Searching for Technical Debt – An Empirical, Exploratory, and Descriptive Case Study

Rolf-Helge Pfeiffer
IT University of Copenhagen
Copenhagen, Denmark
ropf@itu.dk

***Abstract*—Commonly, Technical Debt (TD) is used as metaphor to describe *"technical compromises that are expedient in the short term, but that create a technical context that increases complexity and cost in the long term"* [1]. Since TD is a metaphor, there does not exist a uniform understanding of what concretely such *"technical compromises"* are. Practitioners, researchers, and tools all subsume and consider widely different concepts as TD. In this paper, we set out to empirically and exploratorily, identify potential *"technical compromises"* that increase cost and complexity of modifications of two open-source database systems (Apache `Cassandra` and GCHQ `Gaffer`). In a manual investigation of 217 commits that are associated to 40 of the most costly and complex issues, we find that refactorings in the sense of Ur-TD [2] are often related to high complexity of modifications and that high cost is due to organization and coordination of work. Other than that, we cannot identify any *"technical compromises"* that can explain high cost and complexity of the studied contributions.**

I. Introduction

In 1992, Cunningham describes the iterative development of the financial system `WyCash` and introduces the Technical Debt (TD) metaphor: *"Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite."* [3]. Since then, the metaphor has attracted a lot of attention leading to many different interpretations on what TD actually is. For example, practitioners consider TD anything from *"a metaphor for the accumulation of unresolved issues in a software project"* [4], *"the difference between what was promised and what was actually delivered"*[1], over *"hard-to-read code, lack of test automation, duplication, tangled dependencies, etc."*[2], to *"delayed technical work that is incurred when technical shortcuts are taken"* [5]. Tools, such as, `SonarQube` with the underlying Squale model or `CAST AIP`, consider TD to be mainly code smells or other structural patterns that can be detected via static analysis [6], [7]. Also researchers present various definitions of TD, e.g., as *"...the invisible results of past decisions about software that affect its future"* [8], *"those internal software development tasks chosen to be delayed, but that run a risk of causing future problems if not done eventually"* [9], or as *"a metaphor, referring to the eventual financial consequences of trade-offs between shrinking product time to market and poorly specifying, or implementing a software product, throughout all development phases"* [10].

This lack of common understanding of TD and its sources is confirmed by research [11]–[13]. Countering that conceptual fragmentation, the participants of Dagstuhl Seminar 16162 observe that the *"software engineering community is converging on defining technical debt as making technical compromises that are expedient in the short term, but that create a technical context that increases complexity and cost in the long term"* [1]. Since it is not defined what such technical compromises (TCs) precisely are –they are only generally described as certain *"design or implementation constructs"*–, our goal in this paper is to identify concrete instances of TCs that are constitutional for TD. That is, we take the definition of TD [1] literally and apply it in our empirical and exploratory study of two open-source database systems (Apache `Cassandra` and GCHQ `Gaffer`). The driving research question for our work is: Can we identify *technical compromises* that cause work to be most costly and complex?

After introducing two case systems (Sec. II), we manually inspect 40 of the most complex and costly contributions from these two systems (Sec. III). We conclude (Sec. V), finding that developers actually pay back TD in the sense of Cunningham's intended meaning of TD (now called *Ur-TD* [2]), i.e., refactorings to adapt systems to new mental models and use cases. Other than that, we cannot identify any TCs that can explain increased cost and complexity of the studied contributions.

The main contribution of this paper is the empirical case study, in which we attempt to distill TC (and thereby TD) out of costly and complex development work (tickets with associated commits). We are not aware of a similar study. A reproduction kit with all code and data is available online[3].

II. Background

1) Terminology: Software projects often organize work via *issue trackers*, such as, Atlassian's `Jira`. Work items in *issue trackers* are called *tickets* or *issues*; we use both terms synonymously. *Issues* can contain descriptions of any task, e.g., enhancements, new features, bugs, etc. Amongst others, tickets can be *created* and *closed* or *resolved* (we use the latter two synonymously). Tickets can be *resolved* without any modification of software, e.g., unwanted features or not reproducible bugs are marked as *won't fix* (or similar) and the respective issue is closed without a modification of the respective software. Alternatively, tickets are *resolved* via work that modifies the respective software via one or more *commits* to the project's Version Control System (VCS). We call one or more commits that resolve a ticket a *contribution*. Usually,

commits refer to a corresponding ticket via a ticket identifier in the commit message. The *lead time* of a contribution [14] is the difference of ticket closing time and ticket creation time.

