# Methods and Tools for TD Estimation and Forecasting: A State-of-the-art Survey

Dimitrios Tsoukalas[*][†], Miltiadis Siavvas[*][‡], Marija Jankovic[*], Dionysios Kehagias[*],
Alexander Chatzigeorgiou[†], Dimitrios Tzovaras[*]
[*] Centre for Research and Technology Hellas, Thessaloniki, Greece
[†] Department of Applied Informatics, University of Macedonia, Thessaloniki 54643, Greece
[‡] Imperial College London, SW7 2AZ, London, United Kingdom
tsoukj@iti.gr, siavvasm@iti.gr, jankovicm@iti.gr, diok@iti.gr, achat@uom.gr, dimitrios.tzovaras@iti.gr

*Abstract*—**Technical debt (TD), a metaphor inspired by the financial debt of economic theory, indicates quality compromises that can yield short-term benefits in the software development process, but may negatively affect the long-term quality of software products. Numerous techniques, methods, and tools have been proposed over the years for estimating and managing TD, providing a variety of options to the developers and project managers of software applications. However, apart from managing TD, predicting its future value is equally important since this knowledge is expected to facilitate decision-making tasks regarding software implementation and maintenance, such as incurring or paying off TD instances. To this end, the purpose of the present study is to (i) summarize the work that has been conducted until today in the field of TD estimation and forecasting, and (ii) to identify existing open issues that have not been adequately addressed yet and require further research. The present survey led to two interesting observations. Firstly, none of the existing TD estimation methods and tools has reached a desired level of maturity, while a large volume of previously uninvestigated metrics and techniques exist that could potentially increase the completeness of TD estimation. Secondly, no notable contributions exist in the field of TD forecasting, indicating that it is a scarcely investigated field. The latter constitutes the main finding of the present literature review, since TD forecasting could lead to the development of practical decision-making mechanisms, which could assist developers and project managers in taking proactive actions regarding TD repayment.**

***Keywords—technical debt; technical debt estimation; technical debt forecasting***

## I. Introduction

The term Technical Debt was first introduced in 1992 by Ward Cunningham [1] to describe the problem of introducing long-term problems to software products, by not resolving existing quality issues early enough in the overall software development lifecycle (SDLC). The TD metaphor was initially related to software implementation (i.e. at the code level), but was gradually extended to all phases of the SDLC, i.e. software architecture, design, documentation, requirements, and testing [2]. The TD notion was inspired by the concept of the financial debt of economic theory, leading to the adoption of a multitude of financial theories for its identification, repayment, quantification etc. As in financial debt, TD incurs interest payments in the form of increased future costs owing to the earlier quick and dirty design and implementation choices. However, managing TD is more complicated than managing financial debt because of the uncertainty involved [3].

The efficient management of TD requires a clear understanding of the state of the art of Technical Debt Management (TDM). One of the most dominant characteristics of TD is its interdisciplinary nature since it combines elements from both software engineering and financial theory [4]. As a result, the methods proposed in the literature for managing TD follow two different paths and thus, can be classified into two broad schools of thought. The first one is the financial aspect of TD, which includes approaches such as Portfolio management, Real options and software economics [4]. The second one is the software engineering aspect of TD, which includes estimation methods such as calculation models, code metrics, operational metrics, etc. [5], with the SQALE method being the most widely used among them [6].

Although the number of various techniques, methods and tools for managing TD continues to proliferate, they have not yet reached the desired level of maturity [5]. Besides, since no commonly accepted standard for estimating and managing TD [5] exists, it is not clear how these tools map to TDM activities like identification, measurement, or repayment. As a result, researchers, developers and managers perceive the concept of TD in different ways and are unable to distinguish between the software quality compromises that can be attributed as TD and those that cannot.

Nevertheless, the evolution of a software system usually implies an analogous evolution of its TD as well. A method or tool that would assist software project managers in decision-making in uncertainty by predicting future TD of a software system is of paramount importance. However, while various researchers have addressed the topic of forecasting the evolution of various aspects directly or indirectly related to the TD of a software project, such as code smells [7], fault-proneness [8] and evolution trends [9], no concrete approaches have been proposed so far regarding the forecasting of TD itself.

To this end, in this study two important TD-related aspects, namely TD estimation and TD forecasting, are theoretically examined. In particular, the purpose of this paper is to review the most significant attempts in the broader field of TD estimation and forecasting, identify existing open issues of high interest, and potentially propose directions for future research. Hence, this paper can act as a reference for researchers that wish to contribute in the field of TD, to gain a solid understanding of existing solutions and identify open issues that require further research. All these are presented in detail in the rest of the paper.

The rest of the paper is structured as follows: Section 2 describes the related work on TD estimation methods and tools, as well as forecasting methods and techniques under the scope of Software Evolution analysis. Section 3 describes the open issues that were identified through the present survey in the field of TD estimation and forecasting and proposes possible contributions. Finally, Section 4 concludes the paper and presents potential directions for future work.

