotti, Georgios Gousios, and Diomidis Spinellis. Impact of software engineering research in practice: A patent and author survey analysis, 2022, DOI 10.1109/TSE.2022.3208210.
Abstract: Existing work on the practical impact of software engineering (SE) research examines industrial relevance rather than adoption of study results, hence the question of how results have been practically applied remains open. To answer this and investigate the outcomes of impactful research, we performed a quantitative and qualitative analysis of 4 354 SE patents citing 1 690 SE papers published in four leading SE venues between 1975–2017. Moreover, we conducted a survey on 475 authors of 593 top-cited and awarded publications, achieving 26% response rate. Overall, researchers have equipped practitioners with various tools, processes, and methods, and improved many existing products. SE practice values knowledge-seeking research and is impacted by diverse cross-disciplinary SE areas. Practitioner-

oriented publication venues appear more impactful than researcher-oriented ones, while industry-related tracks in conferences could enhance their impact. Some research works did not reach a wide footprint due to limited funding resources or unfavorable cost-benefit trade-off of the proposed solutions. The need for higher SE research funding could be corroborated through a dedicated empirical study. In general, the assessment of impact is subject to its definition. Therefore, academia and industry could jointly agree on a formal description to set a common ground for subsequent research on the topic.

[**Kreuzberger2022**] Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirschl. Machine learning operations (mlops): Overview, definition, and architecture, 2022.
Abstract: The final goal of all industrial machine learning (ML) projects is to develop ML products and rapidly bring them into production. However, it is highly challenging to automate and operationalize ML products and thus many ML endeavors fail to deliver on their expectations. The paradigm of Machine Learning Operations (MLOps) addresses this issue. MLOps includes several aspects, such as best practices, sets of concepts, and development culture. However, MLOps is still a vague term and its consequences for researchers and professionals are ambiguous. To address this gap, we conduct mixed-method research, including a literature review, a tool review, and expert interviews. As a result of these investigations, we provide an aggregated overview of the necessary principles, components, and roles, as well as the associated architecture and workflows. Furthermore, we furnish a definition of MLOps and highlight open challenges in the field. Finally, this work provides guidance for ML researchers and practitioners who want to automate and operate their ML products with a designated set of technologies.

[**Kudrjavets2022**] Gunnar Kudrjavets, Nachiappan Nagappan, and Ayushi Rastogi. Do small code changes merge faster? a multi-language empirical investigation, 2022, DOI `10.1145/3524842.3528448`.

[**Kuhrmann2022**] Marco Kuhrmann, Paolo Tell, Regina Hebig, Jil Klunder, Jurgen Munch, Oliver Linssen, Dietmar Pfahl, Michael Felderer, Christian R. Prause, Stephen G. MacDonell, Joyce Nakatumba-Nabende, David Raffo, Sarah Beecham, Eray Tuzun, Gustavo Lopez, Nicolas Paez, Diego Fontdevila, Sherlock A. Licorish, Steffen Kupper, Gunther Ruhe, Eric Knauss, Ozden Ozcan-Top, Paul Clarke, Fergal McCaffery, Marcela Genero, Aurora Vizcaino, Mario Piattini, Marcos Kalinowski, Tayana Conte, Rafael Prikladnicki, Stephan Krusche, Ahmet Coskuncay, Ezequiel Scott, Fabio Calefato, Svetlana Pimonova, Rolf-Helge Pfeiffer, Ulrik Pagh Schultz, Rogardt Heldal, Masud Fazal-Baqaie, Craig Anslow, Maleknaz Nayebi, Kurt Schneider, Stefan Sauer, Dietmar Winkler, Stefan Biffl, Maria Cecilia Bastarrica, and Ita Richardson. What makes agile software development agile? *IEEE Transactions on Software Engineering*, 48(9):3523–3539, Sep 2022, DOI `10.1109/tse.2021.3099532`.

[**Kumar2022**] Pranjay Kumar, Davin Ie, and Melina Vidoni. On the developers' attitude towards CRAN checks. In *Proc. International Conference on Program Comprehension (ICPC)*. ACM, May 2022, DOI 10.1145/3524610.3528389.

Abstract: R is a package-based, multi-paradigm programming language for scientific software. It provides an easy way to install third-party code, datasets, tests, documentation and examples through CRAN (Comprehensive R Archive Network). Prior works indicated developers tend to code workarounds to bypass CRAN's automated checks (performed when submitting a package) instead of fixing the code-doing so reduces packages' quality. It may become a threat to those analyses written in R that rely on miss-checked code. This preliminary study card-sorted source code comments and analysed StackOverflow (SO) conversations discussing CRAN checks to understand developers' attitudes. We determined that about a quarter of SO posts aim to bypass a check with a workaround; the most affected are code-related problems, package dependencies, installation and feasibility. We analyse these checks and outline future steps to improve similar automated analyses.

[**Kuttal2021**] Sandeep Kaur Kuttal, Xiaofan Chen, Zhendong Wang, Sogol Balali, and Anita Sarma. Visual resume: Exploring developers' online contributions for hiring. *Information and Software Technology*, 138:106633, Oct 2021, DOI 10.1016/j.infsof.2021.106633.

Abstract: Context: Recruiters and practitioners are increasingly relying on online activities of developers to find a suitable candidate. Past empirical studies have identified technical and soft skills that managers use in online peer production sites when making hiring decisions. However, finding candidates with relevant skills is a labor-intensive task for managers, due to the sheer amount of information online peer production sites contain. Objective: We designed a profile aggregation tool—Visual Resume—that aggregates contribution information across two types of peer production sites: a code hosting site (GitHub) and a technical Q&A forum (Stack Overflow). Visual Resume displays summaries of developers' contributions and allows easy access to their contribution details. It also facilitates pairwise comparisons of candidates through a card-based design. We present the motivation for such a design and design guidelines for creating such recruitment tool. Methods: We performed a scenario-based evaluation to identify how participants use developers' online contributions in peer production sites as well as how they used Visual Resume when making hiring decisions. Results: Our analysis helped in identifying the technical and soft skill cues that were most useful to our participants when making hiring decisions in online production sites. We also identified the information features that participants used and the ways the participants accessed that information to select a candidate. Conclusions: Our results suggest that Visual Resume helps in participants evaluate cues for technical and soft skills more efficiently as it presents an aggregated view of candidate's contributions, allows drill down to details

about contributions, and allows easy comparison of candidates via movable cards that could be arranged to match participants' needs.

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

[**Langhout2021**] Chris Langhout and Maurício Aniche. Atoms of confusion in java, 2021.

[**LawrenceDill2022**] Carolyn J Lawrence-Dill, Robyn L Allscheid, Albert Boaitey, Todd Bauman, Edward S Buckler, 4th, Jennifer L Clarke, Christopher Cullis, Jack Dekkers, Cassandra J Dorius, Shawn F Dorius, David Ertl, Matthew Homann, Guiping Hu, Mary Losch, Eric Lyons, Brenda Murdoch, Zahra-Katy Navabi, Somashekhar Punnuri, Fahad Rafiq, James M Reecy, Patrick S Schnable, Nicole M Scott, Moira Sheehan, Xavier Sirault, Margaret Staton, Christopher K Tuggle, Alison Van Eenennaam, and Rachael Voas. Ten simple rules to ruin a collaborative environment. *PLoS Comput. Biol.*, 18(4):e1009957, Apr 2022, DOI 10.1371/journal.pcbi.1009957.

[**Leelaprute2022**] Pattara Leelaprute, Bodin Chinthanet, Supatsara Wattanakriengkrai, Raula Gaikovina Kula, Pongchai Jaisri, and Takashi Ishio. Does coding in pythonic zen peak performance? preliminary experiments of nine pythonic idioms at scale, 2022.

[**Leinonen2022**] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A. Becker. Using large language models to enhance programming error messages, 2022.

[**Leite2022**] Leonardo Leite, Nelson Lago, Claudia Melo, Fabio Kon, and Paulo Meirelles. A theory of organizational structures for development and infrastructure professionals. *IEEE Transactions on Software Engineering*, pages 1–30, 2022, DOI 10.1109/tse.2022.3199169.

Abstract: DevOps and continuous delivery have impacted the organizational structures of development and infrastructure groups in software-producing organizations. Our research aims at revealing the different options adopted by the software industry to organize such groups, understanding why different organizations adopt distinct structures, and discovering how organizations handle the drawbacks of each structure. We interviewed 68 carefully-selected IT professionals, 45 working in Brazil, 10 in the USA, 8 in Europe, 1 in Canada, and 4 in globally distributed teams. By analyzing these conversations through a Grounded Theory process, we identified conditions, causes, reasons to avoid, consequences, and contingencies related to each discovered structure (segregated departments, collaborative departments, API-mediated departments, and single department). In this way, we offer a theory to explain organizational structures for development and infrastructure professionals. This theory can support practitioners and researchers in comprehending and discussing the DevOps phenomenon and its related issues, and also provides valuable input to practitioners' decision-making.

[**Leven2022**] William Levén, Hampus Broman, Terese Besker, and Richard Torkar. The broken windows theory applies to technical debt, 2022.

[**Li2019**] Chen Li, Emily Chan, Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. Towards a framework for teaching debugging. In *Proceedings of the Twenty-First Australasian Computing Education Conference on - ACE '19*. ACM Press, 2019, DOI 10.1145/3286960.3286970.

Abstract: Debugging is an important component of software development, yet most novice programmers are not explicitly taught to apply systematic strategies or processes for debugging. In this paper we adapt a framework developed for teaching troubleshooting to the debugging domain, and explore how the literature on teaching debugging maps to this framework. We identify debugging processes that are fundamental for novices to learn, aspects of debugging that novices typically struggle to develop, and shortcomings of tools designed to support teaching of debugging.

[**Li2022**] Annie Li, Madeline Endres, and Westley Weimer. Debugging with stack overflow. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Software Engineering Education and Training*. ACM, May 2022, DOI 10.1145/3510456.3514147.

Abstract: Debugging can be challenging for novice and expert programmers alike. Programmers routinely turn to online resources such as Stack Overflow for help, but understanding of debugging search practices, as well as tool support to find debugging resources, remains limited. Existing tools that mine online help forums are generally not aimed at novices, and pro-

grammers face varying levels of success when looking for online resources. Furthermore, training online code search skills is pedagogically challenging, as we have little understanding of how expertise impacts programmers' web search behavior while debugging code.We help fill these knowledge gaps with the results of a study of 40 programmers investigating differences in Stack Overflow search behavior at three levels of expertise: novices, experienced programmers who are novices in Python (the language we use in our study), and experienced Python programmers. We observe significant differences between all three levels in their ability to find posts helpful for debugging a given error, with both general and language-specific expertise facilitating Stack Overflow search efficacy and debugging success. We also conduct an exploratory investigation of factors that correlate with this difference, such as the display rank of the selected link and the number of links checked per search query. We conclude with an analysis of how online search behavior and results vary by Python error type. Our findings can inform online code search pedagogy, as well as inform the development of future automated tools.

[**Li2023**] Ze Shi Li, Nowshin Nawar Arony, Kezia Devathasan, and Daniela Damian. "software is the easy part of software engineering" – lessons and experiences from a large-scale, multi-team capstone course, 2023.
Abstract: Capstone courses in undergraduate software engineering are a critical final milestone for students. These courses allow students to create a software solution and demonstrate the knowledge they accumulated in their degrees. However, a typical capstone project team is small containing no more than 5 students and function independently from other teams. To better reflect real-world software development and meet industry demands, we introduce in this paper our novel capstone course. Each student was assigned to a large-scale, multi-team (i.e., company) of up to 20 students to collaboratively build software. Students placed in a company gained first-hand experiences with respect to multi-team coordination, integration, communication, agile, and teamwork to build a microservices based project. Furthermore, each company was required to implement plug-and-play so that their services would be compatible with another company, thereby sharing common APIs. Through developing the product in autonomous sub-teams, the students enhanced not only their technical abilities but also their soft skills such as communication and coordination. More importantly, experiencing the challenges that arose from the multi-team project trained students to realize the pitfalls and advantages of organizational culture. Among many lessons learned from this course experience, students learned the critical importance of building team trust. We provide detailed information about our course structure, lessons learned, and propose recommendations for other universities and programs. Our work concerns educators interested in launching similar capstone projects so that students in other institutions can reap the benefits of large-scale, multi-team development.

