A Survey of Self-Admitted Technical Debt

Giancarlo Sierra[a], Emad Shihab[a,*], Yasutaka Kamei[b]

[a] *Data-Driven Analysis of Software (DAS) Lab, Department of Computer Science and Software Engineering, Concordia University, Canada*
[b] *Principles of Software Languages Group (POSL), Kyushu University, Japan*

Abstract

Technical Debt is a metaphor used to express sub-optimal source code implementations that are introduced for short-term benefits that often need to be paid back later, at an increased cost. In recent years, various empirical studies have focused on investigating source code comments that indicate Technical Debt – often referred to as Self-Admitted Technical Debt (SATD). Since the introduction of SATD as a concept, an increasing number of studies have examined various aspects pertaining to SATD. Therefore, in this paper we survey research work on SATD, analyzing the characteristics of current approaches and techniques for SATD detection, comprehension, and repayment. To motivate the submission of novel and improved work, we compile tools, resources, and data sets made available to replicate or extend current SATD research. To set the stage for future work, we identify open challenges in the study of SATD, areas that are missing investigation, and discuss potential future research avenues.

Keywords: Self Admitted Technical Debt, Software Maintenance, Literature Survey, Source Code Comments

1. Introduction

As software undergoes its development and maintenance, developers are not always able to contribute code as required by specification. In 1992, Ward

*Corresponding author
Email addresses: g_sierr@encs.concordia.ca (Giancarlo Sierra), eshihab@encs.concordia.ca (Emad Shihab), kamei@ait.kyushu-u.ac.jp (Yasutaka Kamei)

Cunningham first introduced the metaphor of considering the "not-quite-right code" as a form of debt [1]. This came to be know as the *Technical Debt* (TD) metaphor, which explains the concept of delivering a solution that is not complete, temporary or sub-optimal; thus incurring in debt to obtain short-term benefits that have to be paid over the long-term with an increased cost. Developers experience different factors that can lead them to introduce technical debt, such as deadline pressure, existing low quality code, bad software process, or business reality [2]. Technical Debt can be introduced both consciously or unconsciously, and as found recently, developers tend to underestimate the consequences of repaying the debt, possibly leading to ever-growing problems [3]. Because of its clear importance to the software process and quality, an abundant amount of research has investigated TD [4, 5]. While in the past most studies focused on detecting and managing debt found in source code, the research scope has gradually grown to include additional software artifacts, e.g., documentation or requirements [6, 7].

In 2014, Potdar and Shihab [8] took a new research direction by conducting an exploratory study on source code comments that point to debt instances. The authors first referred to this phenomenon as *Self-Admitted Technical Debt* **(SATD)**. Their rational being that when developers consciously introduce debt (i.e., code that is either incomplete, defective, temporary, or simply sub-optimal) and acknowledge so in the form of comments they *self-admit* it. Brief examples of these comments are: *"TODO: - This method is too complex, lets break it up"* from ArgoUml, and *"Hack to allow entire URL to be provided in host field"* from JMeter [9, 10].

Potdar and Shihab extracted a large set of source code comments from 4 large open source systems and manually analyzed them to point at debt instances. As found by their investigation, this phenomenon occurs commonly in software systems [8]. Since then, a number of studies focusing on various aspects of SATD have emerged, exploring and improving on approaches and techniques to better identify, understand and manage SATD. This recent and increasing turn out of empirical work in this branch of TD denotes the importance given

to it by the Software Engineering community. Taking into consideration that this research track is fairly recent, the early efforts of current studies on SATD remain scattered in focus and face various challenges to overcome. We believe it is the right time to reflect on recent accomplishments in the area and examine open problems to pave the path for future work.

Therefore, this paper presents a survey of SATD studies from recent years, i.e., since the original ICSME paper that proposed SATD. Through our examination of the published papers, we find that the vast majority of SATD research work can be categorized into three categories: work focusing on the **detection** of SATD, work that aims to improve the **comprehension** of SATD, and work focusing on the **repayment** of SATD. Hence, we structure our survey to reflect these 3 main categories. Specifically, our paper provides an overview of past and current works in the detection, comprehension and repayment of SATD. Moreover, to support and promote further research in the domain, we identify potential future avenues for SATD research and discuss its current challenges. Throughout this survey we also point at available resources such as tools and datasets that can serve as foundations or baselines for new SATD studies. A compiled table with the published artifacts and online references from the surveyed work is available online[1].

The remainder of this paper is organized as follows: Section 2 describes the objectives, scope and literature selection for the survey; Section 3 analyses and compares the findings and contributions of current SATD studies; Section 4 goes over the possible future research avenues in this area and its challenges. Lastly, Section 5 presents the conclusions and limitations of the survey.

2. Preliminaries

This section details the scope and selection of studies for our survey. We also provide definitions for the terms we use throughout the paper. Finally, we

[1] http://das.encs.concordia.ca/uploads/SATD-Survey-Published-artifacts.pdf

present a high-level overview of the SATD literature published to date.

2.1. ***Scope and paper selection***

The focus of this paper is Self-Admitted Technical Debt as a sub-domain of Technical Debt. We clarify that work focusing entirely on Technical Debt (and not SATD specifically) is not in scope and refer our readers to recent literature that focused on that area (e.g., [4, 5]). To select the papers included in this survey we used both the references from known SATD research, and academic work available online through popular search engines, namely: Google Scholar, ACM, and IEEE. To begin, we chose the Potdar and Shihab's exploratory study as the cornerstone for this survey since it is the first to investigate the SATD phenomenon and remains as the most cited work in the area [8]. Hence our survey encloses work published since its release year (2014) until the compilation date of this survey (July 2018). We searched for all the papers that cited Potdar and Shihab's in the aforementioned online search engines using the keywords "SATD" and "Self-admitted Technical Debt", limiting the results to papers released since 2014. A complete list of the initial studies that we selected and did not select is available online[2].

Once we identified a paper related to SATD, we applied a snowball approach to find other relevant cited work [11]. We repeated this procedure for each work that cited Potdar and Shihab's, however, we did not find any other (new) SATD related papers that were not already included in the initial list or found by the search engines. Given that SATD is fairly new and due to the amount of mainstream work in the area we were able to select, we do not perform a systematic literature study; we leave that for the near future when the amount of SATD-related work justifies such kind of survey.

2.2. ***Definitions***

We classified the surveyed papers into 3 main categories tied to the life cycle stages of SATD, i.e., the sequence of phases that an instance of SATD

[2] http://das.encs.concordia.ca/uploads/SATD-Survey-Initial-Paper-Selection.pdf

goes through, from its introduction, to its evolution, and lastly its removal from a software system. Hence, the work is aligned along three categories: the Detection, Comprehension, and Repayment of SATD. We elaborate on what studies fall under each category below:

- **Detection** studies - those that focus on proposing, studying or improving: approaches, techniques, and tools to identify or detect instances of SATD.
- **Comprehension** studies - those that investigate the phenomenon of SATD itself and are dedicated to understand the life cycle of SATD. These studies encompass topics such as: introduction, diffusion, evolution, removal of SATD, or its relation with different aspects of the software process.
- **Repayment** studies - those that propose, validate, or replicate: approaches, techniques, and tools that seek to remove (i.e., fully repay) or mitigate (i.e., partially repay) SATD instances.

*2.3. **Overview of selected papers***

Given the scope and definitions above, Table 1 presents a chronologically ordered overview of the primary SATD studies. Note that those marked with a star (*) are studies whose focus in not dedicated to SATD, however, a relevant portion of them addresses SATD and presents findings related to its comprehension or detection, so we consider them within the primary group. Although related work without a direct contribution or finding on SATD is not considered within the selected group of papers, we mention and reference such work throughout this survey since they support the papers we selected or serve as links to potential future avenues in this area. In Table 1 we observe that 50% of the primary SATD papers focus on comprehension, 55% on detection, while only 10% focus on repayment. Note that 3 studies are classified as having 2 topics of focus, hence these percentages overlap. Regarding the paper's publication avenues, 60% of them are published in conferences, 20% in journals, and another 20% were presented in workshops.

Table 1: Overview of primary SATD studies.

