An Analysis of Techniques and Methods for Technical Debt Management: a Reflection from the Architecture Perspective

Carlos Fernández-Sánchez Juan Garbajosa Carlos Vidal Agustín Yagüe

Abstract—**Technical debt is a metaphor referring to the consequences of weak software development. Managing technical debt is necessary in order to keep it under control, and several techniques have been developed with the goal of accomplishing this. However, available techniques have grown disperse and managers lack guidance. This paper covers this gap by providing a systematic mapping of available techniques and methods for technical debt management, covering architectural debt, and identifying existing gaps that prevent to manage technical debt efficiently.**

I. INTRODUCTION

Technical debt is a metaphor referring to the consequences of weak software development. Cunningham [1] introduced the term technical debt. This metaphor has been used during the past several years as a means of making the intrinsic cost of internal quality weaknesses visible in software. Uncontrolled technical debt has a negative impact on software development [2], causing undesired effects in the quality of the product developed [3], in the developers' morale [4], and in the team's velocity. Managing technical debt is necessary in order to keep it under control. Technical debt management consists of identifying the source of extra cost when changing software and analyzing when it is profitable to invest effort into improving the software system. In practical terms, good management of technical debt means projects will have an amount of technical debt that will not prevent the organization from achieving its business objectives. Otherwise, software development efficiency and sustainability [5], and even the organization's strategic goals, may be seriously compromised. But the goal of technical debt management is not to strive for zero debt. In a world with finite resources some technical debt is inevitable [3]. From a business perspective, reducing technical debt is a good idea only if it leads to increased profitability [6]. However, too much technical debt might force the company to spend all of its efforts simply keeping the system operational instead of increasing value by adding new capabilities [3]. In a few words, if technical debt is not managed, a company could stop being profitable.

Available technical debt management techniques have grown disperse, and managers lack guidance when it comes to building a suite that supports the right set of techniques for managing technical debt, keeping in mind issues such as organizational goals, business strategies, time horizons, risk factors, financial constraints, and tax considerations. This paper covers this gap by providing a literature review, using the systematic mapping methodology (see Section II), of available techniques and methods for technical debt management, especially for architectural debt, and identifying existing gaps that prevent to manage technical debt efficiently. The results from the mapping have been analyzed from the perspective of software architecture, as architecture is considered nowadays central to software development.

The paper's structure is as follows. Section II describes the methodology used in this study. Section III describes the concept of technical debt and some related works. Section IV presents the results of the review performed. Section V discusses the results from the perspective of software architecture. Section VI discusses the validity of the study. Finally, conclusions are presented in Section VII.

II. METHODOLOGY

The research methodology selected to obtain the defined goals was the systematic mapping methodology. This study was performed following Petersen et al.'s guide [7]. Systematic mapping studies are designed to provide a broad overview of a research area, and consequently, suitable for the goal of this research. The goal of this systematic mapping is to respond to the following research question: **RQ1**–What techniques or approaches can be used to support technical debt management? The chain ***"technical debt"*** was used to search for studies about technical debt in the following digital repositories: IEEE Xplore, ACM, Scopus, ScienceDirect, Web of Science and SpringerLink. The last search was performed on December 2nd, 2014. The total number of articles obtained (without duplicates) was 615 and the number of selected studies was 25.

III. BACKGROUND AND RELATED WORK

Falessi et al. [8] identify requirements for tools necessary for managing technical debt. This work has been used as a starting point for our analysis. We have extended it by incorporating how other authors have considered these requirements and by identifying different techniques used to implement the requirements.

A. *Types of Technical Debt*

Several types of technical debt exist based on where they originate [3]. To classify the articles analyzed, this study uses the types defined by Tom et al. [3]. This is relevant because based on the type of technical debt, it will be possible to use different techniques to manage it.

According to the classification in [3], five different types of technical debt exist, considering its origin. **Code debt**: one way in which technical debt can manifest is in the form of poorly written code, including code duplication and complexity [3]. **Design and architectural debt**: architectural debt could be the result of sub-optimal upfront solutions, or solutions that become sub-optimal as technologies and patterns become superseded [3]. **Environment debt**: environment debt is the type of debt that manifests in the environment of an application, which includes development-related processes as well as hardware, infrastructure and supporting applications [3]. **Knowledge distribution and documentation debt**: this is the type of technical debt originating from a lack in knowledge distribution (e.g., absence of knowledge distribution between team members) [3]. **Testing debt**: testing debt is the type of technical debt that derives from, for example, a lack of test scripts, which leads to the need to manually retest the system before every release, or insufficient test coverage regardless of whether tests are automated or manually run [3].