In the body of this paper, we are relying the definition of TD as *"technical compromises that are expedient in the short term, but that create a technical context that increases complexity and cost in the long term"* [1]. Since not further specified by the authors, we use the term *complexity* similar to Basili [15], to denote the difficulty of contributing and integrating work into existing software, the Contribution Complexity (CC) [16]. We consider *cost* to be the lead time of contributions.

Note, we use the term *refactoring* not only for behavior-preserving code transformations but to describe any work that tries to improve maintainability, understandability, etc. [17].

2) Case Systems: Due to limited resources and space, we decide a priori to study only two open-source Database Management Systems (DBMSs). `Gaffer` is a large-scale entity and relation DBMS[4] (graph database), which is created mainly by the British Government Communications Headquarters (GCHQ). Since 2015 it is an open-source project. Its sources are available on Github[5], and the project uses Github's issue tracker[6]. Originally developed by Facebook [18], Apache `Cassandra`[7] is an open-source, distributed, wide-column store, NoSQL DBMS. In 2008 it was open-sourced and since 2010 it is an Apache top level project. The project uses Jira[8] as issue tracker and its sources are available on Github[9]. Both systems are written mainly in Java and both are under Apache-2.0 license.

III. RQ: Can we identify technical compromises that cause work to be most costly and complex?

In this section, we try to identify TCs that may cause contributions to take long or to be complex.

1) Method: We export all tickets, ticket identifiers, creation and closing times, etc., from the projects' issue trackers and store them as CSV files. For all closed tickets with contribution, we compute the lead time (t_{lead}) as the difference between ticket closing and creation time.

We identify contributions by mapping commits to respective issues via string references in commit messages. For example, commits refer to tickets via strings matching the regular expressions `(Gh |gh-)\d+( |$)` or `CASSANDRA-\d+( |$)` in `Gaffer` and `Cassandra` respectively.

For all resolved issues, we compute the *contribution complexity* with the `ConCom` tool [16], which maps complexities of contributions to a score labeled *low*, *moderate*, *medium*, *elevated*, or *high*. Inspired by Basili [15], CC is a metric that indicates, how difficult it is for a developer to modify existing software with a given contribution. The metric combines size-based and entropy-based metrics to assess the size and dispersion of changes (change scattering within files and across methods) and thereby the complexity of contributing and integrating a change to a system.

There are 7,877 and 821 resolved tickets with contribution for `Cassandra` and `Gaffer` respectively. Average lead times for `Cassandra` are ca. $61.9d$ ($min \approx 2min$, $max \approx 2\,320d$, $std \approx 156.6d$, $median \approx 9.3d$) `Cassandra` and ca. $38.9d$ ($min \approx 6min$, $max \approx 1\,399d$, $std \approx 120.3d$, $median \approx 7.2d$) for `Gaffer`.

From all closed issues with contributions (filtered for outliers with $1.5 \times IQR$ rule) [19], which excludes those lead times larger than 1.5 times the interquartile range of the 0.25 and 0.75 quantiles, we select the ten issues with longest lead times from `Cassandra` and `Gaffer` respectively. Furthermore, we select ten of the most complex (CC) issues from both systems respectively. That is, we select all issues with *high* CC score (five from `Cassandra` and three from `Gaffer`) and we randomly sample five issues with *elevated* CC from `Cassandra` and seven from `Gaffer`. The 40 selected issues with links to issue trackers and links to the corresponding commits are automatically converted into a Jupyter notebook, which serves as protocol during manual inspection of contributions. During inspection, we first read each of the 40 tickets with associated discussions and thereafter, we examine each of the 217 associated commits on Github directly. We map each contribution to a *kind* of change to be able to coarsely indicate the purpose of a contribution. We use the four change types *corrective* (Cor), *preventive* (Prv), *adaptive* (Adp), and *perfective* (Prf) change from ISO/IEC 14764 [20] for categorization. *Corrective* changes address errors and faults in the software, *preventive* changes increase its understanding and maintainability, *adaptive* changes adapt software to changing environments, and *perfective* changes introduce new features and adapt it to evolving requirements.

