Software Ticks Need No Specifications

Christoph Reichenbach
Department of Computer Science

Lund University
Lund, Sweden
christoph.reichenbach@cs.lth.se

***Abstract*—Software bugs cost time, money, and lives. They drive software research and development efforts, and are central to modern software engineering. Yet we lack a clear and general definition of what bugs *are*. Some bugs are *defects*, clearly defined as failures to meet some requirement or specification. However, there are many forms of undesirable program behaviour that are completely compatible with a typical program's specification.**

In this paper, we argue that the lack of a criterion for identifying non-defect bugs is hampering the development of tools that find and fix bugs. We propose such a criterion, based on the idea of *wasted effort*, discuss how bugs that meet our definition of *software ticks* can complement defects, and sketch how our definition can help future work on software tools.

***Index Terms*—software ticks, software bugs, software defects**

I. Introduction

Software bugs are a natural part of software engineering. Many prominent advances in our field are due to programming language design to prevent certain bugs [1]–[3], methodologies to improve how software teams manage or understand bugs [4]–[7], and tools that automatically detect bugs [8]–[17] or even fix them [18], [19].

However, the literature seems to lack a clear definition of what a software bug *is*. This lack of clarity means that designers of bug-finding tools lack a general test for deciding whether (or in what context) a given bug pattern is *appropriate*. The impact of this question extends beyond bug finding: as we make progress in synthesising code [20], in mining and applying bug fixes [21], and in other strategies for manually or automatically evolving software [22], we need to ask whether these systems can introduce new bugs.

The nature of a bug is more than a philosophical question. Folklore already tells us that drawing the line between *bug* and (unexpected) *feature* can be difficult [23], and the literature confirms this idea: Koru and Tian [24] note that in an Open Source setting "the concept of defect is broad, including failures, faults, changes, new requirements, new functionalities, ideas, and tasks," where they describe "defect" and "bug" as "closely related" concepts. Herzig et al. [25] found that more than a third of bug reports in five Open Source projects were not actual bug reports. Sun [26] and Wang [27] surveyed issue tracker reports (including bug reports) and found that between a third and a quarter of "invalid" and "won't fix" issues (respectively) were due to reporters and developers disagreeing on behaviour or developers finding a bug not worth their effort[1]. These two issue categories can be quite prominent; in Mozilla and OpenJDK, there is one "invalid" or "won't fix" issue for roughly every four fixed issues (Table I). Thus, it is common for users and developers to disagree on whether something is a bug and worth fixing.

TABLE I
Fraction of fixed issues, non-issues, and "won't-fix" issues among **Closed** issues in the Mozilla bug tracker and **Done** issues in the OpenJDK tracker.

	Mozilla (2020-10-14)		**OpenJDK** (2020-10-14)	
Category	**Count**	**Fraction**	**Count**	**Fraction**
Total	910561	100%	269400	100%
Fixed	423486	46.5%	173954	64.6%
Non-Issue	71902	7.9%	20304	7.5%
Wont-Fix	54043	5.9%	19824	7.4%

In this paper, we examine software bugs that are not software *defects* (Section II) and find that the distinguishing characteristic for most of them is *wasted effort* (Section III). We propose the term *software tick* for such effort-wasting bugs and show how this definition can guide future bug-related research (Section IV).

II. Characterising Bugs

While we are not aware of a clear definition of *bugs*, the IEEE gives us an actionable definition of (software) *defects*:

> "***defect***: *An imperfection or deficiency in a work product where that work product does not meet its requirements or specifications and needs to be either repaired or replaced* [...]"
>
> — IEEE standard 1044-2009 [28]

This definition is valuable but assumes requirements or specifications. Many programs come with *some* specifications, in the form of human-readable documentation, types, unit tests, assertion checks, or perhaps even formal contracts [29], [30], but we cannot in general expect these to give us a *comprehensive* specification (cf. test coverage in Hilton et al. [31]). In other words, much of any given program's behaviour will be unspecified.

This work was partially supported by Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation.

[1]For Sun [26], we count categories 1, 2, and 3 but note that this survey was on a single closed-source repository. For Wang [27], we count "Expected Behaviour", "Not Worth to Fix", "Hard to Fix", and "Have Workaround".