B. *Elements of Technical Debt Management*

The elements of technical debt management represent concrete requirements to achieve in order to be able to manage technical debt. Three groups of elements were identified as a result of the classification scheme step of the systematic mapping methodology [7], and are displayed in Figure 1. These groups are: core elements, which are focused on estimating systems' technical debt; management elements, which identify what might be necessary to manage technical debt in real projects; and implementation elements, which indicate how these elements are implemented.

Below, the elements are described. **Identification of technical debt items**: the sources of technical debt have to be identified. **Principal estimation**: the principal of a technical debt item is the cost that would have to be paid off to eliminate the technical debt item. **Interest estimation**: the interest of a technical debt item is the extra cost that has to be paid over time if the technical debt item is not eliminated. **Interest uncertainty estimation**: the interest uncertainty represents the probability of paying interest. **Technical debt impact estimation**: technical debt impact has to be translated into economic consequences. This allows a company to perform cost-benefit analysis to identify the items with the most technical debt. **Automated estimates**: to manage technical debt it is desirable to have means to obtain value variables in an automatic way. **Expert opinion**: the expert opinion about a system will add information that it is not possible to obtain from available software information (code, documentation, etc). **Scenario analysis**: the output of the technical debt management has to be in the form of scenario analysis that clarifies the different possible decisions to be made. **Time-to-market**: when managing technical debt, the time to implements the decisions has to be taken into account. **When to implement decisions**: the implementation of a decision about technical debt can be made at different moments. These decisions include incurring technical debt or removing technical debt items. **Tracking technical debt over time**: it is necessary to track technical debt over time. **Visualizing technical debt**: it is necessary to have mechanisms to show how technical debt is impacting the system.

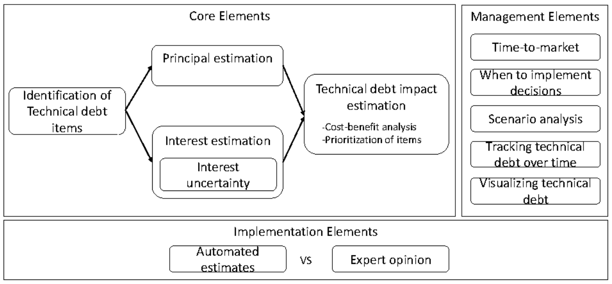

Fig. 1. Framework for the Elements for Technical Debt Management

IV. TECHNIQUES AND APPROACHES

One of the goals of the study (see Section II) was to know the extent to which currently available techniques or methods support technical debt management. The baseline to classify the studies in order to discover lacking areas in the technical debt management state-of-the-art were the elements described in Section III-B, and the types of technical debt described in Section III-A.

Figure 2 shows the number of studies dealing with the different types of technical debt, and the different elements necessary to manage technical debt.

In Table I, the types of technical debt described in Section III-A have been presented as columns, and the elements identified in Section III-B are presented as rows. The following subsection explains in detail how the current approaches either cover or fail to cover the elements. In Section V a discussion of the results is provided.

1) Identify Technical Debt Items: Approaches to identifying items of technical debt are mainly focused on code debt and architectural debt. Approximately 65% of all code anomalies were related to 78% of all architecture problems [9]; therefore it could be thought that these techniques overlap. However Zazworka et al. [10] show that different techniques (modularity violations [11] [12] [13], code smells [14] [15], grime [16],