We have no a priori list of precise *"design or implementation constructs"* that are TCs [1]. Our goal is to exploratively identify these. This open-ended process is inspired by Guo et al. [21], who let developers identify underspecified TD items, where in our case the main author performs identification of TC. Essentially, we try to identify TCs similar to Kitchenham's *transcendental view* of software quality [22], that equates quality to *"something that can be recognized but not defined"*. During identification, we note our observations in our protocol, decide on the kind of change, and finally, we decide for each contribution if cost or complexity is caused by TCs that have to be circumvented.

2) Results: The left-hand side of Tab. I lists the 20 most costly tickets that are resolved with contributions, 10 for `Gaffer` and `Cassandra` respectively. To the right-hand side, the 20 tickets that are resolved with most complex contributions are listed. The full protocol with results is accessible online[10].

In `Gaffer`, issues with longest lead times are closed after 65 to 71 days, which is almost twice the average lead time. These issues are of low, normal, or high priority (prior.), where GH-2024 does not have a priority assigned (n/a). Most contributions to these issues are perfective changes (Prf), i.e., new features, such as, GH-139, which adds Python scripting support to the database or GH-2145, which adds a feature to return partial walks from the underlying graph. GH-259 and GH-1826 are corrective changes (bug fixes), where the former fixes a link in Javadoc and the latter fixes a serialization bug caused by a dependency. The preventive change GH-516 adds a new

Table I
THE 40 MOST COSTLY AND MOST COMPLEX CONTRIBUTIONS WITH MANUALLY IDENTIFIED KIND OF CHANGE AND TECHNICAL COMPROMISES (TCS).

Most costly (t_{lead})												Most complex (Contribution Complexity (CC))											
Gaffer						Cassandra						Gaffer						Cassandra					
Issue	t_{lead}	CC	prior.	kind	TC	Issue	t_{lead}	CC	prior.	kind	TC	Issue	t_{lead}	CC	prior.	kind	TC	Issue	t_{lead}	CC	prior.	kind	TC
GH-2145	71d	low	high	Prf	✗	CAS-2691	113d	low	norm.	Prf	✗	GH-1884	14d	high	n/a	Cor/Prf	✗	CAS-15066	79d	high	high	Prf	✗
GH-1099	70d	elev.	norm.	Prf	✗	CAS-8290	113d	low	low	Cor	✗	GH-776	35d	high	norm.	Prf	✗	CAS-13304	544d	high	urgent	Prv/Prf	✗
GH-139	70d	inter.	high	Prf	✗	CAS-13760	113d	low	low	Prf	✗	GH-538	302d	high	norm.	Prf	✗	CAS-9705	21d	high	norm.	Prv	✗
GH-516	69d	inter.	low	Prv	✗	CAS-10625	112d	low	low	Cor	✗	GH-375	49d	elev.	high	Cor	✗	CAS-9459	91d	high	norm.	Prv	✗
GH-2024	69d	inter.	n/a	Prf	✗	CAS-11464	112d	low	norm.	Cor	✗	GH-720	4d	elev.	high	Prf	✗	CAS-8099	322d	high	norm.	Prv	✗
GH-234	68d	inter.	high	Prf	✗	CAS-5074	112d	low	low	Prf	✗	GH-677	20d	elev.	norm.	Prv	✗	CAS-11383	6d	elev.	norm.	Cor	✗
GH-254	68d	inter.	low	Prf	✗	CAS-5695	112d	inter.	low	Prf	✗	GH-1564	82d	elev.	critical	Adp	✗	CAS-14772	63d	elev.	norm.	Cor/Prf/Prv	✗
GH-259	68d	low	low	Cor	✗	CAS-13119	112d	low	urgent	Cor	✗	GH-1099	22d	elev.	norm.	Prf	✗	CAS-2412	320d	elev.	low	Adp	✗
GH-306	67d	elev.	norm.	Prf	✗	CAS-11127	112d	inter.	norm.	Cor	✗	GH-822	34d	elev.	norm.	Prf	✗	CAS-5062	5d	elev.	norm.	Prf	✗
GH-1826	65d	low	high	Cor	✗	CAS-11152	112d	low	norm.	Cor	✗	GH-855	92d	elev.	n/a	Cor	✗	CAS-7920	264d	elev.	norm.	Prf	✗