As we saw in the introduction, there are many program properties that are not defects but that at least some users and developers find *undesirable*. In the following, we explore bug taxonomies, bug checkers, and other research literature to find program properties that are universally or at least widely undesirable, to see if we can find commonalities among them that allow us to form a useful criterion for non-defect bugs.

A. Bug Taxonomies

Bug taxonomies categorise individual bugs into general bug classes that can offer us some insight into common properties among large sets of bugs. For example, Catolino et al. [32] extract nine bug categories from 1280 bug reports. Their four largest categories are *Program Anomaly* issues (41.3%), including e.g. premature termination; *GUI-related* issues (17%), including aesthetic issues and "unusual" error messages; *Configuration* issues (16%), including incorrect dependencies and paths; and *Test Code-related* issues (7%), including ineffective unit tests. The remaining categories ($\leq$4% each) cover security, performance, access and deprecation, networking, and database issues.

While they do not explore which of their bugs are defects, their descriptions hint at several properties that a wide audience might find undesirable. For instance, their example of premature termination while running tests in the Eclipse IDE (Eclipse issue #92067) is clearly undesirable.

As an example of a *GUI-related* issue, consider report #1906 in the Chromium browser: visiting a website might show an outdated view of that site, and the only mechanism to refresh this view was to restart the browser. This behaviour was clearly undesirable at least for web site developers.

Some other GUI issues are more challenging to classify. Consider Chromium bug reports #150 and #33056, which ask to replicate behaviours from two unrelated software systems (Firefox and EMACS) that clash with intended behaviour in Chromium. Clearly, for some Firefox and EMACS users the Chromium behaviour is undesirable, but for others, Firefox and EMACS behaviour may be foreign and undesirable.

Meanwhile, *Test Code-related* issues (such as their example of ineffective unit tests) illustrate yet another perspective of bugs: Bugs that do not (directly) impair users, but are undesirable for developers.

What we can take away from this work is that:

1) some behaviours are undesirable but need not be defects,
2) some behaviours are trade-offs between one community and another,
3) program properties may be undesirable for developers but invisible to users, and
4) performance issues (including energy, execution time, and memory usage [33]) can be undesirable.

B. Contemporary Bug Checkers

Bug checking tools like FindBugs/SpotBugs [8], PMD [34], or Error Prone [35] scan code for "bug patterns" and report their matches to the code's developers. Hovemeyer and Pugh's key insight in FindBugs was that many small, highly specialised bug checkers might be more useful than a single general-purpose analysis, which means that these tools offer a broad variety of such patterns.

TABLE II
OUR CLASSIFICATION OF FINDBUGS 0.5.0 BUG CATEGORIES. SOME BUGS FALL INTO MULTIPLE CATEGORIES.

Category	Count	Example
Useless	19	DM_STRING_CTOR
Threads	10	NN_NAKED_NOTIFY
Defect	6	FI_EXPLICIT_INVOCATION
Subtle	6	DM_STRING_CTOR
Risky	3	MS_EXPOSE_REP
Stuck	2	SP_SPIN_ON_FIELD
Other	4	RU_INVOKE_RUN

Since these tools offer many hundreds of different patterns, we selected a subset by examining all 50 bug patterns from the original release of FindBugs (0.5.0 [2]). While these bug patterns might be limited by the analysis infrastructure of their time, they are likely to focus on bug patterns that the FindBugs authors considered particularly important.

We marked each bug pattern by *why* this pattern matches undesirable code, and observed a considerable variety among the bug patterns. We summarise our classification in Table II and describe our categories below:

- **Useless** bug patterns capture (seemingly) inefficient code, i.e., code that could be replaced by simpler and more efficient code, and ineffective code. Specifically, nine of the ten patterns describe (suspected) failed method overriding due to typos or parameter type mismatch, but could also rarely also be *Subtle* (see below), though we did not count them as such.
- **Threads** patterns detect flaky concurrency and complex or incorrect concurrency patterns (five instances each). Either of these may be challenging to test for and debug.
- **Defect** patterns catch violations of Java Class Library specifications.
- **Subtle** captures program behaviour that requires in-depth knowledge of the Java language semantics (e.g., static class initialisation order) that at least novice Java programmers will normally lack. Even if the program behaves "as intended", these patterns increase the risk that novice developers will break the code or (needlessly) spend substantial time understanding this code.
- **Risky** captures coding practices, such as exposing mutable arrays, that come with implicit usage contracts that the language cannot enforce (e.g., whether an exposed array must only be read). This category is similar to *Subtle*, but the subtlety here comes from the developers' own APIs.
- **Stuck** captures infinite loops and deadlocks. These are clearly undesirable for end-users.
- **Other** captures several properties that are not necessarily bugs but look very similar to incorrect uses of well-

[2]https://github.com/findbugsproject/findbugs/tree/f81d23a330656a62bf7edafca81309c9d1fe3204

known coding patterns (e.g., defining a non-static field `serialVersionID`, which is the same name as a *static* field that the Java serialisation API can use if present).

As in the previous section, we find several take-aways:

1) The FindBugs designers consider useless and ineffective code to be a bug.
2) Similarly, they find risky and subtle code undesirable.
3) Overall, we find that these patterns make a case against *needless complexity*.

C. Bugs as Complexity

While we lack the space for a complete survey, we consider three strands of bug finding work that look at suspicious code behaviour: API protocols, "bad smells", and clones.

a) API Protocols: Not all sequences of API calls are sensible. Expert developers can give us specifications for how to use these APIS, in the form of API protocols, which we can then use to check for API misuses [14], [17], [36]. We can also *infer* API protocols from execution traces (as in Pradel et al. [37]). While these specifications are not "absolute" in the same sense as hand-written ones (e.g., Pradel et al. report a 49% false positive rate), inferred API protocols allow us to find bugs that manifest as *divergence from the norm*.

b) Bad Smells: Fowler introduces "bad smells" [38] as hints that code needs refactoring. Palomba et al. [39] explored several of these smells in an empirical study with Open Source and industrial developers. They report the following insight: '*Smells related to complex/long source code are generally perceived as an important threat by developers. This happens for Complex Class, God Class, Long Method, and Spaghetti Code*.' The developers did not consider other smells, especially *Class Data Should Be Private*, *Middle Man*, or *Inappropriate Intimacy*, to be similarly serious. We observe that a key difference between these two classes of smells is that the former focus on the complexity of the behaviour of a single entity, whereas the latter focus on positioning functionality within two adjacent layers of abstraction.

c) Code Clones: One smell that Palomba et al. did not explore is *Duplicated Code*. Code clones can contribute to bug spread and complicate maintenance when they evolve independently [40], especially for *structural* clones [41].

All three of these strands of work further confirm that complexity (in which we include *divergence from the norm*) is undesirable. This insight is not surprising; for instance, the field of Cognitive Load Theory [42] has empirically explored for decades how *accidental complexity* (in our terminology) makes it harder for learners to build mental models of the material that they study. In the terms of Cognitive Load Theory, the *extraneous load* of code complexity can inhibit *schema construction* in learners, where schemas are "generic, abstracted knowledge structure" with "default values" [43].

Thus, we argue that unnecessary complexity is undesirable: it creates extra effort when changing code, and it interferes with the developers' ability to understand code.

III. Software Ticks

As we have seen, there are many different classes of static and dynamic program properties that researchers and practitioners consider "bugs". Many of these correspond to *defects* as defined in IEEE 1044-2009, because they explicitly violate a specification, either one that is part of the program itself, or one that is implicit in the framework, libraries, language, or the requirements. We can even go further and argue that there are certain obvious implicit specifications, such as that the code must not enter an "unproductive" infinite loop or rely on "undefined behaviour" (as in C/C++).

However, this leaves out many of the bug types that we saw:

1) *UI Bugs*, in which the program's interface violated user expectations,
2) *Performance Bugs*, in which the program used excessive computational resources,
3) *Unnecessary Complexity* in a program's structure that complicates maintainability and evolution.