[**Liang2022**] Jenny T Liang, Thomas Zimmermann, and Denae Ford. Un-

derstanding skills for OSS communities on GitHub. In *Proc. European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, Nov 2022, DOI 10.1145/3540250.3549082.
Abstract: The development of open source software (OSS) is a broad field which requires diverse skill sets. For example, maintainers help lead the project and promote its longevity, technical writers assist with documentation, bug reporters identify defects in software, and developers program the software. However, it is unknown which skills are used in OSS development as well as OSS contributors' general attitudes towards skills in OSS. In this paper, we address this gap by administering a survey to a diverse set of 455 OSS contributors. Guided by these responses as well as prior literature on software development expertise and social factors of OSS, we develop a model of skills in OSS that considers the many contexts OSS contributors work in. This model has 45 skills in the following 9 categories: technical skills, working styles, problem solving, contribution types, project-specific skills, interpersonal skills, external relations, management, and characteristics. Through a mix of qualitative and quantitative analyses, we find that OSS contributors are actively motivated to improve skills and perceive many benefits in sharing their skills with others. We then use this analysis to derive a set of design implications and best practices for those who incorporate skills into OSS tools and platforms, such as GitHub.

[**Lin2022**] Yu-Tzu Lin, Martin K.-C. Yeh, and Sheng-Rong Tan. Teaching programming by revealing thinking process: Watching experts' live coding videos with reflection annotations. *IEEE Transactions on Education*, 65(4):617–627, Nov 2022, DOI 10.1109/te.2022.3155884.
Abstract: Contribution: Programming is a complex cognitive activity that involves both conceptual understanding and procedural skills, which is challenging for novices. To develop both program comprehension and implementation competency, this study proposed a live-coding-based instruction. Experts' live coding with think-aloud was recorded. Students then learn algorithmic planning and coding skills by observing experts' thinking and coding processes from the videos. To deploy this pedagogical strategy, a learning platform was developed to present the videos to students who could also annotate their reflection about the videos in the system to deepen their understanding of syntax and concepts of computer programming. Background: Traditional lecture-based programming instruction focuses more on the explanation of syntax and concepts but lacks revealing the dynamic and nonlinear thinking and coding process. Research Questions: This study is to explore the effects of live-coding-based instruction on students' programming knowledge, including declarative program knowledge (program comprehension) and procedural program knowledge (coding skills), and whether the instruction changes their attitude toward programming learning or not. Methodology: An empirical study was conducted with 33 high-school students who were novice programmers in one semester to explore the effective-

ness of the live-coding-based instruction and the use of the learning platform. Findings: The experiment results show that watching flowcharting (r = 0.369,p¡0.05) or coding processes (r = 0.409, p¡0.05) of experts improves coding skills. This implies that explicit depiction of algorithmic planning and coding processes are essential for building procedural programming knowledge. In addition, reflection on syntactic content of experts' programming plays an important role in programming (r = 0.511, p¡0.01). The research findings suggest that programming instruction could focus more on developing students' problem-solving abilities by demonstrating the dynamic and nonlinear programming processes and providing opportunities for students to reflect on how syntactic knowledge could be applied to programming.

[**Lin2023**] Jiahuei Lin, Mohammed Sayagh, and Ahmed E. Hassan. The co-evolution of the WordPress platform and its plugins. *ACM Transactions on Software Engineering and Methodology*, 32(1):1–24, Jan 2023, DOI 10.1145/3533700.

Abstract: One can extend the features of a software system by installing a set of additional components called plugins. WordPress, as a typical example of such plugin-based software ecosystems, is used by millions of websites and has a large number (i.e., 54,777) of available plugins. These plugin-based software ecosystems are diferent from traditional ecosystems (e.g., NPM dependencies) in a sense that there is a high coupling between a platform and its plugins, compared to traditional ecosystems for which components might not necessarily depend on each other (e.g., NPM libraries do not depend on a speciic version of NPM or a speciic version of a client software system). The high coupling between a plugin and its platform and other plugins causes incompatibility issues that occur during the co-evolution of a plugin and its platform as well as other plugins. In fact, incompatibility issues represent a major challenge when upgrading WordPress or its plugins. According to our study of the top 500 most-released WordPress plugins, we observe that incompatibility issues represent the 3r d major cause for bad releases, which are rapidly (within the next 24 hours) ixed via urgent releases. 32% of these incompatibilities are between a plugin and WordPress while 19% are between peer plugins. In this paper, we study how plugins co-evolve with the underlying platform as well as other plugins, in an efort to understand the practices that are related support such co-evolution and reduce incompatibility issues. In particular, we investigate how plugins support the latest available versions of WordPress, as well as how plugins are related to each other, and how they co-evolve. We observe that a plugin's support of new versions of WordPress with a large amount of code change is risky, as the releases which declare such support have a higher chance to be followed by an urgent release compared to ordinary releases. Although plugins support the latest WordPress version, plugin developers omit important changes such as deleting the use of removed WordPress APIs, which are removed a median of 873 days after the APIs have been removed from the source code of WordPress. Plugins introduce new releases that are made according to

a median of 5 other plugins, which we refer to as peer-triggered releases. A median of 20% of the peer-triggered releases are urgent releases that ix problems in their previous releases. The most common goal of peer-triggered releases is the ixing of incompatibility issues that a plugin detects as late as after a median of 36 days since the last release of another plugin. Our work sheds light into the co-evolution of WordPress plugins with their platform as well as peer plugins in an efort to uncover the practices of plugin evolution, so WordPress can accordingly design approaches to avoid incompatibility issues.

[**Liu2022**] Eric S. Liu, Dylan A. Lukes, and William G. Griswold. Refactoring in computational notebooks. *ACM Transactions on Software Engineering and Methodology*, Dec 2022, DOI 10.1145/3576036.
Abstract: Due to the exploratory nature of computational notebook development, a notebook can be extensively evolved even though it is small, potentially incurring substantial technical debt. Indeed, in interview studies notebook authors have attested to performing on-going tidying and big cleanups. However, many notebook authors are not trained as software developers, and environments like JupyterLab possess few features to aid notebook maintenance. As software refactoring is traditionally a critical tool for reducing technical debt, we sought to better understand the unique and growing ecology of computational notebooks by investigating the refactoring of public Jupyter notebooks. We randomly selected 15,000 Jupyter notebooks hosted on GitHub and studied 200 with meaningful commit histories. We found that notebook authors do refactor, favoring a few basic classic refactorings as well as those involving the notebook cell construct. Those with a computing background refactored differently than others, but not more so. Exploration-focused notebooks had a unique refactoring profile compared to more exposition-focused notebooks. Authors more often refactored their code as they went along, rather than deferring maintenance to big cleanups. These findings point to refactoring being intrinsic to notebook development.

[**Lorey2022**] Tobias Lorey, Paul Ralph, and Michael Felderer. Social science theories in software engineering research. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, May 2022, DOI 10.1145/3510003.3510076.
Abstract: As software engineering research becomes more concerned with the psychological, sociological and managerial aspects of software development, relevant theories from reference disciplines are in-creasingly important for understanding the field's core phenomena of interest. However, the degree to which software engineering research draws on relevant social sciences remains unclear. This study therefore investigates the use of social science theories in five influential software engineering journals over 13 years. It analyzes not only the extent of theory use but also what, how and where these theories are used. While 87 different theories are used, less than two percent of papers use a social science theory, most theories are used in only one paper,

most social sciences are ignored, and the theories are rarely tested for applicability to software engineering contexts. Ignoring relevant social science theories may (1) under-mine the community's ability to generate, elaborate and maintain a cumulative body of knowledge; and (2) lead to oversimplified mod-els of software engineering phenomena. More attention to theory is needed for software engineering to mature as a scientific discipline.

[**Lu2021**] Kuang-Chen Lu, Ben Greenman, and Shriram Krishnamurthi. Types for tables: A language design benchmark. *Art Sci. Eng. Program.*, 6(2), Nov 2021, DOI 10.22152/programming-journal.org/2022/6/8.
Abstract: Context Tables are ubiquitous formats for data. Therefore, techniques for writing correct programs over tables, and debugging incorrect ones, are vital. Our specific focus in this paper is on rich types that articulate the properties of tabular operations. We wish to study both their expressive power and diagnostic quality. Inquiry There is no "standard library" of table operations. As a result, every paper (and project) is free to use its own (sub)set of operations. This makes artifacts very difficult to compare, and it can be hard to tell whether omitted operations were left out by oversight or because they cannot actually be expressed. Furthermore, virtually no papers discuss the quality of type error feedback. Approach We combed through several existing languages and libraries to create a "standard library" of table operations. Each entry is accompanied by a detailed specification of its "type," expressed independent of (and hence not constrained by) any type language. We also studied and categorized a corpus of (student) program edits that resulted in table-related errors. We used this to generate a suite of erroneous programs. Finally, we adapted the concept of a datasheet to facilitate comparisons of different implementations. Knowledge Our benchmark creates a common ground to frame work in this area. Language designers who claim to support typed programming over tables have a clear suite against which to demonstrate their system's expressive power. Our family of errors also gives them a chance to demonstrate the quality of feedback. Researchers who improve one aspect—especially error reporting—without changing the other can demonstrate their improvement, as can those who engage in trade-offs between the two. The net result should be much better science in both expressiveness and diagnostics. We also introduce a datasheet format for presenting this knowledge in a methodical way. ubiquitous, and the expressive power of type systems keeps growing. Our benchmark and datasheet can help lead to more orderly science. It also benefits programmers trying to choose a language.

[**Luders2022**] Clara Marie Lüders, Abir Bouraffa, and Walid Maalej. Beyond duplicates. In *Proc. International Conference on Mining Software Repositories (MSR)*. ACM, May 2022, DOI 10.1145/3524842.3528457.
Abstract: Software projects use Issue Tracking Systems (ITS) like JIRA to track issues and organize the workflows around them. Issues are often interconnected via different links such as the default JIRA link types Duplicate, Relate, Block, or Subtask. While previous research has mostly focused on

analyzing and predicting duplication links, this work aims at understanding the various other link types, their prevalence, and characteristics towards a more reliable link type prediction. For this, we studied 607,208 links connecting 698,790 issues in 15 public JIRA repositories. Besides the default types, the custom types Depend, Incorporate, Split, and Cause were also common. We manually grouped all 75 link types used in the repositories into five general categories: General Relation, Duplication, Composition, Temporal/Causal, and Workflow. Comparing the structures of the corresponding graphs, we observed several trends. For instance, Duplication links tend to represent simpler issue graphs often with two components and Composition links present the highest amount of hierarchical tree structures (97.7%). Surprisingly, General Relation links have a significantly higher transitivity score than Duplication and Temporal/ Causal links. Motivated by the differences between the link types and by their popularity, we evaluated the robustness of two state-of-the-art duplicate detection approaches from the literature on the JIRA dataset. We found that current deep-learning approaches confuse between Duplication and other links in almost all repositories. On average, the classification accuracy dropped by 6% for one approach and 12% for the other. Extending the training sets with other link types seems to partly solve this issue. We discuss our findings and their implications for research and practice.

[**Lunn2021**] Stephanie Lunn, Monique Ross, Zahra Hazari, Mark Allen Weiss, Michael Georgiopoulos, and Kenneth Christensen. The impact of technical interviews, and other professional and cultural experiences on students' computing identity. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. ACM, Jun 2021, DOI `10.1145/3430665.3456362`.

Abstract: Increasingly companies assess a computing candidate's capabilities using technical interviews (TIs). Yet students struggle to code on demand, and there is already an insufficient amount of computing graduates to meet industry needs. Therefore, it is important to understand students' perceptions of TIs, and other professional experiences (e.g., computing jobs). We surveyed 740 undergraduate computing students at three universities to examine their experiences with the hiring process, as well as the impact of professional and cultural experiences (e.g., familial support) on computing identity. We considered the interactions between these experiences and social identity for groups underrepresented in computing - women, Black/African American, and Hispanic/Latinx students. Among other findings, we observed that students that did not have positive experiences with TIs had a reduced computing identity, but that facing discrimination during technical interviews had the opposite effect. Social support may play a role. Having friends in computing bolsters computing identity for Hispanic/Latinx students, as does a supportive home environment for women. Also, freelance computing jobs increase computing identity for Black/African American students. Our findings are intended to raise awareness of the best way for educators to help

diverse groups of students to succeed, and to inform them of the experiences that may influence students' engagement, resilience, and computing identity development.

[**MacNeil2022**] Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. Experiences from using code explanations generated by large language models in a web software development e-book, 2022.

[**Martin2023**] Florence Martin, Swapna Kumar, Albert D Ritzhaupt, and Drew Polly. Bichronous online learning: Award-winning online instructor practices of blending asynchronous and synchronous online modalities. *Internet High. Educ.*, 56(100879):100879, Jan 2023, DOI 10.1016/j.iheduc.2022.100879.