example to Gaffer's user guide. For Cassandra, most costly issues have lead times of 112/113 days, which is also almost twice the average lead time. These issues are mostly of low and normal priority, only CAS-13119 is of priority *urgent*. Most long lasting contributions to Cassandra are corrective changes, e.g., CAS-8290, which fixes failing system start due to erroneous log file handling, or CAS-10625, which fixes a bug that prevents large dates from being read from the database.

Remember, the most complex issues in Tab. I, are not filtered for lead time outliers. Consequently, multiple lead times of the most complex contributions are bigger than those of the most costly issues. For both systems, there are contributions that implement more than a single change type, e.g., GH-1884, CAS-13304, and CAS-14772 fix bugs, refactor, or implement new features in one contribution. Note, the issues with highest CC in Tab. I have a quite wide spread of lead times ranging from 14 to 302 days in Gaffer and from 21 to 544 days in Cassandra.

3) Analysis: Many of the contributions of long lasting issues are small. The many *low* and *intermediate* CC scores indicate small and concise changes. For example, *a)* commit 6c5ea1 resolves CAS-13119 ($t_{lead} \approx 112d$) by adding a single truth value as argument to a method call, which fixes a bug in a test. The issue is created on Jan. 11th 2017 together with a fix but it is first reviewed on May 3rd. *b)* Commit 4008e9 adds an extra parameter to a method call, which fixes inconsistent behavior between different versions of the database's query language (CAS-11152, $t_{lead} \approx 112d$). The issue is created on Feb. 10th 2016 together with a fix but it is first reviewed on May 31st. *c)* Commits 2f1d6c and 021df0 remove a link in Javadoc to resolve GH-259 ($t_{lead} \approx 68d$). The issue is created on Jun. 10th 2016, the first commits are from Aug. 16th, and a day later the issue is closed after review.

More complex contributions with long lead times follow similar distributions of work activity. For example, 27 commits add support for visibilities to Gaffer's Parquet store (GH-1099, created on Jul. 26th 2017). It is a non-trivial feature, since the concept of *visibility* depends on the concept of *authorizations*, both of which have to be implemented with corresponding tests. The commit that introduces most of the new functionality (2611e5) is a refactoring that decouples two packages by moving code across them. Subsequently, the new feature evolves into its final form, mainly via code structure reorganization (f37ed3, 2089f5), a rename refactoring (ea3223), addition and modification of tests (11c1a7, 36a682, bc85d1, 1fafec), etc. The first commit of the contribution is from Sep. 4th 2017, most development happens between Sep. 6th to 14th, and after review (Sep. 19th), the contribution is merged to the main branch and the ticket is closed on Oct. 4th.

For none of the issues with longest lead times, we can identify TCs as a cause. Long lead times appear to be caused mainly by the way work is organized and coordinated, see the long periods of inactivity illustrated above.

Two of the most complex contributions are, e.g., CAS-8099 and GH-538. All the complex contributions (Tab. I to the right) that we examine, follow a similar pattern: A bigger change that addresses the core of the issue is accompanied by many smaller changes that integrate the solution into its environment or that evolve it into a final state. Due to constrained space, we illustrate this only for GH-538 and CAS-8099.