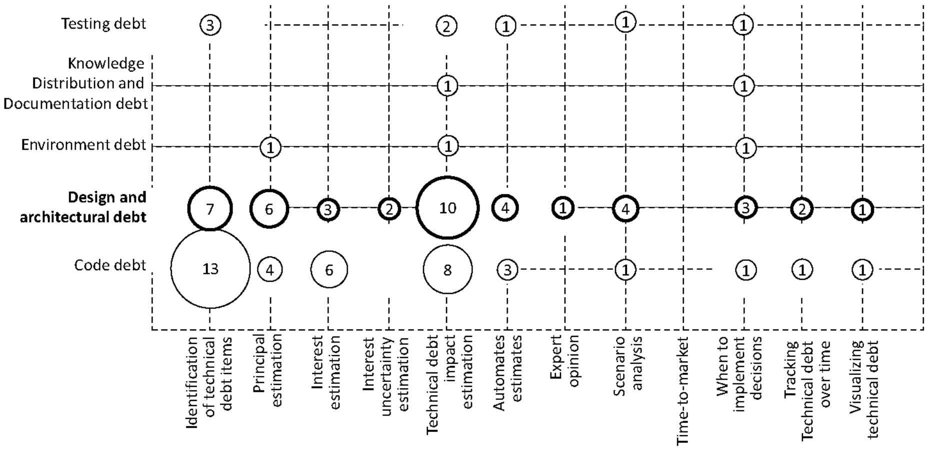

Fig. 2. Systematic Mapping - Types of Technical Debt and Elements for Technical Debt Management

and automatic static analysis [ASA] issues [17] [18]) do not overlap. That is, they point different technical debt items with regard to maintainability including code and architectural debt. Gat and Heintz [19] use Cutter's technical debt assessment, that is a combination of techniques: static and dynamic analysis of code deficits. It calculates the total technical debt in a system, and does not identify architectural debt items

Focusing on code technical debt, Nugroho et al. [20] propose a method based on lines of code, code duplication, McCabe's cyclomatic complexity, parameter counts, and dependency counts to score software on the basis of its maintainability. This approach calculates the total technical debt in a system but does not identify concrete technical debt items. Other methods, use the combination of code metrics and thresholds to define detection strategies [21], rules [22], or quality requirements [23]. Several other authors have used detection strategies [21] to identify code smells [15] [14] [24]. Also, different rules have been used to detect test smells [25]. In general, the studies that have focused on detecting code smells (e.g., God and brain code smells) show that a correlation exists between classes with code smells (or some of them) and change-prone, change size, change frequency, or error-prone classes [26] [27] [28] [15] [24] [14] [29]. A big issue is if the data have to be normalized with respect to the size in lines of code [15] [14]. Close to the code and test smell detection, Vetroét al. [18] [17] use ASA of source code to identify potential defects. But these studies do not indicate which code smells have to be analyzed to identify architectural technical debt.

To identify architectural technical debt, Cai et al. [12] use modularity violation detection (design rule violation) and rare class analysis to detect architectural debt items. For modularity violation detection, they use a tool called Clio [11] that allows one to detect files that change together when they are not supposed to be coupled. This tool uses a clustering technique to identify the system's modules, thus allowing one to analyze the modules' dependencies [30]. For rare class analysis, they use an algorithm to classify the files, considering their participation in patches created to fix detected bugs. A more recent tool, based on the same concepts of Clio, is Titan [31] which is based on the concept of design rule spaces [32]. Another kind of problem in modularity involves underutilized dependencies that slow down the build process, blow up the code size, and can indicate poor cohesion [33].

Other approaches that Izurieta and Bieman [34] [16] describe use design pattern grime as an indicator of maintainability decay. Design pattern grime is the buildup of unrelated artifacts in classes that play roles in a design pattern realization.

2) Principal Estimation: Studies reveal two main strategies to estimate the principal. The first one is based on having a repository of similar changes and projects. Based on this accumulated knowledge, it is assumed that a similar problem in a similar project will imply a same effort to solve the problem. Following this criteria, in [20] and [35] a function of the estimated percentage of lines of code to be changed and an estimation of effort per line of code are used. Both variables are estimated using statistical information collected from other projects using the same technology. Curtis et al. [22] detect code and architectural violations and use information obtained from several projects to estimate the effort needed to solve these kinds of violations, considering the programming language. The second strategy consists of detecting technical debt items and utilizing the typical effort estimation that the organization uses [36] [37] [12] [29].