Author(s) [Reference], Year	Title	Venue	Venue Type	Focus
Potdar & Shihab [8], 2014	An Exploratory Study on Self-Admitted Technical Debt.	ICSME	Conference	Comprehension, detection
Maldonado & Shihab [9], 2015	Detecting and Quantifying Different Types of Self-Admitted Technical Debt.	MTD	Workshop	Comprehension, detection
Freitas Farias *et al.* [12], 2015	A Contextualized Vocabulary Model for Identifying Technical Debt on Code Comments.	MTD	Workshop	Detection
Wehaibi *et al.* [13], 2016	Examining the Impact of Self-admitted Technical Debt on Software Quality.	SANER	Conference	Comprehension
Freitas Farias *et al.* [14], 2016	Investigating the Identification of Technical Debt Through Code Comment Analysis.	ICEIS	Conference	Detection
Bavota & Russo [15], 2016	A Large-Scale Empirical Study on Self-Admitted Technical Debt.	MSR	Conference	Comprehension
Vassallo *et al.* [16], 2016	Continuous Delivery Practices in a Large Financial Organization.	ICSME	Conference	Comprehension*
Kamei *et al.* [17], 2016	Using Analytics to Quantify the Interest of Self-Admitted Technical Debt.	TDA	Workshop	Comprehension
Mensah *et al.* [18], 2016	Rework Effort Estimation of Self-Admitted Technical Debt.	TDA	Workshop	Repayment, detection
Ichinose *et al.* [19], 2016	ROCAT on KATARIBE: Code Visualization for Communities.	ACIT	Conference	Detection*
Maldonado *et al.* [10], 2017	Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt.	TSE	Journal	Detection
Palomba *et al.* [20], 2017	An Exploratory Study on the Relationship between Changes and Refactoring.	ICPC	Conference	Comprehension*
Miyake *et al.* [21], 2017	A Replicated Study on Relationship Between Code Quality and Method Comments.	ACIT	Conference	Comprehension*
Maldonado *et al.* [22], 2017	An Empirical Study on the Removal of Self-Admitted Technical Debt.	ICSME	Conference	Comprehension
Zampetti *et al.* [23], 2017	Recommending when Design Technical Debt Should be Self-Admitted.	ICSME	Conference	Detection
Mensah *et al.* [24], 2018	On the Value of a Prioritization Scheme for Resolving Self-Admitted Technical Debt.	JSS	Journal	Repayment

Huang *et al.* [25], 2018	Identifying Self-Admitted Technical Debt in Open Source Projects using Text Mining.	EMSE	Journal	Detection
Liu *et al.* [26], 2018	SATD Detector: A Text-Mining-Based Self-Admitted Technical Debt Detection Tool.	ICSE	Conference	Detection
Zampetti *et al.* [27], 2018	Was Self-Admitted Technical Debt Removal a real Removal? An In-Depth Perspective.	MSR	Conference	Comprehension
Yan *et al.* [28], 2018	Automating Change-level Self-admitted Technical Debt Determination.	TSE	Journal	Detection

3. Analysis and comparison of current work

In this section we first go over the techniques, tools, and approaches presented by current research work in SATD. We first present work that focused on identifying instances of debt, then we present empirical studies that have studied the phenomenon to understand it, and finally contributions that aim to manage and repay it. A list of the software projects studied by the surveyed work is available online[3], along with how each study validates TD.

3.1. ***Detection of SATD***

In the life cycle of SATD, debt instances are first introduced by developers into the source code; thus naturally, the first step to study this phenomenon is to identify it. In the past, several studies have focused on source code comments, their management, and co-evolution with code; while others focused on the identification and management of Technical Debt [29, 30, 31, 4, 5]. However, these studies did not investigate or relate the presence of technical debt within the content of comments. Inspired by such previous work, Potdar and Shihab were the first to look at source code comments to identify technical debt, and introduced the term of *Self-Admitted Technical Debt*, referring to code that is either incomplete, defective or temporary, and that is knowingly introduced by developers [8]. 7 different approaches to detect SATD have appeared in literature since; 6 of them identify SATD at the file level looking at the revision history of a repository, while 1 approach aims to detect SATD at the change level. In this subsection, we present the 6 approaches that work at the file level divided in two groups: i) those approaches that are based on the identification of textual patterns in comments, which we name "pattern-based approaches"; and ii) those that apply more advanced and automated techniques, such as machine learning classifiers or natural language processing, which we name "machine learning approaches". Lastly, we present the only approach that focuses on

[3] http://das.encs.concordia.ca/uploads/SATD-Survey-Studied-Projects.pdf

detecting SATD at the change level, and a comparison between the surveyed approaches.

3.1.1. ***Pattern-based Approaches***

As a first step in SATD identification at the file level, Potdar and Shihab extracted 101,762 source code comments from 4 large open source systems using the *srcML* toolkit [32], and manually read through them to expose patterns that indicate SATD. In total, the authors identified 62 patterns and made them publicly available to enable further research [33]. Some examples of the identified patterns are: *hack, fixme, is problematic, this isn't very solid, probably a bug, hope everything will work, fix this crap.* Using these patterns, their study found that SATD can exist in up to 31% of files; a finding that triggered further research in this domain.

For the remaining of this survey, we will refer to the usage of these 62 patterns as the **pattern-based detection** approach. This approach allows for an easier SATD identification than simple manual inspection of comments, which is time-consuming and requires expertise. However, because these patterns resulted from analyzing 4 projects only, they may not generalize well if used to detect SATD in other software systems, compromising the accuracy of the approach. Additionally, in case the set of patterns has to be extended, additional effort must be spent manually inspecting source code comments from different projects and surfacing new patterns that can be used for detecting TD in comments.

Following up to the previous findings, Maldonado and Shihab manually inspected the comments of another 5 open source projects, this time however, with a motivation to explore the different types of SATD contained in them [9]. They found 5 main types of SATD: design, defect, documentation, requirement and test debt (See 3.2). Instead of *srcML*, the tool *JDeodorant* was used to parse the extracted comments [34]. Four filtering heuristics were introduced to remove irrelevant comments, which are: a) removing license comments; b) aggregating consecutive single-line comments; c) removing commented source code; and d) removing Javadoc comments. To ensure these heuristics do not

filter out SATD instances, comments containing task-reserved words ("todo", "fixme", or "xxx") were not removed. The implementation of these heuristics proved to reduce the amount of comments to analyze manually by 77% on average, easing detection efforts. To contribute with the identification of specific types of SATD, the output dataset of classified comments by types was made publicly available to the community [35].

Motivated to facilitate the detection of SATD using the pattern-based approach, Ichinose *et al.* extended their proposed code visualization tool *ROCAT*, which renders the source code of a project as city-like virtual reality environments to support SATD [19]. With this visualization model, buildings are constructed for each source file, their dimensions are based on software product metrics, and SATD instances are rendered based on comments that contain the patterns surfaced by Potdar and Shihab [8]. This provides developers with a high-level view of a system's source code that includes visual cues of SATD instances, removing the need of reading comments to visualize where SATD occurs in their source code. *Rocat* was integrated with *Kataribe* [36], a Git hosting service; with this, any project registered on *Kataribe* can benefit from *Rocat's* visualization capabilities.

An alternative and extension to the pattern-based detection approach was later proposed by Freitas *et al.* [12], who introduced **CVM-TD**, a Contextualized Vocabulary Model for Identifying TD of different types in source code comments. This model relies on identifying word classes, namely: nouns, verbs, adverbs, and adjectives that are related to Software Engineering terms and code tags used by developers such as "TODO" [37]. The goal of applying the CVM-TD model, which can be automated, is to obtain a subset of comments that will likely contain SATD. The proposed vocabulary focuses on words that can be systematically related to each other and then mapped to different types of TD as defined by Alves *et al.* [7, 38]. To validate CVM-TD, an empirical study was conducted on Apache Lucene and JEdit, from which comments were extracted using *eXcomment* [39], a tool that uses an Abstract Syntax Tree to store useful comment-related information and filtered with heuristics similar to the ones

proposed by Maldonado and Shihab [9]. The empirical evaluation of the model showed a considerable difference in the comments returned by the model and the ones validated to contain SATD. This finding suggested a low detection performance and pointed at the need to enhance how the word classes are mapped to different types of SATD to improve the model.

Later in 2016, Freitas *et al.* [14] conducted an additional experiment on CVM-TD to characterize its overall accuracy and the factors that influence its detection. This time, the CVM-TD model was applied to ArgoUML; the output comments were given to 3 researchers with expertise in TD to create an oracle of comments that actually indicate TD. The same output was also given to 32 Software Engineers with varied experience in the field and different English reading levels to flag those suggesting TD. The experiment found that the English reading skills of the participants influenced their identification of TD, but this was not affected by their experience. Based on the TD oracle, the CVM-TD model's output served experienced and non-experienced developers alike, allowing them to have an accuracy on average of 0.673 when detecting TD comments; a better performance than previously reported [12]. The experiment also requested participants to highlight the patterns that induced marking a comment as TD, which surfaced common patterns and TD indicating comments to extend the vocabulary of CVM-TD [40, 41]. Note that in both empirical studies by Freitas *et al.* (i.e., [12, 14]), the authors do not explicitly refer to source code comments that aid in the detection of TD as SATD, nevertheless, we consider both studies within scope as they study this same precise phenomenon.

Mensah *et al.* proposed the use of text mining in SATD detection [18]. Their approach aims to estimate the effort needed to resolve SATD (See 3.3) and is composed of 5 phases. The first 3 phases of the approach are aimed at the extraction, detection and classification of SATD; it is built on top of a pattern-based approach and a dictionary from the dataset of comments classified into different SATD types published by Maldonado and Shihab [9]. We will refer to this approach as **Text mining**. Improving from the pattern-based approach, this one first preprocesses comments to remove special punctuation characters

and stop words; however, this introduces a drawback. Removing punctuation characters such as *!* or *?* can potentially take away semantic meaning from comments; i.e., the removal of a simple question mark could alter the meaning or intention of a developer's comment. Moreover, no filters such as the heuristics proposed and used previously (e.g.,[9, 14]) were applied to reduce preprocessing.

