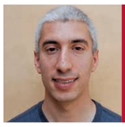
Editor: **Jeffrey C. Carver**
University of Alabama
carver@cs.ua.edu



Editor: **Lorin Hochstein**
Netflix
lhochstein@netflix.com

Technical Debt Problems and Concerns

Jeffrey C. Carver, Xabier Larrucea, Alexander Serebrenik, and Miroslaw Staron

THIS ARTICLE REPORTS papers about technical debt (TD) from the 2021 IEEE/Association for Computing Machinery (ACM) International Conference on technical debt (TechDebt'21), the 43rd IEEE/ACM International Conference on Software Engineering: Journal First Track (ICSE-JF'21), the 43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP'21), and the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR'21). Feedback or suggestions are welcome. In addition, if you try or adopt any of the practices included in the column, please send us and the authors a note about your experiences.

"Worst Smells and Their Worst Reasons," by Davide Falessi and Rick Kazman, investigates the type of "code bad smells" involved when defining a quality gate. The authors ran a survey with 71 developers where participants were asked about their experience and opinions on the smells. The purpose is to discriminate smells that have no reason to be introduced from smells that have a good reason to be introduced. After applying a filtering process to the SonarCloud for Java database, they ended up with 80 out of 133 potential worst smells. After analyzing their frequency, change-proneness, and severity associated with the worst smells, the authors identified the "worst reasons," such as "lack of knowledge," "improper language transfer assumption," and "intentional mistake" among others. The authors conclude that the worst smells seem to be similar to non-worst smells in terms of frequency and severity. This paper appears in TechDebt'21. Access it at https://bit.ly/PD-2022-May-01.

"On the Lack of Consensus Among Technical Debt Detection Tools," by Jason Lefever and colleagues, describes the analysis of six TD detection tools (Structure 101, SonarQube, Designite, DV8, Archinaut, and SCC) on 10 open source projects to determine the extent to which they agree upon the scores of key metrics and which files contain TD. The results show little agreement among the tools for identifying the most problematic files. Surprisingly, the tools show disagreements on basic measures such as size (LOC), complexity, and cycles. From this study, the authors suggest the use of measures combining structural and historical information for identifying files containing TD and the necessity of creating an industrial benchmark. This paper appears in ICSE-SEIP'21. Access it at https://bit.ly/PD-2022-May-02.

"Business-Driven Technical Debt Prioritization: An Industrial Case Study," by Rodrigo Rebouças de Almeida and colleagues, investigates the effects eight factors have on the prioritization of TD issues. The authors performed a five-month industrial evaluation of a business-driven approach to align business and technical criteria with the amount of TD observed. They examined how the proposed approach and other factors (e.g., stakeholder's perceptions) impacted TD prioritization. The approach includes a prioritization rule that defines how the TD items must be prioritized based on the business value perception captured by supported business processes and features as well as the product and service state. The approach visualizes the TD items according to the business and technical priority before the debt is repaid. The authors observed a misalignment between the prioritization of the TD and

Digital Object Identifier 10.1109/MS.2021.3133734
Date of current version: 18 April 2022

the prioritization rules when stakeholders of different domains are involved. The new approach adds a business perspective to TD prioritization and helps solve the conflicting business points of view among stakeholders. This paper appears in TechDebt'21. Access it at https://bit.ly/PD-2022-May-03.

"Experiences on Managing Technical Debt with Code Smells and AntiPatterns," by Jacinto Ramirez Lahti, Antti-Pekka Tuovinen, and Tommi Mikkonen, describes the experiences of a company in detecting code smells and antipatterns as sources of TD. The authors investigate recurring problems in system design, how to detect antipatterns, and whether static analysis and manual code inspection can identify code smells. Based on the interoperability and testing problems in the company, the proposed solution aims to systematically identify the antipatterns detected via code smells that block the software updates. The solution provides visualization of the data based on the metrics from two code inspection tools to confirm the code smells. In addition, linking the detected antipatterns to code smells provides guidance to software developers on how to fix design issues and improve the code structure during refactoring, thereby reducing the accrued TD. This paper appears in TechDebt'21. Access it at https://bit.ly/PD-2022-May-04.