[**Mashey2021**] John R. Mashey. Interactions, impacts, and coincidences of the first golden age of computer architecture. *IEEE Micro*, 41(6):131–139, Nov 2021, DOI 10.1109/mm.2021.3112876.
Abstract: In their 2018 Turing Award lecture and 2019 paper, John Hennessy and David Patterson reviewed computer architecture progress since the 1960s. They projected a second golden age akin to the first, approximately 1986–1996, when new instruction set architectures, almost all reduced instruction set computers (RISCs), revolutionized the industry, eliminated most minicomputer vendors, rivaled mainframes, and began a takeover of supercomputing. The C language and derivatives came to pervade systems programming, whereas Unix derivatives came to run many servers, desktops, and smartphones. Such outcomes were not inevitable but depended on evolutionary interactions of computer architecture and languages, industry dynamics, and sometimes random coincidences.

[**Matsubara2022**] Patricia G. F. Matsubara, Igor Steinmacher, Bruno Gadelha, and Tayana Conte. The best defense is a good defense: adapting negotiation methods for tackling pressure over software project estimates, 2022, DOI 10.1145/3510455.3512775.

[**May2019**] Anna May, Johannes Wachs, and Anikó Hannák. Gender differences in participation and reward on stack overflow. *Empirical Software Engineering*, 24(4):1997–2019, Feb 2019, DOI 10.1007/s10664-019-09685-x.
Abstract: Programming is a valuable skill in the labor market, making the underrepresentation of women in computing an increasingly important issue. Online question and answer platforms serve a dual purpose in this field: they form a body of knowledge useful as a reference and learning tool, and they provide opportunities for individuals to demonstrate credible, verifiable expertise. Issues, such as male-oriented site design or overrepresentation of men among the site's elite may therefore compound the issue of women's underrepresentation in IT. In this paper we audit the differences in behavior and outcomes between men and women on Stack Overflow, the most popular of these Q&A sites. We observe significant differences in how men and

women participate in the platform and how successful they are. For example, the average woman has roughly half of the reputation points, the primary measure of success on the site, of the average man. Using an Oaxaca-Blinder decomposition, an econometric technique commonly applied to analyze differences in wages between groups, we find that most of the gap in success between men and women can be explained by differences in their activity on the site and differences in how these activities are rewarded. Specifically, 1) men give more answers than women and 2) are rewarded more for their answers on average, even when controlling for possible confounders such as tenure or buy-in to the site. Women ask more questions and gain more reward per question. We conclude with a hypothetical redesign of the site's scoring system based on these behavioral differences, cutting the reputation gap in half.

[**NandSharma2022**] Pankajeshwara Nand Sharma, Bastin Tony Roy Savarimuthu, and Nigel Stanger. Unearthing open source decision-making processes: A case study of Python enhancement proposals. *Softw. Pract. Exp.*, 52(10):2312–2346, Oct 2022, DOI `10.1002/spe.3128`.

Abstract: Good governance practices are pivotal to the success of Open Source Software (OSS) projects. However, the decision-making processes that are made available to stakeholders are at times incomplete and may remain buried and hidden in large amounts of software repository data. This work bridges this gap by unearthing enacted decision-making processes available for Python Enhancement Proposals (PEPs) from 1.54 million email messages that embody decisions made during the evolution of the Python language. This work employs a design science approach in operationalizing a framework called DeMaP miner that is used to discover hidden processes using information retrieval and information extraction techniques. It also uses process mining techniques to visualize the processes, and comparative structural analysis techniques to compare different decision processes. The work identifies a richer set of decision-making activities than those reported on the Python website and in prior research work (48 new decision activities, 199 new pathways and 6 new stages). The extracted decision process has been positively evaluated by a prominent member of the Python steering council. The extracted process can be used for process compliance checking and process improvement in OSS communities. Additionally, the DeMaP Miner framework can be extended and customized to suit other OSS projects, such as the OpenJDK project.

[**Nguyen2022**] Nhan Nguyen and Sarah Nadi. An empirical evaluation of GitHub copilot's code suggestions. In *Proc. International Conference on Mining Software Repositories (MSR)*. ACM, May 2022, DOI `10.1145/3524842.3528470`.

Abstract: GitHub and OpenAI recently launched Copilot, an "AI pair programmer" that utilizes the power of Natural Language Processing, Static Analysis, Code Synthesis, and Artificial Intelligence. Given a natural language description of the target functionality, Copilot can generate corre-

sponding code in several programming languages. In this paper, we perform an empirical study to evaluate the correctness and understandability of Copilot's suggested code. We use 33 LeetCode questions to create queries for Copilot in four different programming languages. We evaluate the correctness of the corresponding 132 Copilot solutions by running LeetCode's provided tests, and evaluate understandability using SonarQube's cyclomatic complexity and cognitive complexity metrics. We find that Copilot's Java suggestions have the highest correctness score (57%) while JavaScript is the lowest (27%). Overall, Copilot's suggestions have low complexity with no notable differences between the programming languages. We also find some potential Copilot shortcomings, such as generating code that can be further simplified and code that relies on undefined helper methods.

[**Nicacio2022**] Jalves Nicacio and Fabio Petrillo. An approach to build consistent software architecture diagrams using devops system descriptors, 2022, DOI 10.1145/3550356.3561567.

[**Noble2022**] James Noble, David Streader, Isaac Oscar Gariano, and Miniruwani Samarakoon. More programming than programming: Teaching formal methods in a software engineering programme, 2022.

[**Olejniczak2020**] Anthony J. Olejniczak and Molly J. Wilson. Who's writing open access (OA) articles? characteristics of OA authors at ph.d.-granting institutions in the united states. *Quantitative Science Studies*, 1(4):1429–1450, Dec 2020, DOI 10.1162/qss_a_00091.

Abstract: The open access (OA) publication movement aims to present research literature to the public at no cost and with no restrictions. While the democratization of access to scholarly literature is a primary focus of the movement, it remains unclear whether OA has uniformly democratized the corpus of freely available research, or whether authors who choose to publish in OA venues represent a particular subset of scholars—those with access to resources enabling them to afford article processing charges (APCs). We investigated the number of OA articles with article processing charges (APC OA) authored by 182,320 scholars with known demographic and institutional characteristics at American research universities across 11 broad fields of study. The results show, in general, that the likelihood for a scholar to author an APC OA article increases with male gender, employment at a prestigious institution (AAU member universities), association with a STEM discipline, greater federal research funding, and more advanced career stage (i.e., higher professorial rank). Participation in APC OA publishing appears to be skewed toward scholars with greater access to resources and job security.

[**Palomba2021**] Fabio Palomba, Damian Andrew Tamburri, Francesca Arcelli Fontana, Rocco Oliveto, Andy Zaidman, and Alexander Serebrenik. Beyond technical aspects: How do community smells influence the intensity of code smells? *IEEE Transactions on Software Engineering*, 47(1):108–129, Jan 2021, DOI 10.1109/tse.2018.2883603.

Abstract: Code smells are poor implementation choices applied by developers during software evolution that often lead to critical flaws or failure. Much in the same way, community smells reflect the presence of organizational and socio-technical issues within a software community that may lead to additional project costs. Recent empirical studies provide evidence that community smells are often—if not always—connected to circumstances such as code smells. In this paper we look deeper into this connection by conducting a mixed-methods empirical study of 117 releases from 9 open-source systems. The qualitative and quantitative sides of our mixed-methods study were run in parallel and assume a mutually-confirmative connotation. On the one hand, we survey 162 developers of the 9 considered systems to investigate whether developers perceive relationship between community smells and the code smells found in those projects. On the other hand, we perform a fine-grained analysis into the 117 releases of our dataset to measure the extent to which community smells impact code smell intensity (i.e., criticality). We then propose a code smell intensity prediction model that relies on both technical and community-related aspects. The results of both sides of our mixed-methods study lead to one conclusion: community-related factors contribute to the intensity of code smells. This conclusion supports the joint use of community and code smells detection as a mechanism for the joint management of technical and social problems around software development communities.

[**Panthaplackel2022**] Sheena Panthaplackel, Milos Gligoric, Junyi Jessy Li, and Raymond J. Mooney. Using developer discussions to guide fixing bugs in software, 2022.

Abstract: Automatically fixing software bugs is a challenging task. While recent work showed that natural language context is useful in guiding bug-fixing models, the approach required prompting developers to provide this context, which was simulated through commit messages written after the bug-fixing code changes were made. We instead propose using bug report discussions, which are available before the task is performed and are also naturally occurring, avoiding the need for any additional information from developers. For this, we augment standard bug-fixing datasets with bug report discussions. Using these newly compiled datasets, we demonstrate that various forms of natural language context derived from such discussions can aid bug-fixing, even leading to improved performance over using commit messages corresponding to the oracle bug-fixing commits.

[**PapapanagiotakisBousy2022**] Iason Papapanagiotakis Bousy, Earl T. Barr, and David Clark. PopArt: Ranked testing efficiency. *IEEE Transactions on Software Engineering*, pages 1–18, 2022, DOI `10.1109/tse.2022.3214796`.

Abstract: Too often, programmers are under pressure to maximize their confidence in the correctness of their code with a tight testing budget. Should they spend some of that budget on finding "interesting" inputs or spend their entire testing budget on test executions? Work on testing efficiency

has explored two competing approaches to answer this question: systematic partition testing (ST), which defines a testing partition and tests its parts, and random testing (RT), which directly samples inputs with replacement. A consensus as to which is better when has yet to emerge. We present Probability Ordered Partition Testing (POPART), a new systematic partition-based testing strategy that visits the parts of a testing partition in decreasing probability order and in doing so leverages any non-uniformity over that partition. We show how to construct a homogeneous testing partition, a requirement for systematic testing, by using an executable oracle and the path partition. A program's path partition is a naturally occurring testing partition that is usually skewed for the simple reason that some paths execute more frequently than others. To confirm this conventional wisdom, we instrument programs from the Codeflaws repository and find that 80% of them have a skewed path probability distribution. POPART visits the parts of a testing partition in decreasing probability order. We then compare POPART with RT to characterise the configuration space in which each is more efficient. We show that, when simulating Codeflaws, POPART outperforms RT after 100,000 executions. Our results reaffirm RT's power for very small testing budgets but also show that for any application requiring high (above 90%) probability-weighted coverage POPART should be preferred. In such cases, despite paying more for each test execution, we prove that POPART outperforms RT: it traverses parts whose cumulative probability bounds that of random testing, showing that sampling without replacement pays for itself, given a nonuniform probability over a testing partition.

[**Pashchenko2022**] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vuln4real: A methodology for counting actually vulnerable dependencies. *IEEE Transactions on Software Engineering*, 48(5):1592–1609, May 2022, DOI `10.1109/tse.2020.3025443`.
Abstract: Vulnerable dependencies are a known problem in today's free open-source software ecosystems because FOSS libraries are highly interconnected, and developers do not always update their dependencies. Our paper proposes Vuln4Real, the methodology for counting actually vulnerable dependencies, that addresses the over-inflation problem of academic and industrial approaches for reporting vulnerable dependencies in FOSS software, and therefore, caters to the needs of industrial practice for correct allocation of development and audit resources. To understand the industrial impact of a more precise methodology, we considered the 500 most popular FOSS Java libraries used by SAP in its own software. Our analysis included 25767 distinct library instances in Maven. We found that the proposed methodology has visible impacts on both ecosystem view and the individual library developer view of the situation of software dependencies: Vuln4Real significantly reduces the number of false alerts for deployed code (dependencies wrongly flagged as vulnerable), provides meaningful insights on the exposure to third-parties (and hence vulnerabilities) of a library, and automatically predicts when dependency maintenance starts lagging, so it

may not receive updates for arising issues.

[**Pelanek2022**] Radek Pelánek and Tomáš Effenberger. The landscape of computational thinking problems for practice and assessment. *ACM Transactions on Computing Education*, Dec 2022, DOI 10.1145/3578269.
Abstract: To provide practice and assessment of computational thinking, we need specific problems students can solve. There are many such problems, but they are hard to find. Learning environments and assessments often use only specific types of problems and thus do not cover computational thinking in its whole scope. We provide an extensive catalog of well-structured computational thinking problem sets together with a systematic encoding of their features. Based on this encoding, we propose a four-level taxonomy that provides an organization of a wide variety of problems. The catalog, taxonomy, and problem features are useful for content authors, designers of learning environments, and researchers studying computational thinking.

[**Pereira2017**] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, Oct 2017, DOI 10.1145/3136014.3136031.
Abstract: This paper presents a study of the runtime, memory usage and energy consumption of twenty seven well-known software languages. We monitor the performance of such languages using ten different programming problems, expressed in each of the languages. Our results show interesting findings, such as, slower/faster languages consuming less/more energy, and how memory usage influences energy consumption. Finally, we show how to use our results to provide software engineers support to decide which language to use when energy efficiency is a concern.

[**Pinckney2022**] Donald Pinckney, Federico Cassano, Arjun Guha, Jon Bell, Massimiliano Culpo, and Todd Gamblin. Flexible and optimal dependency management via max-smt, 2022.