3.1.2. ***Machine learning Approaches***

Moving towards more advanced SATD detection approaches at the file level, Maldonado *et al.* used NLP techniques to automatically identify design and requirement SATD from source code comments [10]. We will refer to this approach as **NLP detection**. The authors extracted, filtered, and manually classified a dataset of 62,566 source code comments from 10 open source projects into 5 different types of SATD: design, test, defect, documentation and requirement debt. This dataset combined 29,473 comments extracted from 5 open source projects, and 33,093 others extracted from additional 5 projects in previous work [9]. With it, the authors trained an NLP maximum entropy classifier (Stanford Classifier) focusing on requirement and design SATD, as they are the most recurrent debt types, making up more than 90% of the SATD comments [9]. The NLP classifier generates a set of feature words that contribute positively or negatively to the classification of a comment. A 10 fold cross-project validation training on 9 projects and testing on the remaining showed that the NLP detection achieved an accuracy that surpassed the previous pattern-based detection. For design debt, the classifier scored an average F1-measure of 0.620, 0.403 for requirement debt, and 0.636 disregarding debt types. The study also presented a top-10 lists of textual features that can be directly used to identify SATD in approaches that do not rely on NLP techniques. These features were found to differ among each other, indicating that developers use distinct vocabularies to admit different kinds of SATD.

Training an NLP classifier can be expensive since it relies on a manual classification of comments, however, Maldonado *et al.* showed that to achieve 90% of the classifiers performance, approximately 23% of the SATD comments were

needed for training, which eases the replication of this approach. To enable further research on SATD, the full resulting dataset of manually classified comments and their resulting NLP classification was made publicly available [42].

The most recent SATD detection technique was presented in 2017 by Huang *et al.* [25], who proposed an approach to automatically detect SATD using text mining and a composite classifier. We will refer to this as the **Ensemble text mining** approach. Its root concept is to determine if a comment indicates SATD or not (without focusing on SATD types) based on training comments from different software projects. For this, the authors leveraged a dataset of 212,413 comments classified by Maldonado *et al.* from 8 open source projects [10, 9]. This approach preprocesses comments by tokenizing, removing stop-words and stemming their descriptions to obtain textual features. Feature selection (Information Gain) is then applied to detect the top 10% most useful features to predict the label of a comment, indicating if it contains SATD or not. Multiple sub-classifiers are trained with a Naive Bayes Multinomial (NBM) technique to determine the label of a comment based on the number of contributing features they have. A composite classifier takes the vote per comment of each sub-classifier to reach a final classification. Several aspects of the ensemble text mining performance were evaluated in terms of F1-score. The approach was benchmarked against the pattern-based and NLP detection of SATD, finding that it performed better than both, had a superior runtime performance, and also required a small portion of comments for training.

The ensemble text mining approach was implemented very recently by Liu *et al.* as an Eclipse plugin named *SATD Detector* [26] to facilitate the detection and management of debt instances directly from an IDE environment. *SATD Detector* parses the source code of a project when it is loaded or edited and applies the ensemble text mining approach to detect and report SATD instances along with their respective locations. This completely automates the detection of SATD with a built-in classifier that can be used out of the box to leverage the best-performing SATD detection technique.

From a different SATD detection perspective, Zampetti *et al.* proposed **TE-**

DIOuS (Technical Debt Identification System), a machine learning approach that recommends to developers when they should self-admit design TD [23]. Instead of analyzing comments, the idea is to leverage source code level features. When a developer adds new code, the approach can analyze it and recommend if it should be flagged (i.e., to be self-admitted as debt) or not. TEDIOuS' identification capabilities relies on readability and structural metrics extracted with a srcML-based tool, and the warnings raised by PMD and CheckStyle, 2 static analysis tools.

TEDIOuS was evaluated using the classified comments of 9 projects from the dataset made available by Maldonado *et al.* [10]. Since these comments were detected at the file level, a matching of comments to the method level was required for TEDIOuS features' scope. Different classifiers were tested with balanced and unbalanced training data using cross validation within a project and across all studied projects. TEDIOuS achieved it best performance using a Random Forest classifier, with a cross-project prediction precision of 67%, 55% recall, and an accuracy of 92%. The features related to readability and structural metrics used by TEDIOuS were found to have a major contribution in recommending design SATD. When compared against DECOR [43], a smell detector tool which leverages different code features, the SATD recommending performance of TEDIOuS proved to be superior.

3.1.3. ***Change-level detection***

All previous SATD detection studies aimed to identify debt instances at the file level. Yan *et al.* [28] proposed a novel approach to automate the detection of SATD at the change level. The idea is to catch the introduction of SATD when a software change occurs, instead of inspecting if a file that was changed previously contains SATD. The authors built a determination model using a Random Forest classification with data labeled from comment analysis, and features extracted from source control repositories. The data labeling leverages an enhanced version of the dataset made available by Maldonado *et al.* [22]; it contains 100,011 manually classified software changes of 7 open source projects,

where each change is labeled as TD-introducing or not; where change is considered TD-introducing when the resulting file version is the first to contain SATD. A total of 25 change features were extracted from the source control repository of the studied systems to characterize each change. These features were divided into 3 dimensions in the study: 16 for the *diffusion* of a change (i.e., amount of changed LOC, files, subsystems, programming languages), 3 for its *history* (i.e., information of the changed files and the developers who made the change), and 6 for its *message* (i.e., information extracted from the change logs).

The proposed model was evaluated performing a stratified 10-fold cross validation repeated 10 times for each of the 7 studied projects. This evaluation considered 2 performance measures: AUC (area under the receiver operating characteristic curve), and Cost-effectiveness, analyzed by controlling the amount of changed LOC inspected by the model. To contrast the model's performance, 4 other baseline models were studied: Random Guess, Naive Bayes, Naive Bayes Multinomial, and Random Forest (the last 3 models used a classification based on change messages only). The study results showed that the proposed model achieves a better performance in terms of AUC (0.82) and cost-effectiveness (0.80) when compared to baseline models, being able to detect more TD-introducing changes across a wide range of changed LOC to inspect. When investigating the importance of the extracted features, the results indicate that all 3 dimensions significantly improve the performance of the compared models, and that the *diffusion* dimension is of most influence when determining TD-introducing changes. The performance achieved by this SATD detection approach is not contrasted with others in Table 2 as the SATD detection of these approaches occur in to different stages of development and thus they differ in nature. The reported performance of the change-level SATD detection is also reported in terms of AUC and not as an F1-score.

3.1.4. ***Comparison and limitations of current approaches***

The original pattern-based approach for SATD detection has the benefit of being simple to replicate with a fixed set of patterns to match against textual

Table 2: Average accuracy benchmark of SATD detection approaches, as reported by Huang *et al.* [25].

Detection Approach	**Reported F1-score**
Pattern-based	0.123
NLP	0.576
Ensemble text mining	0.737

comments. However, it has the drawback of leading to up 25% of false positives, as found by Bavota and Russo [15]. Although the text mining and CVM-TD approaches later built on top of the pattern-based approach with added heuristics, both are still affected by an underlying accuracy problem and are more complex to replicate. These early approaches lead to SATD datasets that supported the creation of more accurate and automated techniques, such as the NLP, TEDIOUS, and ensemble text mining approaches, which implement machine learning. While TEDIOUS recommends when to self-admit technical debt, it scopes to design debt only and is not comparable with other approaches as it looks at source code instead of comments to base its recommendations. In contrast, the NLP detection and ensemble text mining approaches focus of finding SATD in comments with good accuracy. While the NLP approach is limited to detect design and requirement only, the ensemble text mining approach disregards SATD types, and thus, is a more effective all-around approach when looking for SATD in a software repository. Another benefit when compared to other detection approaches, is that this last one does not require manual inspection of comments, which aside from being time consuming is prone to human error. Furthermore, since it was recently implemented as an IDE tool (SATD Detector plugin), it can now be used as a practical solution to detect SATD during or after development.

A performance comparison between SATD detection approaches is presented in Table 2 as benchmarked by Huang *et al.*. This comparison uses the average accuracy values for detecting SATD disregarding debt types [25]. The Text mining and CVM-TD approaches are not included in the benchmark as their TD detection performance were not reported in [14, 18]. Note than the F1-score

for the NLP approach in Table 2 is lower than the value reported by Maldonado *et al.* (0.636) [10]; in either case, the performance of the ensemble text mining approach is higher.

As a recap, the studies that focused on the detection of SATD have contributed with approaches that evolved from simple manual inspection of comments to more complex automated approaches that identify SATD instances accurately, removing manual steps. Similarly, the text mining approach, evolved the classification of SATD types from manual inspection to an automated possibility. In Table 3 we overview the main findings and contributions per SATD detection study, the number of studied projects, and the technique for comment extraction, where applicable. Note that the visualization technique presented in the study by Ichinose *et al.* (i.e.,[19]) can be applied to multiple projects, thus no specific one is studied and no comment extraction is performed. A similar case happens with the contribution by Liu *et al.* (i.e.,[26]), which is a tool implementing the approach proposed by Huang *et al.* (i.e.,[25]). From the observations made in this section, we consider the ensemble text mining detection approach (implemented in the SATD Detector tool) to be the most promising approach to enable future SATD research. Due to its performance and practicality, we believe this tool will promote the detection of SATD, and the compilation of richer datasets to improve the validity of SATD studies.

Table 3: Overview of main contributions per SATD detection study.