Ticket GH-538 requires implementation of a second –more RESTful– web-API besides the current one. The contribution that resolves the issue consists of 32 commits, which introduce the new feature and refactor the previous API accordingly. The first commit (d025bb) introduces a major share of the code of the new API, where much seems to be generated by the API development tool Swagger. The initial commit is followed by many smaller changes, which add and adapt tests (e.g., 57f987, 02e127), adapt code styling (5c31a1), adapt configuration files (d06236, 9794e6), rename methods and classes and adjust imports (cf25a2, 9fc35d, 13e5b2), remove unnecessary code (418a73), add new examples (07d883), integrate results from code reviews (01884a, 9f9c66), or clean up code (e.g., 73bd96, 2e4546).

The work in this contribution is not complex due to a TC in the original web-API. It is complex due to the size of the new API and that it has to be integrated and harmonized with the existing solution. Alone the many tests for the new API and those that have to be adapted to accommodate two versions of an API render the contribution complex.

A large scale refactoring of Cassandra's storage engine (CAS-8099, created Oct. 10th 2014, resolved Aug. 28th 2015) is realized by one of the most complex contributions (a991b6). The author explains in a guide to the refactoring (832459) and in a blog post[11] that the original storage engine was processing tables as maps of ordered maps of binary data (illustrated as `Map<byte[], SortedMap<byte[], Cell>>`), where the byte

array contains partition keys and a `Cell` contains binary data and a timestamp for conflict resolution. He describes, that the original design was chosen due to its simplicity and since it *"was an almost direct match for the original API of Cassandra"*. After the implementation of the Cassandra Query Language (CQL) (CAS-1703 in version 0.8.0-beta1), developers realize that storage engine and CQL operate on different abstractions and therefore, the original version of the storage engine cannot effectively handle all expressible queries. Consequently, developers decide to refactor the storage engine to process maps of ordered maps of `Clustering`s and `Row`s (`Map<byte[], SortedMap<Clustering, Row>>`), where `Row`s aggregate more data than the previous *Cell*s and `Clustering`s aggregate more than the previous partition keys. The contribution that implements this refactoring is so complex, since the corresponding new abstractions (classes) have to be implemented (e.g., `Clustering.java` 66ce38 or `Row.java` 0ee9db) together with multiple other new super- and subclasses. The new abstractions have to be integrated into code that refers to previous abstractions (e.g., ea9ab6 or 07a665), the storage format has to be refactored (e.g., f53baa, c6a66e, or 24deb6), and in total 185 tests have to be removed (e.g., e4101c), adapted (e.g., 323a7f), or added (e.g., 154080).

The original version of the storage engine is not a TC. Originally, it implemented the most apt representation of a solution. First a later change (introduction of CQL), revealed a more appropriate new representation, which developers implement with the refactoring. For none of the most complex issues, we can identify TCs causing them. Instead, we observe that complexity is caused by the size of solutions in combination with required work for evolution or integration and respective tests.

4) Threats to Validity: We are no experts in either of the two systems. However, we believe that the main author who performed the manual inspection of contributions is sufficiently experienced to identify TCs. On top of eight years computer science education and more than five years teaching in programming and software engineering, he has more than five years of experience as professional software developer.

We examine only a tiny sample of 40 issues. A larger sample size might have yielded contributions containing TCs. Furthermore, the set of selected contributions depends on the ability of the two metrics lead time and CC score to accurately represent cost and complexity. However, the project's issue trackers do not provide more precise time information and we are not aware of an alternative to `ConCom` to automatically determine the complexity of contributions.

Filtering lead times for outliers before examining costly contributions may hinder identification of TCs. However, we decide a priori that we do not want to manually examine abandoned or low importance contributions, which cause extreme lead times [23]. If long lead times are caused by high complexity, then we identify such issues via CC scores as illustrated in Tab. I.

The risk of "overlooking" potential TCs or searching in the wrong place can be minimized by extending the set of cases, the amount of studied contributions, and the amount of investigators inspecting them. We plan that as future work.

Open-source systems are less-likely subject to tight schedules, which may cause TD, compared to proprietary software. Our case systems that are developed by a public agency or originally by a large company presumably resemble proprietary software in that regard.