[**Pinto2022**] Gustavo Pinto and Alberto de Souza. Cognitive-driven development helps software teams to keep code units under the limit!, 2022.

[**Pizard2022**] Sebastián Pizard, Diego Vallespir, and Barbara Kitchenham. A longitudinal case study on the effects of an evidence-based software engineering training, 2022, DOI 10.1145/3510456.3514150.

[**Poulos2021**] Alexandra Poulos, Sally A. McKee, and Jon C. Calhoun. Posits and the state of numerical representations in the age of exascale and edge computing. *Software: Practice and Experience*, 52(2):619–635, Sep 2021, DOI 10.1002/spe.3022.
Abstract: Growing constraints on memory utilization, power consumption, and I/O throughput have increasingly become limiting factors to the advancement of high performance computing (HPC) and edge computing

applications. IEEE-754 floating-point types have been the de facto standard for floating-point number systems for decades, but the drawbacks of this numerical representa- tion leave much to be desired. Alternative representations are gaining traction, both in HPC and machine learning environments. Posits have recently been proposed as a drop-in replacement for the IEEE-754 floating-point representa- tion. We survey the state-of-the-art and state-of-the-practice in the development and use of posits in edge computing and HPC. The current literature supports posits as a promising alternative to traditional floating-point systems, both as a stand-alone replacement and in a mixed-precision environment. Development and standardization of the posit type is ongoing, and much research remains to explore the application of posits in different domains, how to best implement them in hardware, and where they fit with other numerical representations.

[**Prana2019**] Gede Artha Azriadi Prana, Christoph Treude, Ferdian Thung, Thushari Atapattu, and David Lo. Categorizing the content of GitHub README files. *Empir. Softw. Eng.*, 24(3):1296–1327, Jun 2019, DOI `10.1007/s10664-018-9660-3`.

[**PreslerMarshall2022a**] Kai Presler-Marshall, Sarah Heckman, and Kathryn T Stolee. Identifying struggling teams in software engineering courses through weekly surveys. In *Proc. Technical Symposium on Computer Science Education (SIGCSE)*. ACM, Feb 2022, DOI `10.1145/3478431.3499367`.

Abstract: Teaming is increasingly a core aspect of professional software engineering and most undergraduate computer science curricula. At NC State University, we teach communication and project-management skills explicitly through a junior-level software engineering course. However, some students may have a dysfunctional team experience that imperils their ability to learn these skills. Identifying these teams during a team project is important so the teaching staff can intervene early and hopefully alleviate the issues. We propose a weekly reflection survey to help the course teaching staff proactively identify teams that may not be on track to learn the course outcomes. The questions on the survey focus on team communication and collaboration over the previous week. We evaluate our survey on two semesters of the undergraduate software engineering course by comparing teams with poor end-of-project grades or peer evaluations against teams flagged on a weekly basis through the surveys. We find that the survey can identify most teams that later struggled on the project, typically by the half-way mark of the project, and thus may provide instructors with an actionable early-warning about struggling teams. Furthermore, a majority of students (64.4%) found the survey to be a helpful tool for keeping their team on track. Finally, we discuss future work for improving the survey and engaging with student teams.

[**PreslerMarshall2022b**] Kai Presler-Marshall, Sarah Heckman, and Kathryn T Stolee. What makes team[s] work? a study of team char-

acteristics in software engineering projects. In *Proc. Conference on International Computing Education Research (ICER)*. ACM, Aug 2022, DOI 10.1145/3501385.3543980.
Abstract: Teaming is a core component in practically all professional software engineering careers, and as such, is a key skill taught in many undergraduate Computer Science programs. However, not all teams manage to work together effectively, and in education, this can deprive some students of successful teaming experiences. In this work, we seek to gain insights into the characteristics of successful and unsuccessful undergraduate student teams in a software engineering course. We conduct semi-structured interviews with 18 students who have recently completed a team-based software engineering course to understand how they worked together, what challenges they faced, and how they tried to overcome these challenges. Our results show that common problems include communicating, setting and holding to deadlines, and effectively identifying tasks and their relative difficulty. Additionally, we find that self-reflection on what is working and not working or external motivators such as grades help some, but not all, teams overcome these challenges. Finally, we conclude with recommendations for educators on successful behaviours to steer teams towards, and recommendations for researchers on future work to better understand challenges that teams face.

[**Qamar2022**] Khushbakht Ali Qamar, Emre Sülün, and Eray Tüzün. Taxonomy of bug tracking process smells: Perceptions of practitioners and an empirical analysis. *Inf. Softw. Technol.*, 150(106972):106972, Oct 2022, DOI 10.1016/j.infsof.2022.106972.

[**Queiroz2022**] Francisco Queiroz, Maria Lonsdale, and Rejane Spitz. Science as a game: conceptual model and application in scientific software design. *Int. j. des. creat. innov.*, 10(4):222–246, Oct 2022, DOI 10.1080/21650349.2022.2088623.
Abstract: Scientific inquiry is often described as, and compared to, a game. This paper expands on that analogy to propose a conceptual model of scientific practice built upon Jesper Juul's game definition, and informed by parallels between the two activities collected from selected works from history and philosophy of science. Moreover, the paper presents a design method, based on the model described, for fostering creative solutions in scientific software user interface design. Results from pilot case studies suggest both model and method are helpful, allowing participants to describe requirements and ideate solutions, as well providing a framework for the exploration of the game-science analogy within the context of scientific research conducted through computational resources.

[**Ragkhitwetsagul2022**] Chaiyong Ragkhitwetsagul and Matheus Paixao. Recommending code improvements based on stack overflow answer edits, 2022.

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

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

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

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

the responded 94 bug reports, which suggests the relevance of security smells for Ansible and Chef scripts amongst practitioners. Conclusion: We observe security smells to be prevalent in Ansible and Chef scripts, similarly to that of the Puppet scripts. We recommend practitioners to rigorously inspect the presence of the identified security smells in Ansible and Chef scripts using (i) code review, and (ii) static analysis tools.

[**Rahman2022**] Mohammad M. Rahman, Foutse Khomh, and Marco Castelluccio. Works for me! cannot reproduce – a large scale empirical study of non-reproducible bugs. *Empirical Software Engineering*, 27(5), May 2022, DOI 10.1007/s10664-022-10153-2.
Abstract: Software developers attempt to reproduce software bugs to understand their erroneous behaviours and to fix them. Unfortunately, they often fail to reproduce (or fix) them, which leads to faulty, unreliable software systems. However, to date, only a little research has been done to better understand what makes the software bugs non-reproducible. In this article, we conduct a multi-modal study to better understand the non-reproducibility of software bugs. First, we perform an empirical study using 576 non-reproducible bug reports from two popular software systems (Firefox, Eclipse) and identify 11 key factors that might lead a reported bug to non-reproducibility. Second, we conduct a user study involving 13 professional developers where we investigate how the developers cope with non-reproducible bugs. We found that they either close these bugs or solicit for further information, which involves long deliberations and counter-productive manual searches. Third, we offer several actionable insights on how to avoid non-reproducibility (e.g., false-positive bug report detector) and improve reproducibility of the reported bugs (e.g., sandbox for bug reproduction) by combining our analyses from multiple studies (e.g., empirical study, developer study). Fourth, we explain the differences between reproducible and non-reproducible bug reports by systematically interpreting multiple machine learning models that classify these reports with high accuracy. We found that links to existing bug reports might help improve the reproducibility of a reported bug. Finally, we detect the connected bug reports to a non-reproducible bug automatically and further demonstrate how 93 bugs connected to 71 non-reproducible bugs from our dataset can offer complementary information (e.g., attachments, screenshots, program flows).

[**Rao2022**] Nikitha Rao, Jason Tsay, Martin Hirzel, and Vincent J. Hellendoorn. Comments on comments: Where code review and documentation meet, 2022, DOI 10.1145/3524842.3528475.

[**Ritschel2022a**] Nico Ritschel, Vladimir Kovalenko, Reid Holmes, Ronald Garcia, and David C. Shepherd. Comparing block-based programming models for two-armed robots. *IEEE Transactions on Software Engineering*, 48(5):1630–1643, May 2022, DOI 10.1109/tse.2020.3027255.
Abstract: Modern industrial robots can work alongside human workers and

coordinate with other robots. This means they can perform complex tasks, but doing so requires complex programming. Therefore, robots are typically programmed by experts, but there are not enough to meet the growing demand for robots. To reduce the need for experts, researchers have tried to make robot programming accessible to factory workers without programming experience. However, none of that previous work supports coordinating multiple robot arms that work on the same task. In this paper we present four block-based programming language designs that enable end-users to program two-armed robots. We analyze the benefits and trade-offs of each design on expressiveness and user cognition, and evaluate the designs based on a survey of 273 professional participants of whom 110 had no previous programming experience. We further present an interactive experiment based on a prototype implementation of the design we deem best. This experiment confirmed that novices can successfully use our prototype to complete realistic robotics tasks. This work contributes to making coordinated programming of robots accessible to end-users. It further explores how visual programming elements can make traditionally challenging programming tasks more beginner-friendly.

[**Ritschel2022b**] Nico Ritschel, Anand Ashok Sawant, David Weintrop, Reid Holmes, Alberto Bacchelli, Ronald Garcia, Chandrika K R, Avijit Mandal, Patrick Francis, and David C. Shepherd. Training industrial end-user programmers with interactive tutorials. *Software: Practice and Experience*, Nov 2022, DOI `10.1002/spe.3167`.

Abstract: Newly released robot programming tools have made it feasible for end-users to program industrial robots by combining block-based languages and lead-through programming. To use these systems effectively, end-users, who usually have limited or no programming experience, require training. To train users, tutoring systems are often used for block-based programming—some even for lead-through programming—but no tutorial system combines these two types of programming. We present CoBlox Interactive Tutorials (CITs), a novel tutoring approach that teaches how to use both the hardware and software components that comprise a typical end-user robot programming environment. As users switch between the two programming styles, CITs provide them with extensive scaffolding, give users immediate feedback on missteps

[**Rombaut2023**] Benjamin Rombaut, Filipe R. Cogo, Bram Adams, and Ahmed E. Hassan. There's no such thing as a free lunch: Lessons learned from exploring the overhead introduced by the greenkeeper dependency bot in npm. *ACM Transactions on Software Engineering and Methodology*, 32(1):1–40, Jan 2023, DOI `10.1145/3522587`.

Abstract: Dependency management bots are increasingly being used to support the software development process, for example to automatically update a dependency when a new version is available. Yet, human intervention is often required to either accept or reject any action or recommendation the

bot creates. In this paper, our objective is to study the extent to which dependency management bots create additional, and sometimes unnecessary, work for their users. To accomplish this, we analyze 93,196 issue reports opened by Greenkeeper, a popular dependency management bot used in open source software projects in the npm ecosystem. We ind that Greenkeeper is responsible for half of all issues reported in client projects, inducing a signiicant amount of overhead that must be addressed by clients, since many of these issues were created as a result of Greenkeeper taking incorrect action on a dependency update (i.e., false alarms). Reverting a broken dependency update to an older version, which is a potential solution that requires the least overhead and is automatically attempted by Greenkeeper, turns out to not be an efective mechanism. Finally, we observe that 56% of the commits referenced by Greenkeeper issue reports only change the client's dependency speciication ile to resolve the issue. Based on our indings, we argue that dependency management bots should (i) be conigurable to allow clients to reduce the amount of generated activity by the bots, (ii) take into consideration more sources of information than only the pass/fail status of the client's build pipeline to help eliminate false alarms, and (iii) provide more efective incentives to encourage clients to resolve dependency issues.

[**Rule2019**] Adam Rule, Amanda Birmingham, Cristal Zuniga, Ilkay Altintas, Shih-Cheng Huang, Rob Knight, Niema Moshiri, Mai H Nguyen, Sara Brin Rosenthal, Fernando Pérez, and Peter W Rose. Ten simple rules for writing and sharing computational analyses in jupyter notebooks. *PLoS Comput. Biol.*, 15(7):e1007007, Jul 2019, DOI `10.1371/journal.pcbi.1007007`.

[**Sanders2019**] Kate Sanders, Judy Sheard, Brett A Becker, Anna Eckerdal, Sally Hamouda, and Simon. Inferential statistics in computing education research. In *Proc. Conference on International Computing Education Research (ICER)*. ACM, Jul 2019, DOI `10.1145/3291279.3339408`.
Abstract: The goal of most computing education research is to effect positive change in how computing is taught and learned. Statistical techniques are one important tool for achieving this goal. In this paper we report on an analysis of ICER papers that use inferential statistics. We present the most commonly used techniques; an overview of the techniques the ICER community has used over its first 14 years of papers, grouped according to the purpose of the technique; and a detailed analysis of three of the most commonly used techniques (t-test, chi-squared test, and Mann-Whitney-Wilcoxon). We identify common flaws in reporting and give examples of papers where statistics are reported well. In sum, the paper draws a picture of the use of inferential statistics by the ICER community. This picture is intended to help orient researchers who are new to the use of statistics in computing education research and to encourage reflection by the ICER community on how it uses statistics and how it can improve that use.