These three grievances inconvenience users or developers and possibly waste machine resources. For *UI Issues*, users must spend effort to adapt to the software's UI norms before they can use it. For *Performance Bugs*, the program wastes time, memory, energy and possibly other resources. Finally, *Unnecessary Complexity* wastes developer time.

Thus, there is a common theme among these three types of non-defect bugs: They *waste effort*.

However, the converse does not hold: wasted effort does not mean that a program has a bug. For instance, we can use the Apache web server's URL-rewriting facilities to calculate our taxes[3], which would certainly be wasteful, but we can hardly consider it a bug in this web server.

A criterion for non-defect bugs must therefore factor out obvious misuse. This can be easy if we have a specification, but even without one, we can often find the *scope* of a piece of software in informal documentation. For instance, the Apache server's README file[4] calls the software "a [...] web server", which clearly marks our example above as out of scope. Below, we use "scope" and "specification" interchangeably.

A. Defining Software Ticks

Since 'bug that wastes effort' is a bit of a mouthful, we here adopt the term *tick* for bugs that take away resources without contributing anything useful.[5]

Definition 1. For a given scope and two software systems s and s', s' is *more resource-efficient* than s iff within the scope, s' can produce equivalent results to s and s' never consumes more and sometimes consumes fewer resources than s.

Definition 2. A *software tick* is a property of the structure or behaviour of a software system s such that there exists a

[3]https://web.archive.org/web/20200605041243/olsner.se/2008/01/21/an-excursion-in-mod_rewrite/

[4]https://github.com/apache/httpd/blob/trunk/README, 2020-11-17

[5]For contexts in which a more formal term is appropriate, we propose to repurpose the term "Software Flaw" as a synonym for "Tick".

procedure for transforming s into a software system s' that (a) lacks this property and (b) is more resource-efficient than s.

To emphasise, we use the term "resources" broadly, encompassing energy, time, and attention of the entities that interact with the software, including users, developers, and other software systems..

According to our definition, a software tick is then a software property due to which the software's resource usage across all resource dimensions is not Pareto-optimal. We thus exclude trade-offs: sacrificing user convenience to reduce memory and execution time is not a tick, since these are separate resource dimensions; thus, we also exclude the cost for fixing the tick itself from our definition.

Conceptually we can relax this definition with "conversion rates" between different forms of effort, or allow small deltas e.g. strictly for machine effort (to emphasise the importance of human effort), but in the absence of empirical studies we leave these refinements for future work.

B. Bugs, Ticks, and Defects

Software defects capture program properties that violate specifications and requirements. In principle, we can show that a program is free of defects.

Software ticks capture deficiencies across multiple forms of software quality, e.g. efficiency, reliability, usability, and maintainability. In general, we cannot show that a program is free of ticks, since we cannot show the absence of possible performance or usability enhancements.

Defects and ticks thus complement each other, one capturing what must not be and one capturing what can be better. These two categories can intersect if software quality is part of a program's specification. While practitioners and the literature do not agree on where to draw the line between bug and non-bug issues [24], the nomenclature of *performance bugs* [33] as well as the popularity of bug patterns for bad coding practices (Section II-B) are points in favour of treating ticks as a category of bugs.

IV. Impact

Ticks are bugs that we can fix with impunity; nobody will miss them. They offer a new perspective on bug detection, bug fixing, and bug ranking, and open new directions for research, as we detail below.

a) Finding and Fixing Ticks: Ticks set a high standard for bug checkers: if the developers remove the tick, the resultant program must be strictly better than it was before. Many of the bug patterns that we find in existing bug checkers (e.g., those based on dead code) match or approximate this definition, with only a few side conditions (cf. *Subtle* behaviour in Section II-B) standing in the way. Others, including "bad smells" such as public fields instead of getters/setters in Java, do not clearly match, since their impact on future maintenance cost may or may not outweigh their benefit.

Ticks also address a conceptual limitation in automatic bug-fixing tools: existing tools in this category focus on *defects*, but when they remove a defect, they may introduce a *tick*. Complementing these tools with automatic *tick removal* can allow us to compensate, and to automatically improve software *quality*. Moreover, we can consider the bug fixing tool's effort in our effort metric to steer tick removal towards better *automatic fixability*.

b) Ranking Ticks: Ranking bug reports can be challenging, but our notion of *effort* adds new metrics to this process.