3) Interest Estimation: For the interest estimation, some studies use an estimated maintenance effort based on information collected from other projects using the same technology [20] [35]. Others use defect likelihood and change likelihood to estimate the technical debt items' impact on system quality; in their case, they focus on classes with the God class code smell [14] [29]. They calculate the defect likelihood on the basis of the times a technical debt item is changed to solve defects. Similarly, they calculate change

TABLE I
IDENTIFIED TECHNIQUES AND APPROACHES FOR TECHNICAL DEBT MANAGEMENT

		Code debt	**Design and architectural debt**	Environment debt	Knowledge distribution and documentation debt	Testing debt
Core elements	Identification of technical debt items	[20] [19] [24] [28] [15] [34] [16] [29] [14] [22] [23] [21] [10]	**[22] [12] [21] [33] [23] [19] [13]**			[23] [19] [25]
	Principal estimation	[20] [35] [22] [29]	**[22] [20] [35] [36] [12] [37]**	[38]		
	Interest estimation	[20] [35] [10] [39] [29] [14]	**[20] [35] [12]**			
	Interest uncertainty estimation		**[12] [37]**			
	Technical debt impact estimation	[20] [35] [22] [40] [23] [29] [14]	**[12] [20] [35] [22] [23] [23] [36] [40] [41] [37]**	[40]	[40]	[23] [40]
Implementation elements	Automated estimates	[21] [22] [23]	**[22] [23] [12] [21]**			[23]
	Expert opinion		**[12]**			
Management elements	Time-to-market					
	When to implement decisions	[40]	**[12] [41] [40]**	[40]	[40]	[40]
	Scenario analysis	[23]	**[12] [23] [41] [37]**			[23]
	Tracking technical debt over time	[21]	**[36] [21]**			
	Visualizing technical debt	[23]	**[42]**			

likelihood on the basis of the number of changes performed in the technical debt item over time.

In [10], the analysis is extended to 30 indicators including modularity violations, several code smells and size to detect and analyze whose correlation with maintainability (defect likelihood and change likelihood).

Specifically focused on architectural debt, Cai et al. [12] use variations in the cost-per-change and cost-per-defect to estimate the interest. They propose three proxy measures of effort for estimating the cost: actions, number of commits/patches; churn, number of lines changed in a file; and discussions, number of textual comments about a file in the developers' discussions. They demonstrate that these three proxy measures of effort are valid, analyzing their correlation with other metrics that other authors have previously studied [43] [44]. However, they do not provide a concrete way of transforming the proxies' measure variations into maintenance effort variations.

Another proxy used to estimate interest is to monitor developers activity [39]. Collecting metrics about the activities with development environment that developers perform when they are working with classes shows the difference in maintenance efforts.

4) Interest Uncertainty Estimation: Several studies propose assigning a probability to the interest estimation. In this way, the interest can be estimated as expected interest. Cai et al. [12] use triangular distribution for the interest estimation that is, a pessimistic value, an optimistic value, and a most-likely value for the interest estimation. However, they do not provide concrete methods to estimate such values. Fernandez-Sanchez et al. [37] use decision trees to model the possible evolutions of interest over time. In this method, this estimation is delegated to project managers or architects. Therefore, it is a challenge how to estimate probability distribution of interest to manage architectural technical debt.

5) Technical Debt Impact Estimation: This section incorporates techniques that focus on the economic consequences of technical debt, perform some cost-benefit analysis on the basis of principal and interest, and/or provide ways for ranking technical debt items considering their impact on the system.

One strategy used to estimate technical debt's economic consequences is oriented to provide a big picture of the whole system without providing low-level detail of how technical debt is distributed in the system. Nugroho et al. [20] propose a method based on code metrics (lines of code, code duplication, McCabe's cyclomatic complexity, parameter counts, and dependency counts) to score software on the basis of its maintainability. This same approach can be seen in [35]. Based on the accumulated data of more than 170 systems, they provide estimations of the cost of change in order to increase the software's score in the ranking. Nevertheless, this method does not provide guidance on how to improve the architecture. It only estimates the economic consequences at whole-system level, while it does not identify the modules or components in which technical debt has accumulated and consequently, the technical debt distribution over the system.

Curtis et al. [22] use average effort—considering the programming language—per type of code or architectural violation detected and the cost per hour to estimate the principal cost. This provides a general view of the system's technical debt, but they do not consider the interest or other aspects in their estimations. Similar to this solution, Letouzey and

Ilkiewicz [23] assign a remediation cost and a non remediation index per each type of quality requirement (i.e., type of technical debt item) defined in their technical debt management model.