[**Schmitt2022**] Paul Schmitt, Jana Iyengar, Christopher Wood, and Barath Raghavan. The decoupling principle. In *Proceedings of the 21st*

ACM Workshop on Hot Topics in Networks. ACM, Nov 2022, DOI 10.1145/3563766.3564112.

Abstract: The three decade struggle to ensure Internet data confidentiality—a key aspect of communications privacy—is finally behind us. Encryption is fast, secure, and standard in all browsers, modern transports, and major protocols. Yet it has long seemed that network privacy is not unified by core principles but a grab bag of techniques and ideas applied to an equally wide range of applications, contexts, layers of infrastructure, and software stacks. Here we attempt to distill a principle—one that is old but seldom discussed as such—for building privacy into Internet services. We explore what privacy properties are desirable and achievable when we apply this principle. We evaluate several classic systems and ones that have been recently deployed with this principle applied, and discuss future directions for network privacy building upon these efforts.

[**Schroder2021**] Michael Schröder and Jürgen Cito. An empirical investigation of command-line customization. *Empirical Software Engineering*, 27(2), Dec 2021, DOI 10.1007/s10664-021-10036-y.

Abstract: The interactive command line, also known as the shell, is a prominent mechanism used extensively by a wide range of software professionals (engineers, system administrators, data scientists, etc.). Shell customizations can therefore provide insight into the tasks they repeatedly perform, how well the standard environment supports those tasks, and ways in which the environment could be productively extended or modified. To characterize the patterns and complexities of command-line customization, we mined the collective knowledge of command-line users by analyzing more than 2.2 million shell alias definitions found on GitHub. Shell aliases allow command-line users to customize their environment by defining arbitrarily complex command substitutions. Using inductive coding methods, we found three types of aliases that each enable a number of customization practices: Shortcuts (for nicknaming commands, abbreviating subcommands, and bookmarking locations), Modifications (for substituting commands, overriding defaults, colorizing output, and elevating privilege), and Scripts (for transforming data and chaining subcommands). We conjecture that identifying common customization practices can point to particular usability issues within command-line programs, and that a deeper understanding of these practices can support researchers and tool developers in designing better user experiences. In addition to our analysis, we provide an extensive reproducibility package in the form of a curated dataset together with well-documented computational notebooks enabling further knowledge discovery and a basis for learning approaches to improve command-line workflows.

[**Schurhoff2022**] Christian Schürhoff, Stefan Hanenberg, and Volker Gruhn. An empirical study on a single company's cost estimations of 338 software projects. *Empirical Software Engineering*, 28(1), Nov 2022, DOI 10.1007/s10664-022-10245-z.

[**Shan2023**] Shawn Shan, Jenna Cryan, Emily Wenger, Haitao Zheng, Rana Hanocka, and Ben Y. Zhao. Glaze: Protecting artists from style mimicry by text-to-image models, 2023.

Abstract: Recent text-to-image diffusion models such as MidJourney and Stable Diffusion threaten to displace many in the professional artist community. In particular, models can learn to mimic the artistic style of specific artists after "fine-tuning" on samples of their art. In this paper, we describe the design, implementation and evaluation of Glaze, a tool that enables artists to apply "style cloaks" to their art before sharing online. These cloaks apply barely perceptible perturbations to images, and when used as training data, mislead generative models that try to mimic a specific artist. In coordination with the professional artist community, we deploy user studies to more than 1000 artists, assessing their views of AI art, as well as the efficacy of our tool, its usability and tolerability of perturbations, and robustness across different scenarios and against adaptive countermeasures. Both surveyed artists and empirical CLIP-based scores show that even at low perturbation levels (p=0.05), Glaze is highly successful at disrupting mimicry under normal conditions (¿92%) and against adaptive countermeasures (¿85%).

[**Shankar2022**] Shreya Shankar, Rolando Garcia, Joseph M. Hellerstein, and Aditya G. Parameswaran. Operationalizing machine learning: An interview study, 2022.

[**Sharafi2022**] Zohreh Sharafi, Ian Bertram, Michael Flanagan, and Westley Weimer. Eyes on code: A study on developers' code navigation strategies. *IEEE Transactions on Software Engineering*, 48(5):1692–1704, May 2022, DOI `10.1109/tse.2020.3032064`.

Abstract: What code navigation strategies do developers use and what mechanisms do they employ to find relevant information? Do their strategies evolve over the course of longer tasks? Answers to these questions can provide insight to educators and software tool designers to support a wide variety of programmers as they tackle increasingly-complex software systems. However, little research to date has measured developers' code navigation strategies in ecologically-valid settings, or analyzed how strategies progressed throughout a maintenance task. We propose a novel experimental design that more accurately represents the software maintenance process in terms of software complexity and IDE interactions. Using this framework, we conduct an eye-tracking study (n=36) of realistic bug-fixing tasks, dynamically and empirically identifying relevant code areas. We introduce a three-phase model to characterize developers' navigation behavior supported by statistical variations in eye movements over time. We also propose quantifiable notion of "thrashing" with the code as a navigation activity. We find that thrashing is associated with lower effectiveness. Our results confirm that the relevance of various code elements changes over time, and that our proposed three-phase model is capable of capturing these significant changes. We dis-

cuss our findings and their implications for tool designers, educators, and the research community.

[**Shimada2022**] Naomichi Shimada, Tao Xiao, Hideaki Hata, Christoph Treude, and Kenichi Matsumoto. Github sponsors: Exploring a new way to contribute to open source, 2022, DOI 10.1145/3510003.3510116.

[**Shome2022**] Arumoy Shome, Luís Cruz, and Arie van Deursen. Data smells in public datasets. In *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI*. ACM, May 2022, DOI 10.1145/3522664.3528621.
Abstract: The adoption of Artificial Intelligence (AI) in high-stakes domains such as healthcare, wildlife preservation, autonomous driving and criminal justice system calls for a data-centric approach to AI. Data scientists spend the majority of their time studying and wrangling the data, yet tools to aid them with data analysis are lacking. This study identifies the recurrent data quality issues in public datasets. Analogous to code smells, we introduce a novel catalogue of data smells that can be used to indicate early signs of problems or technical debt in machine learning systems. To understand the prevalence of data quality issues in datasets, we analyse 25 public datasets and identify 14 data smells.

[**Shreeve2021**] Benjamin Shreeve, Joseph Hallett, Matthew Edwards, Kopo M. Ramokapane, Richard Atkins, and Awais Rashid. The best laid plans or lack thereof: Security decision-making of different stakeholder groups, 2021, DOI 10.1109/TSE.2020.3023735.
Abstract: Cyber security requirements are influenced by the priorities and decisions of a range of stakeholders. Board members and CISOs determine strategic priorities. Managers have responsibility for resource allocation and project management. Legal professionals concern themselves with regulatory compliance. Little is understood about how the security decision-making approaches of these different stakeholders contrast, and if particular groups of stakeholders have a better appreciation of security requirements during decision-making. Are risk analysts better decision makers than CISOs? Do security experts exhibit more effective strategies than board members? This paper explores the effect that different experience and diversity of expertise has on the quality of a team's cyber security decision-making and whether teams with members from more varied backgrounds perform better than those with more focused, homogeneous skill sets. Using data from 208 sessions and 948 players of a tabletop game run in the wild by a major national organization over 16 months, we explore how choices are affected by player background (e.g., cyber security experts versus risk analysts, board-level decision makers versus technical experts) and different team make-ups (homogeneous teams of security experts versus various mixes). We find that no group of experts makes significantly better game decisions than anyone else, and that their biases lead them to not fully comprehend what they are defending or how the defenses work.

[**Shrestha2021**] Nischal Shrestha, Titus Barik, and Chris Parnin. Unravel: A fluent code explorer for data wrangling. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. ACM, Oct 2021, DOI 10.1145/3472749.3474744.

Abstract: Data scientists have adopted a popular design pattern in programming called the fluent interface for composing data wrangling code. The fluent interface works by combining multiple transformations on a data table—or dataframes—with a single chain of expressions, which produces an output. Although fluent code promotes legibility, the intermediate dataframes are lost, forcing data scientists to unravel the chain through tedious code edits and re-execution. Existing tools for data scientists do not allow easy exploration or support understanding of fluent code. To address this gap, we designed a tool called Unravel that enables structural edits via drag-and-drop and toggle switch interactions to help data scientists explore and understand fluent code. Data scientists can apply simple structural edits via drag-and-drop and toggle switch interactions to reorder and (un)comment lines. To help data scientists understand fluent code, Unravel provides function summaries and always-on visualizations highlighting important changes to a dataframe. We discuss the design motivations behind Unravel and how it helps understand and explore fluent code. In a first-use study with 14 data scientists, we found that Unravel facilitated diverse activities such as validating assumptions about the code or data, exploring alternatives, and revealing function behavior.

[**Silva2022**] Yasin N Silva, Alexis Loza, and Humberto Razente. DBSnap-eval. In *Proc. Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, Jul 2022, DOI 10.1145/3502718.3524822.

Abstract: Learning to construct database queries can be a challenging task because students need to learn the specific query language syntax as well as properly understand the effect of each query operator and how multiple operators interact in a query. While some previous studies have looked into the types of database query errors students make and how the availability of expected query results can help to increase the success rate, there is very little that is known regarding the patterns that emerge while students are constructing a query. To be able to look into the process of constructing a query, in this paper we introduce DBSnap-Eval, a tool that supports tree-based queries (similar to SQL query plans) and a block-based querying interface to help separate the syntax and semantics of a query. DBSnap-Eval closely monitors the actions students take to construct a query such as adding a dataset or connecting a dataset with an operator. This paper presents an initial set of results about database query construction patterns using DBSnap-Eval. Particularly, it reports identified patterns in the process students follow to answer common database queries.

[**Soltani2020**] Mozhan Soltani, Felienne Hermans, and Thomas Bäck. The significance of bug report elements. *Empir. Softw. Eng.*, 25(6):5255–5294, Nov 2020, DOI 10.1007/s10664-020-09882-z.

Abstract: AbstractOpen source software projects often use issue repositories, where project contributors submit bug reports. Using these repositories, more bugs in software projects may be identified and fixed. However, the content and therefore quality of bug reports vary. In this study, we aim to understand the significance of different elements in bug reports. We interviewed 35 developers to gain insights into their perceptions on the importance of various contents in bug reports. To assess our findings, we surveyed 305 developers. The results show developers find it highly important that bug reports include crash description, reproducing steps or test cases, and stack traces. Software version, fix suggestions, code snippets, and attached contents have lower importance for software debugging. Furthermore, to evaluate the quality of currently available bug reports, we mined issue repositories of 250 most popular projects on Github. Statistical analysis on the mined issues shows that crash reproducing steps, stack traces, fix suggestions, and user contents, have statistically significant impact on bug resolution times, for $\tilde{7}0\%$, $\tilde{7}6\%$, $\tilde{5}5\%$, and $\tilde{3}3\%$ of the projects. However, on avarage, over 70% of bug reports lack these elements.

[**Spinellis2023**] Diomidis Spinellis. Open reproducible publication research, 2023.
Abstract: Considerable scientific work involves locating, analyzing, systematizing, and synthesizing other publications. Its results end up in a paper's "background" section or in standalone articles, which include meta-analyses and systematic literature reviews. The required research is aided through the use of online scientific publication databases and search engines, such as Web of Science, Scopus, and Google Scholar. However, use of online databases suffers from a lack of repeatability and transparency, as well as from technical restrictions. Thankfully, open data, powerful personal computers, and open source software now make it possible to run sophisticated publication studies on the desktop in a self-contained environment that peers can readily reproduce. Here we report a Python software package and an associated command-line tool that can populate embedded relational databases with slices from the complete set of Crossref publication metadata,1 ORCID author records,2 and other open data sets, for in-depth processing through performant queries. We demonstrate the software's utility by analyzing the underlying dataset's contents, by visulizing the evolution of publications in diverse scientific fields and relationships between them, by outlining scientometric facts associated with COVID-19 research, and by replicating commonly-used bibliometric measures of productivity and impact.

[**Stokes2022**] Chase Stokes and Marti Hearst. Why more text is (often) better: Themes from reader preferences for integration of charts and text, 2022.

[**Storey2022**] Margaret-Anne Storey, Brian Houck, and Thomas Zimmermann. How developers and managers define and trade productivity for quality. In *Proc. Conference on Human Aspects of Software Engineering (CHASE)*. ACM, May 2022, DOI 10.1145/3528579.3529177.