"Preventing Technical Debt by Technical Debt Aware Project Management," by Marion Wiese, Matthias Riebisch, and Julian Schwarze, describes the definition and evaluation of a framework to support the prevention and repayment of TD. The novelty of this framework is that the repayment of intentionally contracted TD is part of the project plan. Developers record tasks related to TD as tickets in the project backlog so that they are tagged and handled depending on their category. The framework supports four categories of TD management 1) maintenance (e.g., refactoring of badly structured code); 2) maintenance project (i.e., maintenance tasks that require more than five days of development time, such as changing the underlying architecture); 3) TD (i.e., tasks that are related to internal software quality); and 4) deconstruction (i.e., a special type of TD ticket that cannot be processed during the project, such as the deconstruction of legacy code). The authors evaluated the framework in an IT unit of a German publishing house via a survey and a quantitative analysis of the TD tickets. The results showed that two-thirds of the survey respondents thought these tickets prevent TD and achieve a timely repayment of TD items. The main benefits provided by the framework are 1) evaluating different suboptimal and optimal solution options in advance prevents unnecessary TD; 2) by making project managers responsible for the TD accumulated, their willingness to incur TD decreases; and 3) the project and unit managers receive an overview of the accumulating TD while the project is running, thus supporting better decision making. This paper appears in TechDebt'21. Access it at https://bit.ly/PD-2022-May-05.

Self-Admitted TD

Self-admitted TD (SATD) is a set of TD that is reflected in annotations left by developers as comments in the source code or elsewhere that provide a reminder about portions of the software that contain TD. Researchers have presented several papers on this topic at the Mining Software Repositories (MSR) conference and the International Conference on Software Engineering (ICSE).

One of the important challenges in studies of SATD is the detection of SATD comments. In their MSR'21 paper, "Data Balancing Improves Self-Admitted Technical Debt Detection" (https://bit.ly/PD-2022-May-06), Murali Sridharan and colleagues study the impact of data balancing on detecting SATD. The authors observe with the appropriate data-level balancing traditional machine learning, i.e., XGBoost with SMOTE, outperforms the state-of-the-art deep learning approach when applied within the project (F1 0.83 versus 0.75), but not when applied across projects. In the latter case, deep learning exhibits better generalization capabilities (F1 0.70 versus 0.77). For all of the techniques considered, such words as "todo" and "hack" are features contributing most to the detection of SATD, reflecting that SATD code is "not ready (yet)." Rather than trying to improve the automatic detection of SATD, in "Identifying Self-Admitted Technical Debts with Jitterbug: A Two-Step Approach" (https://bit.ly/PD-2022-May-07), Yu and colleagues proposed Jitterbug, a two-step approach, that starts with identifying the "easy to find" SATD using a novel pattern recognition technique and then applies machine learning to assist human experts in manually detection of the remaining "hard to find" SATD. Empirical evaluation on a well-known data set of open source Java projects has shown that Jitterbug outperforms both baseline approaches using traditional machine learning techniques, and each one of the Jitterbug steps is performed in isolation. In their paper "Wait for It: Identifying 'On-Hold' Self-Admitted

ABOUT THE AUTHORS

JEFFREY C. CARVER is a professor in the Department of Computer Science, University of Alabama, Tuscaloosa, Alabama, 35487, USA. Contact him at carver@cs.ua.edu.

XABIER LARRUCEA is a project leader at TECNALIA, Basque Research and Technology Alliance, Spain, and an IEEE Senior Member. Contact him at xabier.larrucea@tecnalia.com.

ALEXANDER SEREBRENIK is a professor at Eindhoven University of Technology, Eindhoven, 5600MB, The Netherlands. Contact him at a.serebrenik@tue.nl.

MIROSLAW STARON is a professor in the Software Engineering Division, Chalmers University of Technology and the University of Gothenburg, Gothenburg, SE-412 96, Sweden. Contact him at miroslaw.staron@cse.gu.se.