For architectural debt, Cai et al. [12] use three file metrics to measure maintenance efforts. They use churn (lines of code changed in each change), actions (commits in which the file has been involved), and discussions (a metric based on the text mining of different sources, such as discussion forums and commit descriptions). They demonstrate that these three metrics are correlated with maintenance effort and proposed them to estimate the benefit of refactoring. However a clear way of transforming the three metrics' values into a refactoring benefit is not shown. They also use the cost to solve defects and the defect rate as variables. Finally, they use real options to combine the different variables, performing an economic analysis of the future costs and benefits of the system. Similarly, Alzaghoul and Bahsoon [38] analyze Web service selection using real options. They consider the technical debt generated when the selected web services do not have enough scalability to support the system's future growth, but they also consider the importance of not wasting resources due to excess of scalability capabilities.

Several authors use cost-benefit analysis to estimate technical debt's impact. The basic way to perform a cost-benefit analysis is to compare the cost of removing technical debt (principal) with the benefit obtained (normally, the avoided cost due to not having interest) [40] [23].

Some of these methods are oriented to release planning [41]. Their goal is to identify when it is more profitable to initially invest in architecture or when it is better to release functional features as soon as possible. That is, this method is oriented to decide when it is better to incur technical debt by adding features as soon as possible or when it is better to add features later.

Some authors add time as a variable to be considered in the analysis. Therefore, they take into account technical debt's possible evolutions over time [36] [20] [35]. Other authors use real options to perform the analysis [12], in this case using a valuation techniques based on Monte Carlo simulations. Also, considering the time frame, other authors use decision trees to perform the cost-benefit analysis [37].

Authors use the estimated cost-benefit ratio to perform a ranking [14] [29] [23]. This ranking is useful when it is necessary to solve the technical debt items with the highest priority. The ranking is dependent on the cost-benefit analysis method. Therefore, only the variables considered in such a method are used to perform the ranking. Another method that Guo and Seaman use [40] consists of applying a model based on the portfolio approach. Portfolio management comes from the finance domain and focuses on selecting the assets that maximize return on investment or minimize investment risk [40].

Several challenges persist. In order to have a realistic cost-benefit analysis, it is necessary to evaluate more than just principal and interest. Many times, authors point out that technical debt is originated when time-to-market restrictions are present. Therefore, the costs of delaying a functionality or release in order to remove technical debt has to be considered. Additionally, the team's capacity to perform tasks should be considered because it is limited.

6) Automated Estimates: Many authors use tools for automatically detecting sources of technical debt (see Section IV-1). Also, historical data can be used to estimate the interest for a specific organization or project [8]. Some authors define code and file metrics used to estimate interest that can be extracted automatically from source code (see Section IV-3). Usually, automated estimates include code/file metrics, effort measures, and the files' evolutionary history over the system's different versions [12].

Two approaches can be identified: one based on having a historical repository of projects with similar characteristics and a second based on mining a project's available resources (source code, control version systems, etc).

7) Expert Opinion: Almost all of the authors suggest the need to use expert knowledge to add information that cannot be estimated in another way. Also, Cai et al. [12] propose a decision-support system for architectural refactoring decisions. Their goal is to give refactoring recommendations to experts (the project manager or the architect). Then, the expert could analyze different scenarios on the basis of the estimations to decide which recommendation to follow.

8) Scenario Analysis: The authors use different kinds of scenarios: (1) scenarios to analyze technical debt goals and estimate the effort required to achieve them [23]; (2) release scenarios to analyze the most profitable release path based on the architectural technical debt incurred [41] [45]; (3) what-if scenarios used to provide managers or architects with the possibility of seeing the estimated impact of different decisions regarding refactoring the architecture in order to improve the software's modularity [12]; (4) and change scenarios to analyze the possible evolution of the technical debt in the system [37].

None of the proposed method analyze several types of scenarios. Furthermore, the methods that use scenarios as part of their analyses focus on concrete kinds of technical debt or use different detection strategies that make them difficult to integrate. Further effort is needed to define methods for estimating architectural debt that allow one to combine all of the types of scenarios.

9) Time-to-Market: None of the authors propose explicit methods for considering time-to-market in order to manage technical debt. Time-to-market is essential to the success of many projects and products. Therefore, it should be considered explicitly in making a decision about when to delay groups of features or when to remove technical debt.