Abstract: Background: Developer productivity and software quality are different but related multi-dimensional lenses into the software engineering process. The terms are used liberally in industry settings, but there is a lack of consensus and awareness of what these terms mean in specific contexts and which trade-offs should be considered. Objective & Method: Through an exploratory survey study with developers and managers at Microsoft, we investigated how these cohorts define productivity and quality, how aligned they are in their views, how aware they are of other views, and if and how they trade quality for productivity. Results: We find developers and managers, as cohorts, are not well-aligned in their views of productivity—developers think more about work activities, while more managers consider performance or quality outcomes. We find developers and managers have more aligned views of what quality means, with the majority defining quality in terms of robustness, while the timely delivery of evolvable features that delight users are also key quality aspects. Over half of the developers and managers we surveyed make productivity and quality trade-offs but with good reasons for doing so. Conclusion: Alignment on how developers and managers define productivity and quality is essential if they are to design effective improvement interventions and meaningful metrics to measure productivity and quality improvements. Our research provides a frame for developers and managers to align their views and to make informed decisions on productivity and quality trade-offs.

[**Strode2022**] Diane Strode, Torgeir Dingsøyr, and Yngve Lindsjorn. A teamwork effectiveness model for agile software development. *Empir. Softw. Eng.*, 27(2), Mar 2022, DOI 10.1007/s10664-021-10115-0.

Abstract: Teamwork is crucial in software development, particularly in agile development teams which are cross-functional and where team members work intensively together to develop a cohesive software solution. Effective teamwork is not easy; prior studies indicate challenges with communication, learning, prioritization, and leadership. Nevertheless, there is much advice available for teams, from agile methods, practitioner literature, and general studies on teamwork to a growing body of empirical studies on teamwork in the specific context of agile software development. This article presents the agile teamwork effectiveness model (ATEM) for colocated agile development teams. The model is based on evidence from focus groups, case studies, and multi-vocal literature and is a revision of a general team effectiveness model. Our model of agile teamwork effectiveness is composed of shared leadership, team orientation, redundancy, adaptability, and peer feedback. Coordinating mechanisms are needed to facilitate these components. The coordinating mechanisms are shared mental models, communication, and mutual trust. We critically examine the model and discuss extensions for very small, multi-team, distributed, and safety-critical development contexts. The model is intended for researchers, team members, coaches, and leaders in the agile community.

[**Tan2022**] Wen Siang Tan, Markus Wagner, and Christoph Treude. Detect-

ing outdated code element references in software repository documentation, 2022.

[**Tan2023**] Xin Tan, Yiran Chen, Haohua Wu, Minghui Zhou, and Li Zhang. Is it enough to recommend tasks to newcomers? understanding mentoring on good first issues. ICSE 2023, 2023.
Abstract: Newcomers are critical for the success and continuity of open source software (OSS) projects. To attract newcomers and facilitate their onboarding, many OSS projects recommend tasks for newcomers, such as good first issues (GFIs). Previous studies have preliminarily investigated the effects of GFIs and techniques to identify suitable GFIs. However, it is still unclear whether just recommending tasks is enough and how significant mentoring is for newcomers. To better understand mentoring in OSS communities, we analyze the resolution process of 48,402 GFIs from 964 repositories through a mix-method approach. We investigate the extent, the mentorship structures, the discussed topics, and the relevance of expert involvement. We find that $\tilde{7}0\%$ of GFIs have expert participation, with each GFI usually having one expert who makes two comments. Half of GFIs will receive their first expert comment within 8.5 hours after a newcomer comment. Through analysis of the collaboration networks of newcomers and experts, we observe that community mentorship presents four types of structure: centralized mentoring, decentralized mentoring, collaborative mentoring, and distributed mentoring. As for discussed topics, we identify 14 newcomer challenges and 18 expert mentoring content. By fitting the generalized linear models, we find that expert involvement positively correlates with newcomers' successful contributions but negatively correlates with newcomers' retention. Our study manifests the status and significance of mentoring in the OSS projects, which provides rich practical implications for optimizing the mentoring process and helping newcomers contribute smoothly and successfully.

[**Tenhunen2023**] Saara Tenhunen, Tomi Männistö, Matti Luukkainen, and Petri Ihantola. A systematic literature review of capstone courses in software engineering, 2023, DOI 10.48550/ARXIV.2301.03554.
Abstract: Context: Tertiary education institutions aim to prepare their computer science and software engineering students for working life. While much of the technical principles are covered in lower-level courses, team-based capstone projects are a common way to provide students with hands-on experience and teach soft skills. Objective: This paper explores the characteristics of software engineering capstone courses presented in the literature. The goal of this work is to understand the pros and cons of different approaches by synthesising the various aspects of software engineering capstone courses and related experiences. Method: In a systematic literature review for 2007–2007, we identified 127 primary studies. These studies were analysed based on their presented course characteristics and the reported course outcomes. Results: The characteristics were synthesised into a taxonomy consisting of duration, team sizes, client and project sources, project implementation, and student assessment. We found out that capstone courses

generally last one semester and divide students into groups of 4–5 where they work on a project for a client. For a slight majority of courses, the clients are external to the course staff and students are often expected to produce a proof-of-concept level software product as the main end deliverable. The courses also offer versatile assessments for students throughout the project. Conclusions: This paper provides researchers and educators with a classification of characteristics of software engineering capstone courses based on previous research. We also further synthesise insights on the reported outcomes of capstone courses. Our review study aims to help educators to identify various ways of organising capstones and effectively plan and deliver their own capstone courses. The characterisation also helps researchers to conduct further studies on software engineering capstones.

[**Tian2022**] Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. What makes a good commit message? In *Proc. International Conference on Software Engineering (ICSE)*. ACM, May 2022, DOI 10.1145/3510003.3510205.

Abstract: A key issue in collaborative software development is communication among developers. One modality of communication is a commit message, in which developers describe the changes they make in a repository. As such, commit messages serve as an "audit trail" by which developers can understand how the source code of a project has changed-and why. Hence, the quality of commit messages affects the effectiveness of communication among developers. Commit messages are often of poor quality as developers lack time and motivation to craft a good message. Several automatic approaches have been proposed to generate commit messages. However, these are based on uncurated datasets including considerable proportions of poorly phrased commit messages. In this multi-method study, we first define what constitutes a "good" commit message, and then establish what proportion of commit messages lack information using a sample of almost 1,600 messages from five highly active open source projects. We find that an average of circa 44% of messages could be improved, suggesting the use of uncurated datasets may be a major threat when commit message generators are trained with such data. We also observe that prior work has not considered semantics of commit messages, and there is surprisingly little guidance available for writing good commit messages. To that end, we develop a taxonomy based on recurring patterns in commit messages' expressions. Finally, we investigate whether "good" commit messages can be automatically identified; such automation could prompt developers to write better commit messages.

[**Tigina2023**] Maria Tigina, Anastasiia Birillo, Yaroslav Golubev, Hieke Keuning, Nikolay Vyahhi, and Timofey Bryksin. Analyzing the quality of submissions in online programming courses, 2023.

Abstract: Programming education should aim to provide students with a broad range of skills that they will later use while developing software. An important aspect in this is their ability to write code that is not only correct but also of high quality. Unfortunately, this is difficult to control

in the setting of a massive open online course. In this paper, we carry out an analysis of the code quality of submissions from JetBrains Academy – a platform for studying programming in an industry-like project-based setting with an embedded code quality assessment tool called Hyperstyle. We analyzed more than a million Java submissions and more than 1.3 million Python submissions, studied the most prevalent types of code quality issues and the dynamics of how students fix them. We provide several case studies of different issues, as well as an analysis of why certain issues remain unfixed even after several attempts. Also, we studied abnormally long sequences of submissions, in which students attempted to fix code quality issues after passing the task. Our results point the way towards the improvement of online courses, such as making sure that the task itself does not incentivize students to write code poorly.

[**Timperley2021**] Christopher S. Timperley, Lauren Herckis, Claire Le Goues, and Michael Hilton. Understanding and improving artifact sharing in software engineering research. *Empirical Software Engineering*, 26(4), May 2021, DOI `10.1007/s10664-021-09973-5`.

[**Tissenbaum2021**] Mike Tissenbaum, David Weintrop, Nathan Holbert, and Tamara Clegg. The case for alternative endpoints in computing education. *Br. J. Educ. Technol.*, 52(3):1164–1177, May 2021, DOI `10.1111/bjet.13072`.
Abstract: This paper argues for a reexamination of the nature and goals of broad computing education initiatives. Instead of starting with specific values or goals, we instead begin by considering various desired endpoints of computing instruction and then work backward to reason about what form learning activities might take and what are the underlying values and principles that support learners in reaching these endpoints. The result of this exercise is a push for rethinking the form of contemporary computing education with an eye toward more diverse, equitable and meaningful endpoints. With a focus on the role that constructionistfocused pedagogies and designs can play in supporting these endpoints, we examine four distinct cases and the endpoints they support. This paper is not intended to encompass all the possible alternate endpoints for computer science education; rather, this work seeks to start a conversation around the nature of and need for alternate endpoints, as a means to reevaluate the current tools and curricula to prepare learners for a future of active and empowered computingliterate citizens.

[**Tiwari2023**] Deepika Tiwari, Martin Monperrus, and Benoit Baudry. Rick: Generating mocks from production data, 2023.
Abstract: Test doubles, such as mocks and stubs, are nifty fixtures in unit tests. They allow developers to test individual components in isolation from others that lie within or outside of the system. However, implementing test doubles within tests is not straightforward. With this demonstration, we introduce RICK, a tool that observes executing applications in order to au-

tomatically generate tests with realistic mocks and stubs. R I C K monitors the invocation of target methods and their interactions with external components. Based on the data collected from these observations, RICK produces unit tests with mocks, stubs, and mock-based oracles. We highlight the capabilities of RICK, and how it can be used with real-world Java applications, to generate tests with mocks.

[**Trautsch2023**] Alexander Trautsch, Johannes Erbel, Steffen Herbold, and Jens Grabowski. What really changes when developers intend to improve their source code: a commit-level study of static metric value and static analysis warning changes. *Empirical Software Engineering*, 28(2), Jan 2023, DOI 10.1007/s10664-022-10257-9.
Abstract: Many software metrics are designed to measure aspects that are believed to be related to software quality. Static software metrics, e.g., size, complexity and coupling are used in defect prediction research as well as software quality models to evaluate software quality. Static analysis tools also include boundary values for complexity and size that generate warnings for developers. While this indicates a relationship between quality and software metrics, the extent of it is not well understood. Moreover, recent studies found that complexity metrics may be unreliable indicators for understandability of the source code. To explore this relationship, we leverage the intent of developers about what constitutes a quality improvement in their own code base. We manually classify a randomized sample of 2,533 commits from 54 Java open source projects as quality improving depending on the intent of the developer by inspecting the commit message. We distinguish between perfective and corrective maintenance via predefined guidelines and use this data as ground truth for the fine-tuning of a state-of-the art deep learning model for natural language processing. The benchmark we provide with our ground truth indicates that the deep learning model can be confidently used for commit intent classification. We use the model to increase our data set to 125,482 commits. Based on the resulting data set, we investigate the differences in size and 14 static source code metrics between changes that increase quality, as indicated by the developer, and changes unrelated to quality. In addition, we investigate which files are targets of quality improvements. We find that quality improving commits are smaller than non-quality improving commits. Perfective changes have a positive impact on static source code metrics while corrective changes do tend to add complexity. Furthermore, we find that files which are the target of perfective maintenance already have a lower median complexity than files which are the target of non-perfective changes. Our study results provide empirical evidence for which static source code metrics capture quality improvement from the developers point of view. This has implications for program understanding as well as code smell detection and recommender systems.

[**Trinkenreich2022**] Bianca Trinkenreich, Igor Wiese, Anita Sarma, Marco Gerosa, and Igor Steinmacher. Women's participation in open source software: A survey of the literature. *ACM Transactions on Software Engineering*

and Methodology, 31(4):1–37, Aug 2022, DOI 10.1145/3510460.
Abstract: Women are underrepresented in Open Source Software (OSS) projects, as a result of which, not only do women lose career and skill development opportunities, but the projects themselves suffer from a lack of diversity of perspectives. Practitioners and researchers need to understand more about the phenomenon; however, studies about women in open source are spread across multiple fields, including information systems, software engineering, and social science. This paper systematically maps, aggregates, and synthesizes the state-of-the-art on women's participation in OSS. It focuses on women contributors' representation and demographics, how they contribute, their motivations and challenges, and strategies employed by communities to attract and retain women. We identified 51 articles (published between 2000 and 2021) that investigated women's participation in OSS. We found evidence in these papers about who are the women who contribute, what motivates them to contribute, what types of contributions they make, challenges they face, and strategies proposed to support their participation. According to these studies, only about 5% of projects were reported to have women as core developers, and women authored less than 5% of pull-requests, but had similar or even higher rates of pull request acceptances than men. Women make both code and non-code contributions and their motivations to contribute include, learning new skills, altruism, reciprocity, and kinship. Challenges that women face in OSS are mainly social, including lack of peer parity and non-inclusive communication from a toxic culture. We found ten strategies reported in the literature, which we mapped to the reported challenges. Based on these results, we provide guidelines for future research and practice.