For example, consider a program that reads in data, performs a complex computation, and produces output. If the program aborts with an execution failure (e.g., an uncaught exception) right after startup that signals that the output file name is missing, then this failure is both *timely* and *actionable*. In other words, this failure might have the form of an uncaught exception, but the user needs little effort to recover.

If the program instead triggers a failure that merely signals that "something went wrong", then this failure is no longer actionable and thus *wastes user effort*. If the failure provides meaningful information but triggers only at the end of data processing, it is no longer timely, wastes machine effort, and may come at additional cost to users [44].

Here, our notion of *wasted user effort* allowed us to see how *timeliness* and *actionability* help rank the effort attached to a potential tick. We expect that *wasted effort* can overall help us better understand when and how to report bugs.

c) Research: As we saw, we can use *wasted effort* to find new types of ticks. Researchers in the area of performance bugs have already explored this space, but we expect that there is substantial room left, especially for user effort: when does the system wait for the user when it could already be making progress? Does the user really have to manually select or confirm this information?

Finding ticks through *wasted developer effort* is more challenging, as this effort depends on the developers' future plans for the software system. However, we can look to *complexity* (Section II-C) as a proxy metric for effort: how much must a developer know to correctly evolve this code? Are developers consistent in what they do? The latter gives us a new perspective on bug patterns that represent "bad style" (e.g., "bad" method names or confusing overloading). If we find many matches for such a pattern, then this pattern may indicate a particular idiosyncrasy. If we can explain this idiosyncrasy though second-order bug rules [45] such as 'all classes that are tagged `@Legacy` trigger this style bug', we can not only report this issue more concisely, but also gain a better understanding of the effort attached to it.

Finally, we argue that our notions of *wasted effort* and *tick* add a new lens through which we can view bug candidates and help us prioritise our research efforts.

V. Conclusions

We have proposed the concept of *software ticks* to describe bugs that waste user effort, developer effort, or machine effort, and discuss how this concept complements *software defects*. Moreover, we have outlined how software ticks offer new directions in finding, managing, and resolving bugs.

REFERENCES

[1] B. C. Pierce, *Types and Programming Languages*. Cambridge, Massachusetts, USA: The MIT Press, 2002.

[2] N. D. Matsakis and F. S. Klock, "The rust language," *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, 2014.

[3] P. R. Wilson, "Uniprocessor garbage collection techniques," in *International Workshop on Memory Management*. Springer, 1992, pp. 1–42.

[4] P. Runeson, "A survey of unit testing practices," *IEEE software*, vol. 23, no. 4, pp. 22–29, 2006.

[5] A. G. Koru and J. Tian, "Defect handling in medium and large open source projects," *IEEE Software*, vol. 21, no. 4, pp. 54–61, 2004.

[6] L. Williams, E. M. Maximilien, and M. Vouk, "Test-driven development as a defect-reduction practice," in *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.* IEEE, 2003, pp. 34–45.

[7] A. Zeller, *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[8] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *Acm sigplan notices*, vol. 39, no. 12, pp. 92–106, 2004.

[9] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 77–88, 2012.

[10] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *International Conference on Computer Aided Verification*. Springer, 2006, pp. 419–423.

[11] C. Flanagan and S. N. Freund, "Fasttrack: efficient and precise dynamic race detection," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 121–133, 2009.

[12] A. P. Tolmach and A. W. Appel, "Debugging standard ML without reverse engineering," in *Proceedings of the 1990 ACM conference on LISP and functional programming*, 1990, pp. 1–12.

[13] C. Reichenbach, N. Immerman, Y. Smaragdakis, E. E. Aftandilian, and S. Z. Guyer, "What can the gc compute efficiently?: A language for heap assertions at gc time," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 256–269. [Online]. Available: http://doi.acm.org/10.1145/1869459.1869482

[14] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler *et al.*, "Cognicrypt: supporting developers in using cryptography," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 931–936.

[15] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977, pp. 238–252.