10) When to Implement Decisions: Several authors identify release planning as the step during which decisions about technical debt management can be executed [40] [41] [45]. Two different decisions have been identified. The first consists of determining when it is necessary to reduce technical debt [40]. The second is oriented to decide whether it is better to implement features as soon as possible (this implies future reworks due to adaptations) or expend some time in architectural tasks and then implement the features. [41].

Real options can help with deciding when refactoring is profitable [12]. If the case is an option-based approach some important estimations are usually required: data inferred from evolution history and the prediction of future changes and cost estimations [12]. Other authors have suggest using portfolio theory with the same objective [40].

11) Tracking Technical Debt over Time: Many articles use project's historical data to estimate the interest (see Section IV-3) based on the evolution of some code or file metrics. However, only a few consider technical debt's evolution over time. Some authors analyze technical debt' evolution over the project releases [36]. Also, Marinescu [21] defines an indicator for tracking technical debt's evolution over the project releases. These indicators provide information about how technical debt is rising or not rising in the system, but they do not provide information about whether or not accumulating technical debt has additional consequences.

12) Visualizing Technical Debt: In general, few methods exist for visualizing technical debt. Most studies show charts featuring the relationship among principal, interest, and/or time. Other studies use design structure matrix (DSM) to show different kinds of relationships between the software modules [42], or dashboards in order to make visible the proportion of lines of code that exceed the quality requirement established [23]. Still, few provide additional tools or techniques for visualizing how the technical debt is impacting the system or which components in the architecture have high architectural debt.

V. DISCUSSION FROM THE ARCHITECTURE PERSPECTIVE

Table I and Figure 2 show how most of the techniques and methods are oriented to satisfy code debt, design, and architectural debt. Section IV-1 described that around 65% of all code anomalies were related to 78% of all architecture problems [9], and Zazworka et al. [10] stated that different techniques such as modularity violations, code smells, grime, and automatic static analysis [ASA] issues do not overlap. However this, design technical debt is somehow in the middle between code and architecture (as the God class smell is), and management techniques for the architectural debt make extensive use of code analysis. Therefore, further studies that address architectural debt introducing a holistic approach are needed.

A second issue is that approaches specific for software architecture are directly addressed in Sections IV-1, IV-5, IV-8: Identify technical debt items, Technical debt impact estimation, and Scenario analysis; and indirectly in Sections IV-2, and IV-3: Principal estimation, and Interest estimation. This shows that additional efforts are required to increase the number of techniques for architectural debt management. This is very clear, for instance, in the case of documentation: how documentation debt caused by lack of design decisions documentation affects the architectural debt has not been addressed at all. This is also the case for environment, and testing debt, though these two would affect architecture, in principle, less.

In addition, the analysis of the different elements described in Section IV shows that a number of concepts are common to several of the elements such as time to market, release management or uncertainty. The uncertainty concept, appears explicitly in the Section IV-4 Interest uncertainty estimation, and implicitly in Sections IV-5, IV-9, IV-10, and IV-11: Technical debt impact estimation, Scenario analysis, Time-to-market, When to implement decisions, and Tracking technical debt over time. The relevance of uncertainty in software architecture has been recently highlighted in [46].

Also it is necessary to perform studies to understand how different techniques can work together (e.g., avoiding overlapping between code, design and architectural debt). Only one study has been found to analyze how different techniques apply to different technical debt items [10].

Finally, Kruchten et al. [47] indicate two main software areas for technical debt: maintenance (errors) and evolution (new features). Studies are more often focused on errors than on addressing changes for adding new features. Studies do not generally analyze if differences exist between the causes of error-prone software or hard-to-change software for adding new features. Currently, studies on software architecture flexibility from a technical debt perspective are unusual; hence, more studies are definitely needed.