[**Trinkenreich2023a**] Bianca Trinkenreich, Klaas-Jan Stol, Igor Steinmacher, Marco Gerosa, Anita Sarma, Marcelo Lara, Michael Feathers, Nicholas Ross, and Kevin Bishop. A model for understanding and reducing developer burnout, 2023.
Abstract: Job burnout is a type of work-related stress associated with a state of physical or emotional exhaustion that also involves a sense of reduced accomplishment and loss of personal identity. Burnt out can affect one's physical and mental health and has become a leading industry concern and can result in high workforce turnover. Through an empirical study at Globant, a large multi-national company, we created a theoretical model to evaluate the complex interplay among organizational culture, work satisfaction, and team climate, and how they impact developer burnout. We conducted a survey of developers in software delivery teams (n=3,281) to test our model and analyzed the data using structural equation modeling, moderation, and multi-group analysis. Our results show that Organizational Culture, Climate for Learning, Sense of Belonging, and Inclusiveness are positively associated with Work Satisfaction, which in turn is associated with Reduced Burnout. Our model generated through a largescale survey can guide organizations in how to reduce workforce burnout by creating a

climate for learning, inclusiveness in teams, and a generative organizational culture where new ideas are welcome, information is actively sought and bad news can be shared without fear.

[**Trinkenreich2023b**] Bianca Trinkenreich, Klaas-Jan Stol, Anita Sarma, Daniel M. German, Marco A. Gerosa, and Igor Steinmacher. Do i belong? modeling sense of virtual community among linux kernel contributors, 2023. **Abstract:** The sense of belonging to a community is a basic human need that impacts an individual's behavior, long-term engagement, and job satisfaction, as revealed by research in disciplines such as psychology, healthcare, and education. Despite much research on how to retain developers in Open Source Software (OSS) projects and other virtual, peer-production communities, there is a paucity of research investigating what might contribute to a sense of belonging in these communities. To that end, we develop a theoretical model that seeks to understand the link between OSS developer motives and a Sense of Virtual Community (SVC). We test the model with a dataset collected in the Linux Kernel developer community (N=225), using structural equation modeling techniques. Our results for this case study show that intrinsic motivations (social or hedonic motives) are positively associated with a sense of virtual community, but living in an authoritative country and being paid to contribute can reduce the sense of virtual community. Based on these results, we offer suggestions for open source projects to foster a sense of virtual community, with a view to retaining contributors and improving projects' sustainability.

[**Truong2022**] Kimberly Truong, Courtney Miller, Bogdan Vasilescu, and Christian Kästner. The unsolvable problem or the unheard answer? In *Proc. International Conference on Mining Software Repositories (MSR)*. ACM, May 2022, DOI `10.1145/3524842.3528488`.
Abstract: Talks at practitioner-focused open-source software conferences are a valuable source of information for software engineering researchers. They provide a pulse of the community and are valuable source material for grey literature analysis. We curated a dataset of 24,669 talks from 87 open-source conferences between 2010 and 2021. We stored all relevant metadata from these conferences and provide scripts to collect the transcripts. We believe this data is useful for answering many kinds of questions, such as: What are the important/highly discussed topics within practitioner communities? How do practitioners interact? And how do they present themselves to the public? We demonstrate the usefulness of this data by reporting our findings from two small studies: a topic model analysis providing an overview of open-source community dynamics since 2011 and a qualitative analysis of a smaller community-oriented sample within our dataset to gain a better understanding of why contributors leave open-source software.

[**Tshukudu2020**] Ethel Tshukudu and Quintin Cutts. Understanding conceptual transfer for students learning new programming languages. In *Proc. Conference on International Computing Education Research (ICER)*. ACM,

Aug 2020, DOI 10.1145/3372782.3406270.
Abstract: Prior research has shown that students face transition challenges between programming languages (PL) over the course of their education. We could not find research attempting to devise a model that describes the transition process and how students' learning of programming concepts is affected during the shift. In this paper, we propose a model to describe PL transfer for relative novices. In the model, during initial stages of learning a new language, students will engage in learning three categories of concepts, True Carryover Concepts, False Carryover Concepts, or Abstract True Carryover Concepts; during the transition, learners automatically effect a transfer of semantics between languages based on syntax matching. In order to find support for the model, we conducted two empirical studies. Study 1 investigated near-novice undergraduate students transitioning from procedural Python to object-oriented Java while Study 2 investigated near-novice postgraduate students doing a transfer from object-oriented Java to procedural Python. Results for both studies indicate that students had little or no difficulty with transitioning on TCC due to positive semantic transfer based on syntax similarities while they had the most difficulty transitioning on FCC due to negative semantic transfer. Students had little or no semantic transfer on ATCC due to differences in syntax between the languages. We suggest ways in which the model can inform pedagogy on how to ease the transition process.

[**Tuna2022**] Erdem Tuna, Vladimir Kovalenko, and Eray Tüzün. Bug tracking process smells in practice. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, May 2022, DOI 10.1145/3510457.3513080.
Abstract: Software teams use bug tracking (BT) tools to report and manage bugs. Each record in a bug tracking system (BTS) is a reporting entity consisting of several information fields. The contents of the reports are similar across different tracking tools, though not the same. The variation in the workflow between teams prevents defining an ideal process of running BTS. Nevertheless, there are best practices reported both in white and gray literature. Developer teams may not adopt the best practices in their BT process. This study investigates the non-compliance of developers with best practices, so-called smells, in the BT process. We mine bug reports of four projects in the BTS of JetBrains, a software company, to observe the prevalence of BT smells in an industrial setting. Also, we survey developers to see (1) if they recognize the smells, (2) their perception of the severity of the smells, and (3) the potential benefits of a BT process smell detection tool. We found that (1) smells occur, and their detection requires a solid understanding of the BT practices of the projects, (2) smell severity perception varies across smell types, and (3) developers considered that a smell detection tool would be useful for six out of the 12 smell categories.

[**Turk2021**] Tomaž Turk. SDFunc: Modular spreadsheet design with sheet-defined functions in microsoft excel. *Software: Practice and Experience*, 52(2):415–426, Sep 2021, DOI 10.1002/spe.3027.

Abstract: The goal of the SDFunc tool is to enable spreadsheet developers to build their model computations in Microsoft Excel according to the modular design approach, that is, the separation of the functionalities into independent, interchangeable modules with interfaces that provide input and output elements. This concept has been theoretically developed in recent years and is known as sheet-defined functions in the literature. In this article, we are presenting our implementation of the tool and the evaluation steps that we took to make the tool interesting and suitable for the assessment of the modular approach in spreadsheet development by the industry, specifically within organizational and companies' settings where the spreadsheet developers and end-users involved in experiments expect to use a well-established spreadsheet platform. We also demonstrated that sheet-defined functions can be implemented by development tools already present in Microsoft Excel.

[**VanBreukelen2023**] Sterre van Breukelen, Ann Barcomb, Sebastian Baltes, and Alexander Serebrenik. "still around": Experiences and survival strategies of veteran women software developers, 2023.

Abstract: The intersection of ageism and sexism can create a hostile environment for veteran software developers belonging to marginalized genders. In this study, we conducted 14 interviews to examine the experiences of people at this intersection, primarily women, in order to discover the strategies they employed in order to successfully remain in the field. We identified 283 codes, which fell into three main categories: Strategies, Experiences, and Perception. Several strategies we identified, such as (Deliberately) Not Trying to Look Younger, were not previously described in the software engineering literature. We found that, in some companies, older women developers are recognized as having particular value, further strengthening the known benefits of diversity in the workforce. Based on the experiences and strategies, we suggest organizations employing software developers to consider the benefits of hiring veteran women software developers. For example, companies can draw upon the life experiences of older women developers in order to better understand the needs of customers from a similar demographic. While we recognize that many of the strategies employed by our study participants are a response to systemic issues, we still consider that, in the shortterm, there is benefit in describing these strategies for developers who are experiencing such issues today.

[**Venturini2023**] Daniel Venturini, Filipe Roseiro Cogo, Ivanilton Polato, Marco A Gerosa, and Igor Scaliante Wiese. I depended on you and you broke me: An empirical study of manifesting breaking changes in client packages. TOSEM 2023, 2023.

Abstract: Complex software systems have a network of dependencies. Developers often configure package managers (e.g., npm) to automatically update dependencies with each publication of new releases containing bug fixes and new features. When a dependency release introduces backward-incompatible changes, commonly known as breaking changes, dependent

packages may not build anymore. This may indirectly impact downstream packages, but the impact of breaking changes and how dependent packages recover from these breaking changes remain unclear. To close this gap, we investigated the manifestation of breaking changes in the npm ecosystem, focusing on cases where packages' builds are impacted by breaking changes from their dependencies. We measured the extent to which breaking changes affect dependent packages. Our analyses show that around 12% of the dependent packages and 14% of their releases were impacted by a breaking change during updates of non-major releases of their dependencies. We observed that, from all of the manifesting breaking changes, 44% were introduced both in minor and patch releases, which in principle should be backward compatible. Clients recovered themselves from these breaking changes in half of the cases, most frequently by upgrading or downgrading the provider's version without changing the versioning configuration in the package manager. We expect that these results help developers understand the potential impact of such changes and recover from them.

[**Vidoni2021**] Melina Vidoni. Evaluating unit testing practices in r packages. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, May 2021, DOI 10.1109/icse43902.2021.00136.
Abstract: "Testing Technical Debt (TTD) occurs due to shortcuts (non-optimal decisions) taken about testing; it is the test dimension of technical debt. R is a package-based programming ecosystem that provides an easy way to install third-party code, datasets, tests, documentation and examples. This structure makes it especially vulnerable to TTD because errors present in a package can transitively affect all packages and scripts that depend on it. Thus, TTD can effectively become a threat to the validity of all analysis written in R that rely on potentially faulty code. This two-part study provides the first analysis in this area. First, 177 systematically-selected, open-source R packages were mined and analysed to address quality of testing, testing goals, and identify potential TTD sources. Second, a survey addressed how R package developers perceive testing and face its challenges (response rate of 19.4%). Results show that testing in R packages is of low quality; the most common smells are inadequate and obscure unit testing, improper asserts, inexperienced testers and improper test design. Furthermore, skilled R developers still face challenges such as time constraints, emphasis on development rather than testing, poor tool documentation and a steep learning curve."

[**Vidoni2022**] Melina Vidoni. Understanding roxygen package documentation in r. *Journal of Systems and Software*, 188:111265, Jun 2022, DOI 10.1016/j.jss.2022.111265.
Abstract: R is a package-based programming ecosystem that provides an easy way to install third-party code, datasets, and examples. Thus, R developers rely heavily on the documentation of the packages they import to use them correctly and accurately. This documentation is often written using Roxygen, equivalent to Java's well-known Javadoc. This two-part study

provides the first analysis in this area. First, 379 systematically-selected, open-source R packages were mined and analysed to address the quality of their documentation in terms of presence, distribution, and completeness to identify potential sources of documentation debt of technical debt that describes problems in the documentation. Second, a survey addressed how R package developers perceive documentation and face its challenges (with a response rate of 10.04%). Results show that incomplete documentation is the most common smell, with several cases of incorrect use of the Roxygen utilities. Unlike in traditional API documentation, developers do not focus on how behaviour is implemented but on common use cases and parameter documentation. Respondents considered the examples section the most useful, and commonly perceived challenges were unexplained examples, ambiguity, incompleteness and fragmented information.

[**Wang2020b**] Zhendong Wang, Yang Feng, Yi Wang, James A Jones, and David Redmiles. Unveiling elite developers' activities in open source projects. *ACM Trans. Softw. Eng. Methodol.*, 29(3):1–35, Jul 2020, DOI `10.1145/3387111`.

Abstract: Open source developers, particularly the elite developers who own the administrative privileges for a project, maintain a diverse portfolio of contributing activities. They not only commit source code but also exert significant efforts on other communicative, organizational, and supportive activities. However, almost all prior research focuses on specific activities and fails to analyze elite developers' activities in a comprehensive way. To bridge this gap, we conduct an empirical study with fine-grained event data from 20 large open source projects hosted on GITHUB. We investigate elite developers' contributing activities and their impacts on project outcomes. Our analyses reveal three key findings: (1) elite developers participate in a variety of activities, of which technical contributions (e.g., coding) only account for a small proportion; (2) as the project grows, elite developers tend to put more effort into supportive and communicative activities and less effort into coding; and (3) elite developers' efforts in nontechnical activities are negatively correlated with the project's outcomes in terms of productivity and quality in general, except for a positive correlation with the bug fix rate (a quality indicator). These results provide an integrated view of elite developers' activities and can inform an individual's decision making about effort allocation, which could lead to improved project outcomes. The results also provide implications for supporting these elite developers.