Author(s) [Reference], Year	**Main Contribution(s) / Finding(s)**	**Studied Systems**	**Comment Extraction**
Potdar & Shihab [8], 2014	Pattern-based detection approach. SATD exists in 2.4% to 31% of files.	4	scrML-based
Maldonado & Shihab [9], 2015	Dataset of classified SATD comments per type. Filtering heuristics.	5	Jdeodorant
Freitas Farias *et al.* [12], 2015	CVM-TD detection approach.	2	eXcomment
Ichinose *et al.* [19], 2015	City-like code and SATD visualization in a virtual reality environment.	N/A	N/A
Freitas Farias *et al.* [14], 2016	Set of Patterns and comments for TD identification in comments.	1	eXcomment
Mensah *et al.* [18], 2016	Text mining detection/classification approach.	4	*Not reported*
Maldonado *et al.* [10], 2017	NLP Detection approach. Data set of classified SATD.	10	JDeodorant
Huang *et al.* [25], 2017	Ensemble text-mining detection approach.	8	NLP Dataset
Zampetti *et al.* [23], 2017	TEDIOuS approach for recommending when to self-admit TD.	9	NLP Dataset
Liu *et al.* [26], 2018	Eclipse plugin to automatically detect SATD.	9	NLP Dataset
Yan *et al.* [28], 2018	Change-level SATD detection approach.	7	Relies on [22]

3.2. ***Comprehension of SATD***

Different studies have been conducted to understand the SATD phenomenon throughout its life cycle, while others investigated its repercussion on the software process itself. A better understanding of SATD enables researchers and practitioners to develop approaches that can be used to manage it. One of the first efforts towards understanding SATD were given by Potdar and Shihab; in their exploratory study they tried to understand the occurrence of SATD, why it is introduced into software projects, and how much of it is removed after its introduction [8]. By using a pattern-based detection in 4 software projects, SATD was found to be common, happening in 2.4% to 31% of studied system's files. Regarding the introduction of SATD, Potdar and Shihab investigated how the experience of developers, time to release pressure, or the complexity of changes induced the addition of debt. Contrary to what was expected, they found that experienced developers introduced most of the SATD, while tight deadlines and change complexity did not affect its introduction. In relation to SATD removal, they found that the majority of SATD is removed in the immediate next release.

3.2.1. ***Types of SATD***

Once SATD was found to be a common phenomena, Maldonado and Shihab [9] decided to quantify and classify the different types of SATD that exist in software projects. In a previous study, Alves *et al.* [7] classified Technical Debt into 13 different types and proposed indicators to identify each of them. Based on these types, Maldonado and Shihab manually analyzed 33,093 comments and classified them, observing that 5 types of SATD existed in source code (design, defect, documentation, requirement, and test debt) [35]. We include brief examples of debt comments as classified by Maldonado and Shihab[9] to help understand the detected SATD types:

- **Design debt:** *"/*TODO: really should be a separate class */"* from ArgoUml.
- **Defect debt:** *"Bug in the above method"* from Apache JMeter.
- **Requirement debt:** *"//TODO no methods yet for getClassname"* from Apache Ant.
- **Documentation debt:** *"**FIXME** This function needs documentation"* from Columba.

- **Test debt:** *"//TODO enable some proper tests!!"* from Apache JMeter.

The remaining 8 types of TD defined by Alves *et al.* [7] were not found since they are not likely to appear in source code comments but in other artifacts. As explained by the authors, build debt for example, would appear in build files and not in the inspected comments extracted from Java files. The quantification results of the study revealed that from over 33 thousand analyzed comments, 7.42% of them (2,457) contained SATD. Regarding the quantification per type, the majority (42% to 84%) of SATD found was design debt, followed by requirement debt, making up 5% to 45% of the debt instances. Defect, documentation, and test debt accounted for less than 10% of the classified SATD cases when combined.

3.2.2. ***Large-scale studies***

To broaden the understanding of the phenomenon, Bavota and Russo [15] conducted a large-scale empirical study in 159 software systems (120 from the Apache ecosystem and 39 from the Eclipse ecosystem) aiming to make a differentiated replication of the initial findings by Potdar and Shihab [8]. Using the pattern-based detection they investigated the diffusion of SATD in open source systems and its evolution across the change history of the studied subjects to see if: i) it increases or decreases over time, ii) how long it remains in the system, iii) how frequently it is fixed, and iv) who introduces or fixes SATD.

A closer look at a statistically significant sample of SATD cases revealed that in contrast with previous findings by Maldonado and Shihab [9], code debt was the most occurring debt type making up 30% of the cases, against a lower 13% for design debt. Furthermore, this inspection surfaced that over 25% of the comments flagged by the pattern-based detection were false positives. Bavota and Russo [15] looked at the introduced, removed and unaddressed SATD comments in the projects' change history and observed that it increases over time because of debt instances being added but not addressed. Although 57% of SATD was found to be removed from source code, it has a long survivability, lasting for more than 1,000 commits on average before being fixed. Inspecting the removed

SATD showed that 63% of the time, the developer who removes a debt instance is the same as the developer who introduced it; while in the remaining 37% of cases the developers who fix SATD have higher experience than those who introduce it. The study also measured the partial correlation between quality code metrics (Coupling, Complexity and Readability) and SATD, but found it is not significant between any of them, an in-line observation with Potdar and Shihab [8].

3.2.3. ***Impact of SATD***

Instead of looking at code quality metrics which were validated to have no clear correlation with SATD, Wehaibi *et al.* [13] investigated the relation between SATD and the quality of software by looking at defects. Their study used a pattern-based detection to find files that contain SATD in the repositories of 5 open source systems; in total 10.17% to 20.14% of files were labeled as SATD files. To find defects, the change history of every subject was mined to find patterns that indicate defects, such as: "defect", "bug ID", "fixed issue #ID". With both datasets the study investigated: i) the amount of defects in files with and without SATD; ii) the percentage of SATD related changes that are defect-inducing; and iii) if changes that involve SATD files are more difficult than the ones that do not. The authors compared the percentage of defects in SATD vs non-SATD files, and the amount of defects in SATD files before and after the debt introduction, however, they found no clear relation between defects and SATD. To observe if SATD-related changes introduced future defects they made use of a bug-introducing change identification algorithm proposed by Sliweski, Zimmerman, and Zeller (SZZ) [44] as implemented in Commit Guru [45], and found that they are less prone to introduce future defects. Lastly, using 4 change difficulty measures from previous work, the authors found that SATD-related changes were more difficult than non-SATD ones.

To clarify the relation between non-SATD source code comments and software quality, Miyake *et al.* [21] partially replicated the study by Wehaibi *et al.* [13] on 4 open source projects. Their results agreed with the previous study,

finding that SATD files are more prone to undergo a defect fix. However, they also found that the mere existence of comments at the method or file level is related to more future code fixes, even if they do not contain SATD. Nevertheless, SATD comments were found to be more effective to identify fix-prone files and methods than comments without SATD.

3.2.4. ***Removal of SATD***

Most of the previous comprehension studies targeted the introduction, diffusion, and evolution of SATD. Early studies also looked into the final stage of SATD, its removal [8, 15], however, their efforts were not dedicated specifically to the removal of debt. Recently, Maldonado *et al.* [22] studied precisely this, investing i) how much SATD is removed from source code; ii) who removes it; iii) how long does it remain in a system; and iv) what leads to removal activities. The authors studied 5 well-commented systems written in Java as subjects, which vary in size, domain and number of contributors. Their study showed that 40.5% to 90.6% of SATD was removed from the study subjects. Comparing the name and e-mail address of the developers who introduced and removed SATD from the repository commits showed that on average 54.5% of SATD is self-removed, i.e., by the same developer who introduced the debt; confirming the finding first presented by Bavota and Russo [15]. A comparison between SATD that is self-removed and the one removed by others indicated that the second survives for longer in a system. Concerning the median survival of SATD, the study found that it can remain in a system between 18 to 172 days before being removed. A survey to developers was also conducted in order to understand what activities lead to the removal and introduction of SATD [46]. The survey revealed that developers mostly add SATD to track potential bugs or code that needs improvement; similar to the finding of Vassallo *et al.* [16]. On the other hand and in-line with the observation by Palomba *et al.* [20], participants indicated that they mostly remove SATD when fixing bugs or adding features, but not as a dedicated activity.

After the above observations on the removal of SATD, Zampetti *et al.* [27]

conducted an in-depth quantitative and qualitative empirical study on the removal of SATD. The authors built on top of the previous work of Maldonado *et al.* [22] by analyzing their same dataset, focusing on the underlying circumstances of SATD removal from source code. The study investigated how much debt was removed by accident, i.e., without the intention of resolving debt, but as a collateral of software evolution. The study found this was the case for 25% to 60% of SATD comments, as they were removed due to full class or method removals. However, 33% to 63% of SATD comments were removed as part of a change in their corresponding method. In the remaining cases, comments were removed without any actual code change, possibly due to developers removing an outdated SATD comment or accepting the debt's risk. By computing the cosine similarity between SATD comments and commit messages, the authors looked for documented evidence of SATD removals, finding that only about 8% of the cases mentioned addressing the debt or justifying why it is not required to do so anymore. The study also looked at the types of changes that happen along SATD removals, finding that developers often apply complex changes across the code but also specific ones related to method (API) calls and control logic. On removals associated with API changes, 55% belong to the addition or editing of features; while removals linked to conditional changes are more diverse but often involve the removal of code.