VI. THREADS TO VALIDITY

Completeness. We have used the chain "technical debt" to search for studies about technical debt. This might have excluded available techniques that could be used for technical debt management but that have not already been published in the context of technical debt. However opening the search to other areas such as maintainability or evolvability would make the study unapproachable. Other studies that demonstrate their validity in this field must address the use of other techniques related to maintenance or evolvability for technical debt management. **Study selection**. The process of article selection was completed following the steps described by Petersen et al. [7]. In order to reduce possible bias two of this article's authors completed the revision, and discrepancies were solved via agreement among the four authors. **Quality of the analyzed articles**. In a literature review, poor quality sources could lead to inaccurate conclusions. In this study, most of the articles analyzed passed a peer-review process, and only little number of relevant technical reports and books that other technical debt studies have referenced have been included. **Personal bias**. The analysis of the articles, aimed at identifying the elements needed for managing technical debt and classifying the techniques, can be influenced by the personal bias of the person analyzing the article. In order to mitigate this risk, the same technique used in the selection was used in this analysis: two authors analyzed the articles independently, and discrepancies were solved via agreement among the four authors.

VII. CONCLUSION

In this paper, the available techniques for technical debt management identified in the current literature have been analyzed from the software architecture perspective. Many of the techniques are oriented to fulfill design and architectural debt, as well as code debt. This analysis shows that further studies are necessary to fully support architectural debt and also essential elements not currently covered, such as time-to-market. In the case of architectural debt, it is necessary to perform more studies to achieve that different metrics, estimates, or techniques can work together.

ACKNOWLEDGMENT

The work reported here has been partially sponsored by the Spanish fund: MESC DPI2013-47450-C2-2-R.

REFERENCES

[1] W. Cunningham, "The wycash portfolio management system," *SIGPLAN OOPS Mess.*, vol. 4, no. 2, pp. 29–30, Dec. 1992.

[2] K. Power, "Understanding the impact of technical debt on the capacity and velocity of teams and organizations: Viewing team and organization capacity as a portfolio of real options," in *MTD Workshop*, 2013.

[3] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *Journal of Systems and Software*, vol. 86, no. 6, pp. 1498–1516, 2013.

[4] L. Peters, "Technical debt: The ultimate antipattern," in *MTD Workshop*, 2014.

[5] J. Holvitie, "Software implementation knowledge management with technical debt and network analysis," in *RCIS*, May 2014, pp. 1–6.

[6] S. Tockey, "Chapter 3 - aspects of software valuation," in *Economics-Driven Software Architecture*, I. Mistrik, R. Bahsoon, R. Kazman, and Y. Zhang, Eds. Boston: Morgan Kaufmann, 2014, pp. 37 – 58.

[7] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic mapping studies in software engineering," in *EASE*, 2008.

[8] D. Falessi, M. Shaw, F. Shull, K. Mullen, and M. Keymind, "Practical considerations, challenges, and requirements of tool-support for managing technical debt," in *MTD Workshop*, 2013.

[9] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa, "On the relevance of code anomalies for identifying architecture degradation symptoms," in *CSMR*, March 2012, pp. 277–286.

[10] N. Zazworka, A. Vetro, C. Izurieta, S. Wong, Y. Cai, C. Seaman, and F. Shull, "Comparing four approaches for technical debt identification," *Software Quality Journal*, vol. 22, no. 3, pp. 403–426, 2014.

[11] S. Wong, Y. Cai, M. Kim, and M. Dalton, "Detecting software modularity violations," in *ICSE*, May 2011, pp. 411–420.

[12] Y. Cai, R. Kazman, C. V. Silva, L. Xiao, and H.-M. Chen, "Chapter 6 - a decision-support system approach to economics-driven modularity evaluation," in *Economics-Driven Software Architecture*, I. Mistrik, R. Bahsoon, R. Kazman, and Y. Zhang, Eds. Boston: Morgan Kaufmann, 2014, pp. 105 – 128.

[13] R. Schwanke, L. Xiao, and Y. Cai, "Measuring architecture quality by structure plus history analysis," in *Proceedings - International Conference on Software Engineering*, 2013, pp. 891–900.

[14] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *MTD Workshop*, 2011.

[15] S. Olbrich, D. Cruzes, and D. I. K. Sjoberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *ICSM*, Sept 2010, pp. 1–10.

[16] C. Izurieta and J. Bieman, "Testing consequences of grime buildup in object oriented design patterns," in *ICST*, April 2008, pp. 171–179.

[17] A. Vetro', M. Torchiano, and M. Morisio, "Assessing the precision of findbugs by mining java projects developed at a university," in *MSR*, May 2010, pp. 110–113.