[**Wang2020c**] Qingye Wang. Why is my bug wontfix? In *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*. IEEE, Feb 2020, DOI `10.1109/ibf50092.2020.9034539`.

Abstract: Developers often use bug reports to triage and fix bugs. However, not every bug can be fixed eventually. To understand the underlying reasons why bugs are wontfix, we conduct an empirical study on three open source projects (i.e., Mozilla, Eclipse and Apache OpenOffice) in Bugzilla. First, we manually analyzed 600 wontfix bug reports. Second, we used the open

card sorting approach to label these bug reports why they were wontfix, and we summarized 12 categories of reasons. Next, we further studied the frequency distribution of the categories across projects. We found that Not Support bug reports are the majority of the wontfix bug reports. Moreover, the frequency distribution of wontfix bug reports across the 12 categories is basically similar for the three open source projects.

[**Wattenbach2022**] Leonhard Wattenbach, Basel Aslan, Matteo Maria Fiore, Henley Ding, Roberto Verdecchia, and Ivano Malavolta. Do you have the energy for this meeting? In *Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems*. ACM, May 2022, DOI 10.1145/3524613.3527812.

Abstract: Context. With "work from home" policies becoming the norm during the COVID-19 pandemic, videoconferencing apps have soared in popularity, especially on mobile devices. However, mobile devices only have limited energy capacities, and their batteries degrade slightly with each charge/discharge cycle. Goal. With this research we aim at comparing the energy consumption of two Android videoconferencing apps, and studying the impact that different features and settings of these apps have on energy consumption. Method. We conduct an empirical experiment by utilizing as subjects Google Meet and Zoom. We test the impact of multiple factors on the energy consumption: number of call participants, microphone and camera use, and virtual backgrounds. Results. Zoom results to be more energy efficient than Google Meet, albeit only to a small extent. Camera use is the most energy greedy feature, while the use of virtual background only marginally impacts energy consumption. Number of participants affect differently the energy consumption of the apps. As exception, microphone use does not significantly affect energy consumption. Conclusions. Most features of Android videoconferencing apps significantly impact their energy consumption. As implication for users, selecting which features to use can significantly prolong their mobile battery charge. For developers, our results provide empirical evidence on which features are more energy-greedy, and how features can impact differently energy consumption across apps.

[**Win2023**] Hsu Myat Win, Haibo Wang, and Shin Hwei Tan. Automatic detecting unethical behavior in open-source software projects, 2023.

Abstract: Given the rapid growth of Open-Source Software (OSS) projects, ethical considerations are becoming more important. Past studies focused on specific ethical issues (e.g., gender bias and fairness in OSS). There is little to no study on the different types of unethical behavior in OSS projects. We present the first study of unethical behavior in OSS projects from the stakeholders' perspective. Our study of 316 GitHub issues provides a taxonomy of 15 types of unethical behavior guided by six ethical principles (e.g., autonomy). Examples of new unethical behavior include soft forking (copying a repository without forking) and self-promotion (promoting a repository without self-identifying as contributor to the repository). We also identify 18 types of software artifacts affected by the unethical behavior. The diverse

types of unethical behavior identified in our study (1) call for attentions of developers and researchers when making contributions in GitHub, and (2) point to future research on automated detection of unethical behavior in OSS projects. Based on our study, we propose Etor, an approach that can automatically detect six types of unethical behavior by using ontological engineering and Semantic Web Rule Language (SWRL) rules to model GitHub attributes and software artifacts. Our evaluation on 195,621 GitHub issues (1,765 GitHub repositories) shows that Etor can automatically detect 548 unethical behavior with 74.8% average true positive rate. This shows the feasibility of automated detection of unethical behavior in OSS projects.

[**Wolter2022**] Thomas Wolter, Ann Barcomb, Dirk Riehle, and Nikolay Harutyunyan. Open source license inconsistencies on GitHub. *ACM Transactions on Software Engineering and Methodology*, Dec 2022, DOI `10.1145/3571852`.

Abstract: Almost all software, open or closed, builds on open source software and therefore needs to comply with the license obligations of the open source code. Not knowing which licenses to comply with poses a legal danger to anyone using open source software. This article investigates the extent of inconsistencies between licenses declared by an open source project at the top level of the repository, and the licenses found in the code. We analysed a sample of 1,000 open source GitHub repositories. We find that about half of the repositories did not fully declare all licenses found in the code. Of these, approximately ten percent represented a permissive vs. copyleft license mismatch. Furthermore, existing tools cannot fully identify licences. We conclude that users of open source code should not only look at the declared licenses of the open source code they intend to use, but rather examine the software to understand its actual licenses.

[**Wyrich2022**] Marvin Wyrich, Justus Bogner, and Stefan Wagner. 40 years of designing code comprehension experiments: A systematic mapping study, 2022.

[**Young2021**] Jean-Gabriel Young, Amanda Casari, Katie McLaughlin, Milo Z. Trujillo, Laurent Hebert-Dufresne, and James P. Bagrow. Which contributions count? analysis of attribution in open source. In *Proc. International Conference on Mining Software Repositories (MSR)*. IEEE, May 2021, DOI `10.1109/msr52588.2021.00036`.

Abstract: Open source software projects usually acknowledge contributions with text files, websites, and other idiosyncratic methods. These data sources are hard to mine, which is why contributorship is most frequently measured through changes to repositories, such as commits, pushes, or patches. Recently, some open source projects have taken to recording contributor actions with standardized systems; this opens up a unique opportunity to understand how community-generated notions of contributorship map onto codebases as the measure of contribution. Here, we characterize contributor acknowledgment models in open source by analyzing thousands of projects

that use a model called All Contributors to acknowledge diverse contributions like outreach, finance, infrastructure, and community management. We analyze the life cycle of projects through this model's lens and contrast its representation of contributorship with the picture given by other methods of acknowledgment, including GitHub's top committers indicator and contributions derived from actions taken on the platform. We find that community-generated systems of contribution acknowledgment make work like idea generation or bug finding more visible, which generates a more extensive picture of collaboration. Further, we find that models requiring explicit attribution lead to more clearly defined boundaries around what is and is not a contribution.

[**Zakaria2022**] Farid Zakaria, Thomas R. W. Scogland, Todd Gamblin, and Carlos Maltzahn. Mapping out the hpc dependency chaos, 2022.

[**Zerouali2021**] Ahmed Zerouali, Tom Mens, and Coen De Roover. On the usage of JavaScript, python and ruby packages in docker hub images. *Science of Computer Programming*, 207:102653, Jul 2021, DOI `10.1016/j.scico.2021.102653`.
Abstract: Docker is one of the most popular containerization technologies. A Docker container can be saved into an image including all environmental packages required to run it, such as system and third-party packages from language-specific package repositories. Relying on its modularity, an image can be shared and included in other images to simplify the way of building and packaging new software. However, some package managers allow to include duplicated packages in an image, increasing its footprint; and outdated packages may miss new features and bug fixes or contain reported security vulnerabilities, putting the image in which they are contained at risk. Previous research has focused on studying operating system packages within Docker images, but little attention has been given to third-party packages. This article empirically studies installation practices, outdatedness and vulnerabilities of JavaScript, Python and Ruby packages installed in 3,000 popular community Docker Hub images. In many cases, these installed packages missed important releases leading to potential vulnerabilities of the images. Our findings suggest that maintainers of Docker Hub community images should invest more effort in updating outdated packages contained in those images in order to significantly reduce the number of vulnerabilities. In addition to this, Python community images are generally much less outdated and much less subject to vulnerabilities than NodeJS and Ruby community images. Specifically for NodeJS community images, elimination of duplicate package releases could lead to a significant reduction in their image footprint.

[**Zhang2021**] Xunhui Zhang, Yue Yu, Georgios Gousios, and Ayushi Rastogi. Pull request decision explained: An empirical overview, 2021.
Abstract: Pull-based development model is widely used in open source, leading the trends in distributed software development. One aspect which has garnered significant attention is studies on pull request decision - iden-

tifying factors for explanation. Objective: This study builds on a decade long research on pull request decision to explain it. We empirically investigate how factors influence pull request decision and scenarios that change the influence of factors. Method: We identify factors influencing pull request decision on GitHub through a systematic literature review and infer it by mining archival data. We collect a total of 3,347,937 pull requests with 95 features from 11,230 diverse projects on GitHub. Using this data, we explore the relations of the factors to each other and build mixed-effect logistic regression models to empirically explain pull request decision. Results: Our study shows that a small number of factors explain pull request decision with the integrator same or different from the submitter as the most important factor. We also noted that some factors are important only in special cases e.g., the percentage of failed builds is important for pull request decision when continuous integration is used.

[**Zhang2022b**] Kaiwen Zhang and Guanjun Liu. Automatically transform rust source to petri nets for checking deadlocks, 2022.

[**Zhang2022c**] Yuxia Zhang, Hui Liu, Xin Tan, Minghui Zhou, Zhi Jin, and Jiaxin Zhu. Turnover of companies in OpenStack: Prevalence and rationale. *ACM Transactions on Software Engineering and Methodology*, 31(4):1–24, Oct 2022, DOI `10.1145/3510849`.

Abstract: To achieve commercial goals, companies have made substantial contributions to large open-source software (OSS) ecosystems such as OpenStack and have become the main contributors. However, they often withdraw their employees for a variety of reasons, which may affect the sustainability of OSS projects. While the turnover of individual contributors has been extensively investigated, there is a lack of knowledge about the nature of companies' withdrawal. To this end, we conduct a mixed-methods empirical study on OpenStack to reveal how common company withdrawals were, to what degree withdrawn companies made contributions, and what the rationale behind withdrawals was. By analyzing the commit data of 18 versions of OpenStack, we find that the number of companies that have left is increasing and even surpasses the number of companies that have joined in later versions. Approximately 12% of the companies in each version have exited by the next version. Compared to the sustaining companies that joined in the same version, the withdrawn companies tend to have a weaker contribution intensity but contribute to a similar scope of repositories in OpenStack. Through conducting a developer survey, we find four aspects of reasons for companies' withdrawal from OpenStack: company, community, developer, and project. The most common reasons lie in the company aspect, i.e., the company either achieved its goals or failed to do so. By fitting the survival analysis model, we find that commercial goals are associated with the probability of the company's withdrawal, and that a company's contribution intensity and scale are positively correlated with its retention. Maintaining good retention is important but challenging for OSS ecosystems, and our re-

sults may shed light on potential approaches to improve company retention and reduce the negative impact of company withdrawal.

[**Zheng2019**] Wei Zheng, Chen Feng, Tingting Yu, Xibing Yang, and Xiaoxue Wu. Towards understanding bugs in an open source cloud management stack: An empirical study of OpenStack software bugs. *J. Syst. Softw.*, 151:210–223, May 2019, DOI 10.1016/j.jss.2019.02.025.

[**Zhou2019**] Jiayuan Zhou, Shaowei Wang, Cor-Paul Bezemer, and Ahmed E. Hassan. Bounties on technical q&a sites: a case study of stack overflow bounties. *Empirical Software Engineering*, 25(1):139–177, Jul 2019, DOI 10.1007/s10664-019-09744-3.

[**Zhu2022**] Shuofei Zhu, Ziyi Zhang, Boqin Qin, Aiping Xiong, and Linhai Song. Learning and programming challenges of rust. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, May 2022, DOI 10.1145/3510003.3510164.

Abstract: Rust is a young systems programming language designed to provide both the safety guarantees of high-level languages and the execution performance of low-level languages. To achieve this design goal, Rust provides a suite of safety rules and checks against those rules at the compile time to eliminate many memory-safety and thread-safety issues. Due to its safety and performance, Rust's popularity has increased significantly in recent years, and it has already been adopted to build many safety-critical software systems. It is critical to understand the learning and programming challenges imposed by Rust's safety rules. For this purpose, we first conducted an empirical study through close, manual inspection of 100 Rust-related Stack Overflow questions. We sought to understand (1) what safety rules are challenging to learn and program with, (2) under which contexts a safety rule becomes more difficult to apply, and (3) whether the Rust compiler is sufficiently helpful in debugging safety-rule violations. We then performed an online survey with 101 Rust programmers to validate the findings of the empirical study. We invited participants to evaluate program variants that differ from each other, either in terms of violated safety rules or the code constructs involved in the violation, and compared the participants' performance on the variants. Our mixed-methods investigation revealed a range of consistent findings that can benefit Rust learners, practitioners, and language designers.