3.2.5. ***SATD Interest***

Several works shed light over the SATD life cycle stages, nevertheless, none had yet proposed a concrete way to measure the interest of SATD, i.e., the increased cost of repaying debt in the future. A recent study by Kamei *et al.* [17] focused on determining a way to measure this precisely. It investigated if the debt instances incur a positive interest (i.e., they become more difficult to repay), negative interest (i.e., become less difficult to repay), or no interest over time. Sixteen different code complexity metrics were first evaluated and then filtered down to 2, namely LOC and Fan-In. The LOC measure was used since it is highly correlated with most of the metrics evaluated initially, excluding

Fan-In, thus both were selected. This work performed a case study on Apache JMeter and used JDeodorant to extract raw comments, which were then filtered and manually validated to contain SATD. To measure the incurred interest, the study scoped to the method-level for the SATD instances and computed the LOC and Fan-In metrics at the moment of their introduction and removal. Results showed that for both measures, 42% to 44% of SATD incurs a positive interest; while around 8% to 13% and 42% to 49% has negative and no interest, respectively. The interest quantification of SATD could be used as a proxy to estimate the effort needed to repay it. In the following subsection we go over additional studies with this focus.

3.2.6. ***Other empirical findings related to SATD***

Two recent studies presented observations related to SATD while looking at different aspects of software development. While studying the continuous integration practices of 152 practitioners from a large financial organization (ING Netherlands), Vassallo *et al.* [16] showed that 88% of the practitioners mentioned self-admitting their bad implementations of code through comments (i.e., SATD). This reflects the practical importance of addressing SATD during the development process. In an alternate scenario, while investigating the relation between 3 types of code changes and refactoring activities, Palomba *et al.* [20] noticed that in feature-introducing changes, often the refactored files had SATD on its previous version. Because of this, they applied a pattern-based detection to spot SATD in each refactoring activity. Their results showed that 46% of the classes had a SATD instance before being refactored, and 67% of the commits that refactored code also removed a debt instance. This indicates that developers mostly apply refactorings to repay existing debt before introducing new features into their source code.

To summarize the findings and contributions of the above comprehension studies, we present them in Table 4, along with the number of studied software systems. Since comprehension studies rely on a SATD detection approach, we also include them along with the comment extraction tools used in Table 4. Note

that most comprehension studies used a manual inspection or a pattern based detection, while only one study implemented a NLP approach. Certainly this relates to the ease of replicating different detection approaches, but it compromises their effectiveness of studying the phenomenon. We expect and encourage future studies to implement the more recent and accurate SATD detection approaches.

Table 4: Overview of main findings per SATD comprehension study.

Author(s) [Reference], Year	Contribution(s) / Finding(s)	Studied Systems	Detection Approach	Comment Extraction
Potdar & Shihab [8], 2014	- More experienced developers tend to introduce more SATD. - Time to release pressure and change complexity do not play a major role in SATD introduction. - Most of SATD is removed in the next immediate next release.	4	Manual	srcML based
Maldonado & Shihab [9], 2015	- Identified 5 different types of SATD. - The most common type of SATD is design or requirement debt.	5	Manual	JDeodorant
Bavota & Russo[15], 2016	- There is no clear relation between code quality metrics and SATD. - The amount of SATD increases over time in a system. - Code debt occurs more than design and requirement debt. - SATD lasts for a long time in source code before being removed. - About 57% of SATD is removed from source code; 63% of the time by who introduced it, 37% by other experienced developers.	159	Pattern based	srcML
Wehaibi *et al.* [13], 2016	- There is no clear relation between defects and SATD. - TD files defectiveness increases after the introduction of TD. - SATD changes lead to less future defects than non-SATD changes. - SATD changes are more difficult to perform. - Empirical evidence that TD affects the development process by making it more complex. - The impact of SATD is not related to defects, rather in making	5	Pattern based	Ad-hoc. Python

	future changes more difficult to perform.			
Vassallo *et al.* [16], 2016	- Most practitioners self-admitting their bad implementations of code through comments.	N/A	N/A	N/A
Kamei *et al.* [17], 2016	- 42% to 44% of SATD incurs in positive interest. 8% to 13% and 42% to 49% has negative and no interest, respectively.	1	Manual	JDeodorant
Miyake *et al.* [21], 2017	- SATD comments are more effective than non-SATD comments when identifying fix-prone files and methods.	4	Pattern based	Ad-hoc, Java
Palomba *et al.* [20], 2017	- Developers mostly apply refactorings to repay SATD before introducing new features.	3	Pattern based	srcML
Maldonado *et al.* [22], 2017	- SATD can remain in a system between 18 to 172 days. - Developers mostly remove SATD when fixing bugs or adding features, and use SATD to track future bugs and bad implementation areas. - Most of SATD is removed, and most of it is also self-removed.	5	NLP detection	srcML based
Zampetti *et al.* [27], 2018	- A large percentage of SATD removals are accidental. - Only around 8% of SATD removals are documented in commits. - While removing SATD, developers mostly apply complex changes but also, specific ones to method calls and conditionals.	5	NLP detection	srcML based

3.3. ***Repayment of SATD***

Previously, we surveyed work that contributed towards the comprehension of SATD on its removal (section 3.2.4), and interest growth (section 3.2.5). Although those studies explain how and who removes SATD, and propose a way to measure the growth or decline of SATD over time, they do not propose approaches towards managing or repaying debt. In this section we go over studies that tackle this problem.

As a subset of Technical Debt, the ultimate goal of studying SATD is to propose approaches that focus on removing it from a system, i.e., repaying the admitted debt. In this regard, a couple of recent studies have presented techniques to estimate the effort and prioritize the resolution of SATD. In 2016, Mensah *et al.* [18] proposed an approach to estimate the rework effort needed to resolve SATD, measured in LOC. The authors used the text mining approach to identify debt instances in 4 open source projects and classify them by type with a dictionary derived from the work by Maldonado and Shihab [9]. The measure of estimated rework effort is calculated giving term weights to debt instances based on their frequency of SATD indicators, i.e., one of the patterns found by Potdar and Shihab [8], and expressed the average commented LOC per SATD-prone file (files that contain comments with debt indicators) in a system. The study found that on average, an effort of between 13 and 32 commented LOC need to be addressed per SATD-prone file. This estimated effort fluctuates based on the type of debt to be addressed, with documentation requiring the least amount of effort, and design debt needing the most.

More recently, Mensah *et al.* [24] extended their rework effort estimation study and combined it with a 6-step SATD prioritization scheme. This new approach aims to inspect SATD instances and classify them by how urgently they need to be addressed and estimate the rework effort they require. Similarly to their previous work, this estimation is computed in a multi-phased approach, where initial steps handle the extraction of comments, identification and classification of debt instances into their types using the text mining approach. Before computing the rework effort estimation, the extracted comments were manually

categorized based on their textual indicators as: i) *major* if they are urgent, or *minor* if they can wait; ii) *complex* based on their difficulty, and *significant* based on their importance; iii) *expected* if the task is pending, and *expedited* if it denotes a rushed or poor implementation. SATD instances that should be prioritized were marked as *vital few* tasks or as *trending-many* tasks, and assigned a possible cause of introduction. Along with the proposal of a repayment approach, this work also presented interesting empirical findings, showing that 31% to 39% of SATD comments are major tasks, and 58% to 69% are minor; while most of the major tasks are complex to resolve for developers. Among the possible causes for SATD introduction, the study found 4 which are the most prominent, being: code smells (23%), complicated and complex tasks (22%), inadequate code testing (21%), and unexpected code performance (17%). Regarding the effort required for the resolution of vital few tasks, i.e., those that should be prioritized, developers would need to address 10 to 25 commented LOC per SATD file.

The concept of classifying the SATD comments into different classes that indicate how difficult, important, and urgent they are can serve as a great contribution to deciding which debt to resolve first. However, is important to note that for both of the above works on repayment output a result in commented LOC, which might not be intuitive for developers or managers, nor the best or only measure to estimate effort or prioritize debt resolution. In either way, both approaches compel the most recent in SATD repayment.

4. Future of SATD Research

In this section, we present promising research avenues based on gaps and opportunities we observe in current studies and discuss the challenges to overcome in order to advance the state of the art. The ideas and calls to actions presented throughout this section are new proposals deduced from our observations, which we support with related literature.

4.1. Future challenges in SATD detection

4.1.1. Improving validity

SATD detection can benefit from improved validity, future work should enrich existing datasets and expose new ones using state of the art detection and classification approaches. Since TD can also be self-admitted in other software artifacts, such as commit messages or issue comments, richer datasets should not be limited to SATD found in source code comments only. We expand on these ideas below:

- **Richer datasets.** As we see in the work surveyed in Section 3, most of recent work relies on data from design and requirement SATD [10, 23, 25, 27, 28]. This originates in the dataset made available by Maldonado and Shihab [9], where design and requirement debt was detected far more frequently than other debt types. This limits approaches such as the NLP and ensemble text mining approaches to be restricted on classifying debt instances in all existing types. Using a tool such as SATD Detector can support the creation of larger datasets with more instances of the rarer SATD types. Such datasets can then be complemented by artificial balancing techniques to enable better classification approaches. Another challenge with current datasets is that they are scarce, and limited in size and diversity of projects they contain. Huang *et al.* [25] found that cross-project training increased the performance of identification classifiers. Thus, SATD detection approaches will benefit of having richer datasets to train on.