[18] A. Vetro, M. Morisio, and M. Torchiano, "An empirical validation of findbugs issues related to defects," in *EASE*, April 2011, pp. 144–153.

[19] I. Gat and J. D. Heintz, "From assessment to reduction: How cutter consortium helps rein in millions of dollars in technical debt," in *MTD Workshop*, 2011.

[20] A. Nugroho, J. Visser, and T. Kuipers, "An empirical model of technical debt and interest," in *MTD Workshop*, 2011.

[21] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM Journal of Research and Development*, vol. 56, no. 5, pp. 9:1–9:13, Sept 2012.

[22] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the principal of an application's technical debt," *Software, IEEE*, vol. 29, no. 6, pp. 34–42, Nov 2012.

[23] J. Letouzey and M. Ilkiewicz, "Managing technical debt with the sqale method," *Software, IEEE*, vol. 29, no. 6, pp. 44–51, Nov 2012.

[24] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, "Building empirical support for automated code smell detection," in *ESEM*, 2010.

[25] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *ICSM*, Sept 2012, pp. 56–65.

[26] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *J. Syst. Softw.*, vol. 80, no. 7, pp. 1120–1128, Jul. 2007.

[27] F. Khomh, M. Di Penta, and Y. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *WCRE*, Oct 2009, pp. 75–84.

[28] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *ESEM*, 2009.

[29] N. Zazworka, C. Seaman, and F. Shull, "Prioritizing design debt investment opportunities," in *MTD Workshop*, 2011.

[30] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi, "Design rule hierarchies and parallelism in software development tasks," in *ASE*, Nov 2009, pp. 197–208.

[31] L. Xiao, Y. Cai, and R. Kazman, "Titan: A toolset that connects software architecture with quality analysis," in *FSE*, 2014.

[32] ——, "Design rule spaces: A new form of architecture insight," in *ICSE*, 2014.

[33] P. Wang, J. Yang, L. Tan, R. Kroeger, and J. David Morgenthaler, "Generating precise dependencies for large software," in *MTD Workshop*, 2013.

[34] C. Izurieta and J. Bieman, "How software designs decay: A pilot study of pattern evolution," in *ESEM*, Sept 2007, pp. 449–451.

[35] J. de Groot, A. Nugroho, T. Back, and J. Visser, "What is the value of your software?" in *MTD Workshop*, 2012.

[36] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. da Silva, A. Santos, and C. Siebra, "Tracking technical debt: An exploratory case study," in *ICSM*, 2011.

[37] C. Fernandez-Sanchez, J. Diaz, J. Perez, and J. Garbajosa, "Guiding flexibility investment in agile architecting," in *HICSS*, 2014.

[38] E. Alzaghoul and R. Bahsoon, "Cloudmtd: Using real options to manage technical debt in cloud-based service selection," in *MTD Workshop*, May 2013.

[39] V. Singh, W. Snipes, and N. A. Kraft, "A framework for estimating interest on technical debt by monitoring developer activity related to code comprehension," in *MTD Workshop*, 2014.

[40] Y. Guo and C. Seaman, "A portfolio approach to technical debt management," in *MTD Workshop*, 2011.

[41] R. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In search of a metric for managing architectural technical debt," in *WICSA/ECSA*, Aug 2012, pp. 91–100.

[42] J. Brondum and L. Zhu, "Visualising architectural dependencies," in *MTD Workshop*, 2012.

[43] A. MacCormack, J. Rusnak, and C. Y. Baldwin, "Exploring the structure of complex software designs: An empirical study of open source and proprietary code," *Management Science*, vol. 52, no. 7, pp. 1015–1030, 2006.

[44] J. Carriere, R. Kazman, and I. Ozkaya, "A cost-benefit framework for making architectural decisions in a business context," in *ICSE*, vol. 2. New York, New York, USA: ACM Press, May 2010, p. 149.

[45] J. Ho and G. Ruhe, "When-to-release decisions in consideration of technical debt," in *MTD Workshop*, 2014.

[46] E. Letier, D. Stefan, and E. T. Barr, "Uncertainty, risk, and information value in software requirements and architecture," in *ICSE*, 2014.

[47] P. Kruchten, R. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *Software, IEEE*, vol. 29, no. 6, pp. 18–21, Nov 2012.