IV. Related Work

Previous work on TD identification relies on the assumption that small code patterns (code smells) that can be detected via static code analysis have detrimental effects on maintainability. For example, a study of 745 systems with `CAST AIP` (and its 1,200 static analysis rules) [24], an analysis of Hadoop with `FindBugs`, `codevizard`, etc. [25] or a study of 66 open-source Java systems with `SonarQube` [26] all identify the number of certain code patterns and equate these to TD. Similarly, Tufano et al. [27] associate code smells to TD when showing that these are usually introduced on artifact creation and remain in them. However, it is unclear if code patterns that are identifiable by static analysis rules increase development cost and complexity over time. For example, Abbes et al. [28] show that first the combination of two object-oriented anti-patterns increase cost of maintenance whereas Sjøberg et al. [29] demonstrate a limited effect of code smells on increased maintenance effort. Therefore, we decide in this study to investigate the reverse, i.e., can TD be identified in most costly and complex contributions.

Recent work on self-admitted TD aims to identify TCs via matching text patterns in source code that hint at suboptimal solutions [30] or via correspondingly labeled tickets in issue trackers [31]. Xavier et al. [31] find, that design and architecture related refactorings are the main TD issues in issue trackers. That corresponds to our finding of large refactorings, i.e., resolution of Ur-TD [2], amongst the most complex contributions.

Researchers devise TD into various sub-forms, e.g., design debt, architecture debt, etc. [9]. However, it remains unclear what precisely constitutes these forms of debt and to which degree they are responsible for increased cost and complexity.

V. Conclusions & Future Work

In this paper, we search for TD by identifying *"technical compromises (TCs) that are expedient in the short term, but that create a technical context that increases complexity and cost in the long term"* [1]. We manually inspect 40 of the most complex and costly contributions with associated 217 commits from `Cassandra` and `Gaffer` to identify TCs.

We find that high lead times are caused mainly by organization and coordination of work and that high complexity is caused by the size and non-triviality of applied changes that require thorough testing and integration. Concerning our research question: in the studied contributions, we cannot identify any TCs, i.e., *"design or implementation constructs"* [1], that can explain their high cost and complexity.

In future work, we plan to extend this work by studying more contributions, by extending the set of case systems and by verifying our results with the actual developers of the case systems and other domain experts.

1) Implications for Research: Since we cannot find any TCs that are constituting TD, but since we can find that developers apply refactorings when the abstractions encoded in software and mental models diverge, we believe that *TD* should be used as described in Ur-TD [2], [3]. Ur-TD occurs *"when my ideas diverge from my code"* [2]. The drawback for research is, that Ur-TD *"is generally not detectable by static analysis [since] thoughts are stubbornly hidden from static analysis tools"* [2]. To identify diverging mental models and their manifestations in software, one would have to switch research focus more on how developers work and interact with software than analyzing software artifacts solely automatically.

2) Implications for Practice: Since *TD* is such an overloaded metaphor, see Sec. I, we believe that practitioners should refer directly to the software qualities that are of concern instead of relying on a metaphor. However, an important lesson is that when mental models and the abstractions that are encoded in software diverge too much from each other it is important to *"bite the bullet"*[12] and apply even large scale refactorings as, for example, done in CAS-8099.

REFERENCES

[1] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing technical debt in software engineering (dagstuhl seminar 16162)," in *Dagstuhl Reports*, vol. 6, no. 4. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[2] G. Fairbanks, "Ur-technical debt," *IEEE Software*, vol. 37, no. 04, 2020.

[3] W. Cunningham, "The wycash portfolio management system," 1992.

[4] C. Birchall, *Re-Engineering Legacy Software*. Manning Publications Co.

[5] S. McConnell, "Managing technical debt," Construx Software Builders, Inc, Tech. Rep., 2008. [Online]. Available: http://www.construx.com/uploadedfiles/resources/whitepapers/Managing%20Technical%20Debt.pdf

[6] J.-L. Letouzey, "The sqale method for managing technical debt definition document," 2016. [Online]. Available: http://www.sqale.org/wp-content/uploads/2016/08/SQALE-Method-EN-V1-1.pdf

[7] F. A. Fontana, R. Roveda, and M. Zanoni, "Technical debt indexes provided by tools: A preliminary discussion," in *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*. IEEE, 2016.