- **Detection in other software artifacts.** The majority of work surveyed in Section 3.1 detected SATD through source code comments. There are other software artifacts that contain extracts of human interaction and communication, such as issue messages, commit messages, or even discussions in git repositories. These artifacts can also hold text where technical debt is self-admitted by developers. Dai and Kruchten [47] studied the possibility of detecting TD with issue comments, finding that although

developers do not explicitly mention TD inside issues, they do so indirectly. Their study surfaced over 114 useful key words that can be used to detect different types of TD from the description and summaries of issues. This is a similar finding to the patterns surfaced by Potdar and Shihab [8] for SATD. Bellomo *et al.* [3] also investigated the existence of TD indicators within issues messages and found that developers are aware of the concept of TD, and they refer to it when filing issues. This might indicate that technical debt is also self-admitted in issue messages.

Nowadays there is a plethora of repositories that can be mined to investigate the occurrence and diffusion of SATD in alternate software artifacts. One example is *JIRA*, a repository presented by Ortu *et al.* [48] which contains data from the Jira Issue Tracking System. It consists of over one thousand open source projects with 700 thousand issue reports, and 2 million issue comments. As its authors suggest, it can be mined to retrieve information about TD, and thus potentially, SATD. The investigation of how much debt found within issues is also self-admitted by developers and the usefulness of this approach remains as future work. Considering the above software artifacts for an approach such as the SATD change-level determination proposed by Yan *et al.* [28] could also yield a promising future. Including features extracted from different software artifacts can complement the 3 dimensions studied by Yan *et al.* to extend the set of features taken from source code and change history, potentially resulting in improved TD determination models. As detecting SATD at the change level presents different benefits to software developers in contrast to detection at the file level, there is broad potential and room for further investigation on the topic.

Call to action:

- *Mine larger sets of software repositories from different domains to produce richer SATD datasets.*

- *Study the presence of SATD in other software artifacts, such as the messages and descriptions of issues and commits.*

4.1.2. ***Improving traceability and adoption***

In Section 3.1, we surveyed several approaches for SATD detection with different characteristics and techniques that allow them to achieve performances that surpass their predecessors. Each has an application, as well as points in favor and against that facilitate their replication. For instance, one could argue that manual detection and pattern-based approaches (see Section 3.1.1) are the easiest to replicate, however, doing is time-consuming and relies on human expertise. On the other hand, automated approaches that use machine learning are scalable but rely on a training dataset to achieve a comprehensive performance (see Section 3.1.2). Future work should aim to facilitate the replication of detection approaches to promote their adoption, and to develop tools to increase the admittance, quality, and traceability of SATD. One materialized example for this is SATD Detector, where the ensemble text mining was implemented as a tool ready for use in development time. Certainly, any approach or technique that can be offered as a tool is the best proxy to improve the traceability and adoption of SATD. We describe actionable ideas that can support this based on opportunities we observe from previous related work below:

- **Visualization tools.** Alongside improved detection techniques, both researchers and practitioners can always benefit from tools that implement them. An interesting avenue comes from the visualization approach presented by Ichinose *et al.* [19]; city-like views in a virtual reality environment combined with an automated detection and classification approach could provide a highly intuitive interface for SATD identification and management. Visualization tools can also be extended to estimate the repayment effort of detected SATD with an approach such as the one proposed by Mensah *et al.* [24]. In this scenario visual cues could point at debt that can be repaid in the source code. The development of a tool that can display where SATD is located and offer an estimation of the effort

required to address it would strongly enable developers to manage and repay SATD in their repositories.

- **Annotation of comments.** While classifying grammar smells, Stijlaart and Zaytsev [49] pointed at the "Shortage Smells" as missing pieces of grammar. As a subset of this, "Debt" smells were defined to happen when comments clearly denote debt but are missing an annotation that will facilitate its traceability, such as *"TODO"* or *"FIXME"*. In this case, an approach or tool that adds these annotations would solve grammar smells by self-admitting the technical debt. For this to be feasible, researches can use one of the more recent SATD detection approaches and add special annotations to comments that are missing them. In this way, SATD will be easier to trace by developers using IDEs that support the tracking of these annotations.

- **Reduction of false positives.** Another important challenge is to reduce false positives in SATD detection. One of the issues with the approaches analyzed in Section 3 is that most of them look at comments directly, disregarding the source code in scope. For example, the pattern-based approach was found to produce over 25% of false positives [15]. Although more advanced detection approaches have been presented, they still focus on source code comments only. Such approaches might find cases indicating debt that was already repaid but its corresponding self-admitted annotation was never removed. On this regard, Sridhara proposed a technique to validate the up-to-date status of comments that include *ToDo* annotations [50]. This is a hybrid approach that considers both, source code and comments. Future work can improve on such technique and extend it to work on any comment that indicates SATD, and not only those with *ToDo* annotations. Moreover, as seen in section 3.1.2, TEDIOUS is the only detection approach that inspects source code instead of comments to recommend when design technical debt should be self-admitted. Certainly, a way to mitigate false positives in future SATD detection efforts

can emerge from using a hybrid approach that inspects the source code in scope and comments of a debt instance.

Call to action:

- *Develop tools that enable a categorized visualization of SATD to support its management.*
- *Develop a detection approach that adds annotations to debt comments that are missing them.*
- *Develop detection approaches that inspect and analyze both, comments and source code for improved accuracy.*

4.2. ***Future and challenges in SATD comprehension***

To deepen the understanding of SATD, research work should identify observations on this phenomena that apply across projects and can be generalized. In Section 3.2, we surveyed work that studied large sets of systems or specifically tried to diversify their subjects in domain and programming language [15, 13]. Nevertheless, a clear challenge to overcome is that most findings and contributions on SATD (see Table 4) and its effects in software development came from studying open source systems that were mostly written in Java (see the software projects studied by the surveyed work in Section 3). Future research should extend to investigate proprietary software or systems that are written in various programming languages. This will aid towards the generalization of current findings or contrast new observations in different scenarios and environments. Similar to previous efforts such as the empirical SATD study by Bavota and Ruso on 159 projects [15], important findings on SATD should be investigated in large scale to confirm they generalization.

We remark that the studies covered by this survey consider a scenario where identifying the introduction of TD is valuable for the development process, and where the management and repayment of TD are desired practices. More importantly, in the case of SATD, the assumed scenario is one where the use of source

code comments is intrinsic to the development process. However, this may not generalize to all software development, as it depends on the used methodologies and policies in place. An example may be a case of proprietary software were the introduction of comments is not allowed or exceptional. Note that in our survey, we did not find any SATD study that worked on proprietary software systems. Investigating the relation between the introduction, management, and repayment of SATD in different development methodologies remains as future work. This will help to achieve a more general and thorough comprehension oh the phenomena. Below, we present actionable ideas for future research to broaden the comprehension of SATD:

- **Examine other kinds of impact.** Previous work has investigated the impact of SATD on software quality, but only in the scope of software defects [13]. As Wehaibi *et al.* showed, defects do not seem to have a direct relationship with SATD. However, this is the only finding on the impact of SATD among the papers that focus on the comprehension of the phenomenon (see Section 3.2). Therefore, we believe that future work should seek a deeper understanding of different aspects in which SATD can impact the development process. We observe the opportunity to investigate on the impact of SATD in aspects such as: effort in future maintenance and evolution (e.g., code decay), the ability of a system to adapt to new technologies or changes in process, and even the socio-technical impact of SATD.

- **Qualitative classifications.** So far, source code comments that point to TD have been classified following the categories defined by Alves *et al.* [7], such as in the classification work on SATD by Maldonado and Shihab [9]. This is a high-level classification of the comments as they indicate what the debt is about. Another perspective is to investigate their implication in the development process. As an example, the comment: *"//Re-initialising a global in a place where no-one will see it just // feels wrong. Oh well, here goes."* from ArgoUML was classified by Maldonado and Shihab as

design debt [9]. This classification does not inform the developers about its implication; perhaps it implies a feature addition, a bug fix or another software maintenance tasks. A study using such level of taxonomy was presented by Panichella *et al.* [51], who classified mobile app user reviews into useful categories related to maintenance tasks. Replicating such taxonomy in the area of SATD can provide developers with better insight on the implications of SATD. Improving the overall understanding of the debt instances on their systems to support their management.