Technical Debt" (https://bit.ly/PD-2022-May-08), Maipradit and colleagues focused on a special kind of SATD, on-hold SATD, i.e., debt that contains a condition to indicate that a developer is waiting for a certain event or an updated functionality that is implemented elsewhere. The best classifier designed by the authors achieves a mean precision of 0.63 and a mean recall of 0.68. Furthermore, for 35% of the on-hold SATD instances, the classifier was capable of correctly identifying the specific conditions that determine what the developer is waiting for, e.g., a date, a certain bug being fixed, or a certain version being released. Discussion in the three papers suggests that the choice of the most appropriate SATD-detection technique is likely to depend on the availability of time and resources: limited time or resources require higher precision that can be further improved through manual analysis similar to Jitterbug.

Another line of SATD-research focuses on categorization of SATD instances. For example, in their MSR'21 paper "Waiting Around or Job Half-Done? Sentiment in Self-Admitted Technical Debt" (https://bit.ly/PD-2022-May-09), Gianmarco Fucci and colleagues present a new taxonomy of eight categories: functional issues, poor implementation choices, wait(-ing), deployment issues, partially/not implemented functionality, testing issues, documentation issues, and misalignment. The majority of the comments were in the poor implementation choices category, followed by the partially implemented category. In addition to classifying the comments, the authors also studied the sentiment of the comments. The results show that the SATD about functional issues conveys more negative sentiment. Also, when developers have to wait for others, they tend to communicate negative sentiment. However, developers are more neutral when reporting poor implementation choices, misalignment, or documentation/testing issues. This relationship between the sentiment of the comment and the type of the comment can be indicative of where the SATD "hurts" software engineers the most. In "Self-Admitted Technical Debt in R Packages: An Exploratory Study" (https://bit.ly/PD-2022-May-10), Melina Vidoni uses an existing categorization of SATD instances and discusses the prevalence of different categories in 503 repositories of R packages. She found that SATD in R mostly records code debt (e.g., commented-out code). Another important type of SATD in R is algorithm debt, suboptimal implementations of algorithm logic, akin to SATD in research software and in sharp contrast with traditional software. Furthermore, a large amount of SATD in R was related to CRAN, the main repository for R packages. This type of SATD is aimed at stopping the execution of automated tests and bypassing CRAN checks, suggesting that developers view the policies of CRAN aiming at software quality improvement as obstacles rather than useful tools. Since R is a package-based environment, the study raises a concern related to the transitive effect of TD, i.e., data scientists who design solutions based on R should be aware that the foundational packages they use contain TD, and this TD might affect the quality of their solutions. In a similar way, the alarming tendency to bypass automatic tests and CRAN checks call for more careful evaluation of the quality of the packages prior to their adoption.

Finally, Liu and colleagues provide an example of a study of the evolution of SATD, i.e., its introduction and removal through the history of a project. The authors have focused on SATD in seven deep learning frameworks including TensorFlow, PyTorch, and Microsoft Cognitive Toolkit. In their article, "An Exploratory Study on the Introduction and Removal of Different Types of Technical Debt" (https://bit.ly/PD-2022-May-11), Liu and colleagues observed that developers introduce design debt most often during development, while they remove requirement debt most often, and remove design debt the fastest. However, developers introduce more design debt than they remove, leading to the accumulation of the design debt. The authors call for project management to take this into account when balancing the quality of the project and the proposal of new requirements. Moreover, while the previous studies observed that developers self-remove most of the SATD, the authors report a more refined observation. While indeed, most of test debt, design debt, and requirement debt are removed by the developers who introduced them, this is not the case for other types of debt. Compatibility debt and documentation debt are the most removed by the developers with more activities in the project. This calls for timely allocation of the documentation debt tasks to the developers with more activities in the project, and reimplementation of functionality within the broader system architecture as means of addressing compatibility debts associated with the release of qualified dependencies.

Digital Object Identifier 10.1109/MS.2022.3162503