[16] P. O'Hearn, "Separation logic," *Communications of the ACM*, vol. 62, no. 2, pp. 86–95, 2019.

[17] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions," in *OSDI'00*, ser. OSDI'00. USA: USENIX Association, 2000.

[18] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2016.

[19] Y. Tian, J. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *2012 34th international conference on software engineering (ICSE)*. IEEE, 2012, pp. 386–396.

[20] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, *Syntax-guided synthesis*. IEEE, 2013.

[21] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. L. Traon, "Mining fix patterns for findbugs violations," 2018.

[22] R. M. Tsoupidi, R. Castañeda Lozano, and B. Baudry, "Constraint-based software diversification for efficient mitigation of code-reuse attacks," in *Principles and Practice of Constraint Programming*, H. Simonis, Ed. Cham: Springer International Publishing, 2020, pp. 791–808.

[23] Anonymous, *The Hackers' Dictionary of Computer Jargon*, 1992, vol. 38.

[24] A. G. Koru and J. Tian, "Defect handling in medium and large open source projects," *IEEE software*, vol. 21, no. 4, pp. 54–61, 2004.

[25] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 392–401.

[26] J. Sun, "Why are bug reports invalid?" in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2011, pp. 407–410.

[27] Q. Wang, "Why is my bug wontfix?" in *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*, 2020, pp. 45–54.

[28] "IEEE Standard Classification for Software Anomalies - Redline," *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993) - Redline*, pp. 1–25, 2010.

[29] B. Meyer, "Applying'design by contract'," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[30] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of jml: A behavioral interface specification language for java," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 3, pp. 1–38, 2006.

[31] M. Hilton, J. Bell, and D. Marinov, "A large-scale study of test coverage evolution," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 53–63. [Online]. Available: https://doi.org/10.1145/3238147.3238183

[32] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci, "Not all bugs are the same: Understanding, characterizing, and classifying bug types," *Journal of Systems and Software*, vol. 152, pp. 165 – 181, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121219300536

[33] A. B. Sánchez, P. Delgado-Pérez, I. Medina-Bulo, and S. Segura, "Tandem: A taxonomy and a dataset of real-world performance bugs," *IEEE Access*, vol. 8, pp. 107 214–107 228, 2020.

[34] T. Copeland, *PMD applied*. Centennial Books Arexandria, Va, USA, 2005, vol. 10.

[35] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan, "Building useful program analysis tools using an extensible java compiler," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, 2012, pp. 14–23.

[36] O. Legunsen, W. U. Hassan, X. Xu, G. Roundefinedu, and D. Marinov, "How Good Are the Specs? A Study of the Bug-Finding Effectiveness of Existing Java API Specifications," in *ASE'16*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 602–613. [Online]. Available: https://doi.org/10.1145/2970276.2970356

[37] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross, "Statically Checking API Protocol Conformance with Mined Multi-Object Specifications," in *ICSE '12*, ser. ICSE '12. IEEE Press, 2012, p. 925–935.

[38] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[39] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 101–110.

[40] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, May 2009, pp. 485–495.

[41] J. Kanwal, H. A. Basit, and O. Maqbool, "Structural clones: An evolution perspective," in *2018 IEEE 12th International Workshop on Software Clones (IWSC)*, 2018, pp. 9–15.

[42] F. Paas, J. E. Tuovinen, H. Tabbers, and P. W. M. V. Gerven, "Cognitive load measurement as a means to advance cognitive load theory," *Educational Psychologist*, vol. 38, no. 1, pp. 63–71, 2003. [Online]. Available: https://doi.org/10.1207/S15326985EP3801_8

[43] S. T. Fiske and P. W. Linville, "What does the schema concept buy us?" *Personality and Social Psychology Bulletin*, vol. 6, no. 4, pp. 543–557, 1980. [Online]. Available: https://doi.org/10.1177/014616728064006

[44] G. Mark, D. Gudith, and U. Klocke, "The cost of interrupted work: more speed and stress," in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, 2008, pp. 107–110.

[45] K. Li, C. Reichenbach, Y. Smaragdakis, and M. Young, "Second-order constraints in dynamic invariant inference," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 103–113. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491457