Call to action:

- *Investigate SATD in proprietary software systems and in various programming languages (other than Java).*
- *Investigate the impact of SATD on various software engineering aspects, such as maintainability and evolution.*
- *Produce a qualitative taxonomy that reflects the implications of SATD in software maintenance tasks.*

4.3. ***Future challenges in SATD repayment***

4.3.1. ***Quantitatively prioritizing repayment***

Proposing approaches and techniques to mitigate and repay debt is of utmost importance in SATD research. Studies in the past few years have shed light on the importance of this phenomena, but they have mostly focused on detecting and understanding SATD, rather than directly pursuing its resolution. Merely 11% of the studies that we surveyed focus on repayment efforts, thus, there is much work to be done in this area. We present the main challenges to overcome in SATD repayment below:

- **Effort Estimation.** SATD repayment contributions have scoped to prioritize its resolution based on the estimated effort for addressing a debt instance[18, 24]. However, this approach outputs an estimation value in commented LOC, which might not be the best, and certainly not the only

measure to estimate effort [52]. Undoubtedly, how to measure effort remains a challenge to overcome and a milestone to reach when deciding which debt to repay first.

- **Prioritization of SATD.** Certainly prioritizing SATD repayment has to be part of future research work. Given a set of instances of SATD in a project, developers need a approach to recommend which debt to resolve first. Thus, approaches that measure the growth of debt instances and their resolution cost must be combined. Akbarinasaji and Bener [53] presented the idea of adding TD as a financial obligation that can be recorded as type of liability in a balance sheet. To achieve this, TD needs to be identified, quantified, and monetized. Although an approach to monetize SATD has not been presented, some efforts have already taken a step forward, such as the quantification SATD interest by Kamei *et al.* [17]. We argue that SATD prioritization is one of the most important challenges that require attention in this domain, hence we plan to focus on extending existing research work and proposing novel ideas towards this goal in the immediate future.

- **Acceptance of SATD.** Not all SATD has to be repaid, fixing a shortcut or hack in the source code can be more expensive than beneficial. A proper measurement of TD repayment effort can aid developers to decide whether to live with the debt and its risks or not. Such repayment estimation has to consider the potential evolution of the debt as it can incur in positive interest over time [17]. Future work should study the extent of SATD acceptance in software systems and under which conditions.

Call to action:

- *Investigate new measures to estimate the effort required to repay SATD.*
- *Develop approaches to prioritize the repayment of SATD.*
- *Investigate to which extent SATD is or can be accepted in software systems.*

4.3.2. ***Integrating the repayment of SATD***

The activity of repaying TD has to be integrated into the software process. To this matter, the development of new tools and techniques that motivate and facilitate the repayment of SATD is required. We present two ideas that can facilitate this below:

Gamification of SATD repayment. SATD research not only needs to give answers on which debt instance to address first, but also to ease and promote the culture of resolving debt instances as part of the normal activities in the development process. In this regard, the use of mechanisms such as Gamification [54], i.e., the application of game-like features in non-game context could be of benefit. Gamification has increasingly been proving its usefulness to motivate, accelerate and ease human productivity and it has already been studied in the context of software development (i.e.,[55, 56]), thus, it has the potential to support and motivate the repayment of SATD among developers.

Identify who introduced the debt. Knowing which developer self-admitted debt in the first place and the rationale for doing so is important. Siegmund [57] suggested supporting the task of identifying developers who are responsible for a component, and helping them communicate with others who have introduced SATD. Such scenario would require an approach that identifies SATD and determines the developer who introduced it. Enabling a channel of communication between developers can shed light into the rationale behind a debt instance to support is repayment. However, it can be problematic as a debt-introducing developers may no longer be available. Thus, its applicability is limited by the phase at which SATD is managed.

Call to action:

- *Study the usage of gamification techniques to motivate the repayment of SATD.*
- *Complement SATD detection approaches by identifying who introduced the debt to enable communication between developers, facilitating repayment.*

5. Conclusions and limitations

We surveyed empirical research work in the arising topic of SATD, which has developed rather quickly in recent years. This literature survey has been performed on studies related to self-admitted technical debt, as defined by the exploratory study of Potdar and Shihab [8]. We used this study as the cornerstone for our survey and applied snowballing to find related work from it. Although we complemented the lookup for SATD-related work with results from academic search engines, we found no studies that focus on SATD that were not originally found during the snowballing process. Thus, the papers encompassed in this survey are limited to those released after 2014 and until the compilation of this survey in July of 2018. The selected papers are also limited to those returned by the search engines and keywords we used, and only to those that mainly focus on studying SATD (see Section 2.1).

From our survey subjects, we observe how researchers have evolved current approaches from manual observations to automated techniques for detecting and classifying debt instances, and have advanced the overall understanding of the SATD phenomenon in the software development process. Naturally, the focus of SATD studies was clustered in detecting the presence of debt, and understanding its life-cycle. Once detection approaches were accurate and replicable, the focus switched to studying how SATD grows over time and how it is removed from software repositories. We certainly observe a lack of studies focusing on the repayment and management of SATD, which is of critical importance. However, we also notice researchers stepping towards efforts to manage and repay SATD. To this extent, our work highlights several of the challenges to overcome in the area, and presents various promising avenues for future studies based on the gaps and opportunities seen in current research work. Our survey compiles the tools and datasets that can be used as a foundation to motivate and facilitate the submission of novel and improved approaches for managing and ultimately, repaying SATD.

We believe SATD will continue receiving attention in the field the upcoming

years. As an immediate future, we plan on centralizing our efforts on how to prioritize the resolution of SATD.

ACKNOWLEDGMENTS

This research was partially supported by JSPS KAKENHI Grant Numbers JP18H03222.

References

[1] W. Cunningham, The wycash portfolio management system, Proceedings on Object-oriented Programming Systems, Languages and Applications 4 (2) (1992) 29–30.

[2] E. Lim, N. Taksande, C. Seaman, A balancing act: what software practitioners have to say about technical debt, IEEE software 29 (6) (2012) 22–27.

[3] S. Bellomo, R. L. Nord, I. Ozkaya, M. Popeck, Got technical debt? surfacing elusive technical debt in issue trackers, in: Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16, IEEE, 2016, pp. 327–338.

[4] Z. Li, P. Avgeriou, P. Liang, A systematic mapping study on technical debt and its management, Journal of Systems and Software 101 (2015) 193–220.

[5] N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, C. Seaman, Identification and management of technical debt: A systematic mapping study, Information and Software Technology 70 (2016) 100–121.

[6] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, I. Gorton, Measure it? manage it? ignore it? software practitioners and technical debt, in: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE '15, ACM, 2015, pp. 50–60.

[7] N. S. R. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, R. O. Spínola, Towards an ontology of terms on technical debt, in: Proceedings of the 6th International Workshop on Managing Technical Debt, MTD '14, IEEE, 2014, pp. 1–7.

[8] A. Potdar, E. Shihab, An exploratory study on self-admitted technical debt, in: Proceedings of the 30th International Conference on Software Maintenance and Evolution, ICSME '14, IEEE, 2014, pp. 91–100.

[9] E. Maldonado, E. Shihab, Detecting and quantifying different types of self-admitted technical debt, in: Proceedings of the 7th International Workshop on Managing Technical Debt, MTD '15, IEEE, 2015, pp. 9–15.

[10] E. Maldonado, E. Shihab, N. Tsantalis, Using natural language processing to automatically detect self-admitted technical debt, IEEE Transactions on Software Engineering 43 (11) (2017) 1044–1062.

[11] C. Wohlin, Guidelines for snowballing in systematic literature studies and a replication in software engineering, in: Proceedings of the 18th international conference on evaluation and assessment in software engineering, EASE '14, ACM, 2014, pp. 38:1–38:10.

[12] M. A. de Freitas Farias, M. G. de Mendonça Neto, A. B. da Silva, R. O. Spínola, A contextualized vocabulary model for identifying technical debt on code comments, in: Proceedings of the 7th International Workshop on Managing Technical Debt, MTD '15, IEEE, 2015, pp. 25–32.

[13] S. Wehaibi, E. Shihab, L. Guerrouj, Examining the impact of self-admitted technical debt on software quality, in: Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering, Vol. 1 of SANER '16, IEEE, 2016, pp. 179–188.

[14] M. A. de Freitas Farias, J. A. Santos, M. Kalinowski, M. Mendonça, R. O. Spínola, Investigating the identification of technical debt through code com-

ment analysis, in: Proceedings of the 18th International Conference on Enterprise Information Systems, ICEIS '16, Springer, 2016, pp. 284–309.

[15] G. Bavota, B. Russo, A large-scale empirical study on self-admitted technical debt, in: Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16, ACM, 2016, pp. 315–326.

[16] C. Vassallo, F. Zampetti, D. Romano, M. Beller, A. Panichella, M. D. Penta, A. Zaidman, Continuous delivery practices in a large financial organization, in: Proceedings of the 32nd International Conference on Software Maintenance and Evolution, ICSME '16, IEEE, 2016, pp. 519–528.

[17] Y. Kamei, E. d. S. Maldonado, E. Shihab, N. Ubayashi, Using analytics to quantify interest of self-admitted technical debt, in: Proceedings of the 1st International Workshop on Technical Debt Analytics, TDA '16, CEUR-WS, 2016, pp. 68–71.

[18] S. Mensah, J. Keung, M. F. Bosu, K. E. Bennin, Rework effort estimation of self-admitted technical debt, in: Proceedings of the 1st International Workshop on Technical Debt Analytics, TDA '16, CEUR-WS, 2016, pp. 72–75.