[8] P. Kruchten, R. L. Nord, I. Ozkaya, and D. Falessi, "Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 5, 2013.

[9] N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," *Information and Software Technology*, vol. 70, 2016.

[10] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "The financial aspect of managing technical debt: A systematic literature review," *Information and Software Technology*, vol. 64, 2015.

[11] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *Journal of Systems and Software*, vol. 86, no. 6, 2013.

[12] R. O. Spínola, A. Vetrò, N. Zazworka, C. Seaman, and F. Shull, "Investigating technical debt folklore: Shedding some light on technical debt opinion," in *2013 4th International Workshop on Managing Technical Debt (MTD)*. IEEE, 2013.

[13] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? manage it? ignore it? software practitioners and technical debt," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786848

[14] G. Kim, J. Humble, P. Debois, and J. Willis, *The DevOps Handbook:: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution, 2016.

[15] V. R. Basili, "Qualitative software complexity models: A summary," *Tutorial on models and methods for software management and engineering*, 1980.

[16] R.-H. Pfeiffer, "Automatically assessing complexity of contributions to git repositories," in *International Conference on the Quality of Information and Communications Technology*. Springer, 2021.

[17] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012.

[18] A. Lakshman and P. Malik, "Cassandra: Structured storage system on a p2p network," ser. PODC '09. New York, NY, USA: Association for Computing Machinery, 2009. [Online]. Available: https://doi.org/10.1145/1582716.1582722

[19] G. Upton and I. Cook, *Understanding statistics*. Oxford University Press, 1996.

[20] ISO Central Secretary, "Software engineering – software life cycle processes – maintenance," International Organization for Standardization, Geneva, CH, Standard ISO/IEC 14764:2006, 2006. [Online]. Available: https://www.iso.org/standard/39064.html

[21] Y. Guo, R. O. Spínola, and C. Seaman, "Exploring the costs of technical debt management–a case study," *Empirical Software Engineering*, vol. 21, no. 1, 2016.

[22] B. Kitchenham and S. L. Pfleeger, "Software quality: the elusive target [special issues section]," *IEEE software*, vol. 13, no. 1, 1996.

[23] R. Kikas, M. Dumas, and D. Pfahl, "Issue dynamics in github projects," in *International Conference on Product-Focused Software Process Improvement*. Springer, 2015.

[24] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the size, cost, and types of technical debt," in *2012 Third International Workshop on Managing Technical Debt (MTD)*. IEEE, 2012.

[25] N. Zazworka, C. Izurieta, S. Wong, Y. Cai, C. Seaman, F. Shull *et al.*, "Comparing four approaches for technical debt identification," *Software Quality Journal*, vol. 22, no. 3, 2014.

[26] G. Digkas, M. Lungu, A. Chatzigeorgiou, and P. Avgeriou, "The evolution of technical debt in the apache ecosystem," in *European Conference on Software Architecture*. Springer, 2017.

[27] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, 2017.

[28] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *2011 15Th european conference on software maintenance and reengineering*. IEEE, 2011.

[29] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, 2012.

[30] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014.

[31] L. Xavier, F. Ferreira, R. Brito, and M. T. Valente, "Beyond the code: Mining self-admitted technical debt in issue tracker systems," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020.

REFERENCED URLS

[1] https://www.atlassian.com/agile/software-development/technical-debt
[2] https://blog.crisp.se/2013/10/11/henrikkniberg/good-and-bad-technical-debt
[3] https://github.com/HelgeCPH/searching-for-techdebt
[4] https://gchq.github.io/gaffer-doc/
[5] https://github.com/gchq/Gaffer
[6] https://github.com/gchq/Gaffer/issues
[7] https://cassandra.apache.org
[8] https://issues.apache.org/jira/projects/CASSANDRA
[9] https://github.com/apache/cassandra
[10] https://github.com/HelgeCPH/searching-for-techdebt/blob/master/notebooks/Manual%20Search%20Protocol.ipynb
[11] https://www.datastax.com/blog/putting-some-structure-storage-engine
[12] https://issues.apache.org/jira/browse/CASSANDRA-8099