[19] T. Ichinose, K. Uemura, D. Tanaka, H. Hata, H. Iida, K. Matsumoto, Rocat on kataribe: Code visualization for communities, in: Proceedings of the 4th Intl. Cong. on Applied Computing and Information Technology/3rd Intl. Conf. on Computational Science/Intelligence and Applied Informatics/1st Intl. Conf. on Big Data, Cloud Computing, Data Science & Engineering, ACIT-CSII-BCD '16, IEEE, 2016, pp. 158–163.

[20] F. Palomba, A. Zaidman, R. Oliveto, A. De Lucia, An exploratory study on the relationship between changes and refactoring, in: Proceedings of the 25th International Conference on Program Comprehension, ICPC '17, IEEE, 2017, pp. 176–185.

[21] Y. Miyake, S. Amasaki, H. Aman, T. Yokogawa, A replicated study on relationship between code quality and method comments, in: Applied Computing and Information Technology, Springer, 2017, pp. 17–30.

[22] E. Maldonado, R. Abdalkareem, E. Shihab, A. Serebrenik, An empirical study on the removal of self-admitted technical debt, in: Proceedings of the 33rd International Conference on Software Maintenance and Evolution, ICSME '17, IEEE, 2017, pp. 238–248.

[23] F. Zampetti, C. Noiseux, G. Antoniol, F. Khomh, M. Di Penta, Recommending when design technical debt should be self-admitted, in: Proceedings of the 33rd International Conference on Software Maintenance and Evolution, ICSME '17, IEEE, 2017, pp. 216–226.

[24] S. Mensah, J. Keung, J. Svajlenko, K. E. Bennin, Q. Mi, On the value of a prioritization scheme for resolving self-admitted technical debt, Journal of Systems and Software 135 (2018) 37–54.

[25] Q. Huang, E. Shihab, X. Xia, D. Lo, S. Li, Identifying self-admitted technical debt in open source projects using text mining, Empirical Software Engineering 23 (1) (2018) 418–451.

[26] Z. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, S. Li, Satd detector: A text-mining-based self-admitted technical debt detection tool, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, ICSE '18, ACM, 2018, pp. 9–12.

[27] F. Zampetti, A. Serebrenik, M. D. Penta, Was self-admitted technical debt removal a real removal? an in-depth perspective, in: Proceedings of the 15th International Conference on Mining Software Repositories, MSR '18, ACM, 2018, p. 11 pages.

[28] M. Yan, X. Xia, E. Shihab, D. Lo, J. Yin, X. Yang, Automating change-level self-admitted technical debt determination, IEEE Transactions on Software Engineering (2018) 1–1.

[29] L. Tan, D. Yuan, G. Krishna, Y. Zhou, /* icomment: Bugs or bad comments?*, in: Proceedings of the 21st Symposium on Operating Systems Principles, Vol. 41 of SOSP '07, ACM SIGOPS, 2007, pp. 145–158.

[30] M. A. Storey, J. Ryall, R. I. Bull, D. Myers, J. Singer, Todo or to bug, in: Proceedings of the 30th International Conference on Software Engineering, ICSE '08, ACM/IEEE, 2008, pp. 251–260.

[31] B. Fluri, M. Wursch, H. C. Gall, Do code and comments co-evolve? on the relation between source code and comment changes, in: Proceedings of the 14th Working Conference on Reverse Engineering, WCRE '07, IEEE, 2007, pp. 70–79.

[32] M. L. Collard, M. J. Decker, J. I. Maletic, Lightweight transformation and fact extraction with the srcml toolkit, in: Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM '11, IEEE, 2011, pp. 173–184.

[33] A. Potdar, E. Shihab, Satd patterns, http://users.encs.concordia.ca/~eshihab/data/ICSME2014/satd.html, (Accessed on 01/12/2018) (2015).

[34] N. Tsantalis, T. Chaikalis, A. Chatzigeorgiou, Jdeodorant: Identification and removal of type-checking bad smells, in: Proceedings of the 12th European Conference on Software Maintenance and Reengineering, SCAM '08, IEEE, 2008, pp. 329–331.

[35] A. Potdar, E. Shihab, Satd mtd data, http://users.encs.concordia.ca/~eshihab/data/MTD2015/MTD_15_data.zip, (Accessed on 01/12/2018) (2015).

[36] K. Fujiwara, H. Hata, E. Makihara, Y. Fujihara, N. Nakayama, H. Iida, K. Matsumoto, Kataribe: A hosting service of historage repositories, in: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR '14, ACM, 2014, pp. 380–383.

[37] M. A. de Freitas Farias, M. G. de Mendonça Neto, A. B. da Silva, R. O. Spínola, Type of technical debt x software engineering nouns, http://homes.dcc.ufba.br/~marioandre/page/cvm-td/TD_SENouns.pdf, (Accessed on 01/15/2018) (2015).

[38] M. A. de Freitas Farias, M. G. de Mendonça Neto, A. B. da Silva, R. O. Spínola, Cvm-td vocabulary, http://homes.dcc.ufba.br/~marioandre/page/cvm-td/vocabulary_comments.pdf, (Accessed on 01/15/2018) (2015).

[39] M. A. de Freitas Farias, M. G. de Mendonça Neto, A. B. da Silva, R. O. Spínola, Excomment tool, http://goo.gl/9Mgl9m, (Accessed on 01/15/2018) (2015).

[40] M. A. de Freitas Farias, J. A. Santos, M. Kalinowski, M. Mendonça, R. O. Spínola, Cvm-td - most selected patterns by participants, https://drive.google.com/file/d/0BwwEbWFwapG1UlNZbG5lekN1UUk/view, (Accessed on 01/23/2018) (2016).

[41] M. A. de Freitas Farias, J. A. Santos, M. Kalinowski, M. Mendonça, R. O. Spínola, Cvm-td - comments by ratio, https://drive.google.com/file/d/0BwwEbWFwapG1Y2hRaEt1bGFGa2s/view, (Accessed on 01/23/2018) (2016).

[42] E. Maldonado, E. Shihab, N. Tsantalis, Replication package for using natural language processing to automatically detect self-admitted technical debt, https://github.com/maldonado/tse.satd.data, (Accessed on 01/16/2018) (2017).

[43] N. Moha, Y. Gueheneuc, L. Duchien, A. L. Meur, Decor: A method for the specification and detection of code and design smells, IEEE Transactions on Software Engineering 36 (1) (2010) 20–36.

[44] J. Śliwerski, T. Zimmermann, A. Zeller, When do changes induce fixes?,

in: Proceedings of the 2nd International Workshop on Mining Software Repositories, Vol. 30 of MSR '05, ACM, 2005, pp. 1–5.

[45] C. Rosen, B. Grawi, E. Shihab, Commit guru: Analytics and risk prediction of software commits, in: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE '15, ACM, New York, NY, USA, 2015, pp. 966–969.

[46] E. Maldonado, R. Abdalkareem, E. Shihab, A. Serebrenik, Icsme study and survey data, http://das.encs.concordia.ca/uploads/2017/07/maldonado_icsme2017.zip, (Accessed on 01/30/2018) (2017).

[47] K. Dai, P. Kruchten, Detecting technical debt through issue trackers, in: 5th International Workshop on Quantitative Approaches to Software Quality (QuASoQ), IEEE, 2017, pp. 59–65.

[48] M. Ortu, G. Destefanis, B. Adams, A. Murgia, M. Marchesi, R. Tonelli, The jira repository dataset: Understanding social aspects of software development, in: Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE '15, ACM, New York, NY, USA, 2015, pp. 1:1–1:4.

[49] M. Stijlaart, V. Zaytsev, Towards a taxonomy of grammar smells, in: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE '17, ACM, 2017, pp. 43–54.

[50] G. Sridhara, Automatically detecting the up-to-date status of todo comments in java programs, in: Proceedings of the 9th India Software Engineering Conference, ISEC '16, ACM, 2016, pp. 16–25.

[51] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, H. C. Gall, How can i improve my app? classifying user reviews for software maintenance and evolution, in: Proceedings of the 31st International Conference on Software Maintenance and Evolution, ICSME '15, IEEE, 2015, pp. 281–290.

[52] E. Shihab, Y. Kamei, B. Adams, A. E. Hassan, Is lines of code a good measure of effort in effort-aware models?, Information and Software Technology 55 (11) (2013) 1981–1993.

[53] S. Akbarinasaji, A. Bener, Adjusting the balance sheet by appending technical debt, in: Proceedings of the 8th International Workshop on Managing Technical Debt, MTD '16, IEEE, 2016, pp. 36–39.

[54] S. Deterding, D. Dixon, R. Khaled, L. Nacke, From game design elements to gamefulness: defining gamification, in: Proceedings of the 15th international academic MindTrek conference: Envisioning future media environments, MindTrek '11, ACM, 2011, pp. 9–15.

[55] D. J. Dubois, G. Tamburrelli, Understanding gamification mechanisms for software development, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE '13, ACM, 2013, pp. 659–662.

[56] B. Biegel, F. Beck, B. Lesch, S. Diehl, Code tagging as a social game, in: Proceedings of the 30th International Conference on Software Maintenance and Evolution, ICSME '14, IEEE, 2014, pp. 411–415.

[57] J. Siegmund, Program comprehension: Past, present, and future, in: Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering, Vol. 5 of SANER '16, IEEE, 2016, pp. 13–20.