## II. Related Work

### A. The Technical Debt Metaphor

Ward Cunningham introduced the metaphor of Technical Debt [1] in 1992 as follows:

*"Shipping first-time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise"*.

There are several causes for creating TD. Fowler [10] [11] states that software development debt is usually a consequence of time pressure. Software engineers and developers often make non-optimal design decisions to quickly address fast-changing requirements, which is leading to a poorly designed system and increased TD over time. Kruchten et al. [12] assign TD to YAGNI decisions (You Ain't Gonna Need It) that often result in unjustified and unnecessary investments in new features, architecture, over engineering, etc. Martin Fowler [11] proposes TD quadrant, a 22 matrix (Intentionality x Wisdom), to visualize four different pathways that lead to TD. According to his study, it is not enough to discuss if something is a TD or not, but it is crucial to analyze intention (deliberate or inadvertent) and awareness (reckless or prudent). McConnell [13] suggests a similar categorization, arguing that TD may be unintentional and intentional. Unintentional debt is often a consequence of poor coding practices, while intentional debt is a result of non-optimal decisions that are committed on purpose. He proposes a further classification into short-term and long-term debt. Short-term debt is taken tactically to cover smaller gaps with the goal to speed up software release, and it is expected to be paid off quickly. On the other hand, long-term debt is taken strategically having in mind significant software improvements and can be carried out for years. Suryanarayana et al. [14] point out that extreme situation when accumulated TD is enormous and cannot be paid off could lead to technical bankruptcy.

Moreover, several recent studies have highlighted the need to analyze TD from SDLC point of view. Li et al. [5] classify different TD types into ten levels based on the occurrence during the main phases of a software development process (i.e., requirements, design and architecture, implementation, testing, building, documentation, infrastructure, versioning, and defects). According to the authors, requirements TD refers to compromises made between the optimal requirements specification and the actual system implementation, while architectural TD is occurred by not-optimal architecture decisions that affect some horizontal quality aspects, such as maintainability. Design TD refers to code smells such as intensive coupling, God classes, high-complexity, etc. [1]. Code TD is the poorly written code that violates best coding practices or coding rules, such as duplicated code. Test TD refers to incomplete testing coverage while build TD refers to flaws or complexity in the build process of a software system. Documentation TD refers to insufficient or incomplete documentation. Finally, versioning TD refers to the problems in source code versioning, while defect TD refers to defects, bugs, or failures found in software systems. However, several researchers and practitioners are sharing the opinion that visible symptoms of low software quality, such as defects or bugs, should not be considered as TD liabilities [15]. Instead, they suggest that TD should be limited to internal system qualities, primarily maintainability and evolvability. Similarly, Kruchten et al. [12] outline the TD that occurs in various phases of the development process (i.e., code, tests, documentation). They attribute the TD of architecture to bad structural or architectural choices or technological gaps. Finally, Sterling [16] observes that the SDLC process may affect the size of the TD. For example, an agile software development process would create less TD than a waterfall model due to a more flexible response to change.

Like the financial debt, TD also entails paying interest in the form of additional effort that is needed to be spent on maintaining the software due to its declining design-time quality. Several studies have dealt with the notion of interest in TDM. According to Ampatzoglou et al. [4] and Li et al. [5], interest is the most frequently used financial term in TDM research field and is defined through various approaches like references to economic theory [17], or software engineering concepts [18] [19]. Under this perspective, Chatzigeorgiou et al. [20] propose an approach for estimating the breaking point under which the accumulated interest becomes larger than the principal, i.e. the timestamp in which TD of a software product is no longer sustainable. Trying to expand this work, Ampatzoglou et al. [21] instantiate and validate FITTED, a framework that assesses the breaking point of source code modules to support decision making with respect to investments on improving quality of a software.

### B. Technical Debt Estimation Methods and Tools

Technical Debt Management (TDM) is one of the fastest-growing research areas of software technology (90% of the research was published after 2010 [22]) and even has a dedicated conference, namely International Conference on Technical Debt (TechDebt). TDM includes several different activities that assist managers and developers in making TD visible and controllable [5], such as TD identification, estimation, prioritization, prevention and repayment. There are many methods and tools proposed in the literature for supporting TDM activities [23] and, as in the case of financial debt, the management of TD must be programmed founded on the amount of interest and the possibility of repayment over time. Concerning specific approaches to TDM, Brown et al. [2] stress the need to develop new models and techniques for assessing, managing, identifying causes, and repaying TD based on its economic impact. They argue that compensation must be made in such a way that there is a balance between short-term deadlines and long-term viability. Additionally, Seaman et al. [24] identify four approaches to TDM, including Cost-Benefit Analysis, Analytic Hierarchical Process (AHP), Portfolio Management Model and Real Options.

In this study, we will be focusing on TD estimation, a specific activity of TDM that quantifies the benefit and cost of known TD in a software system through estimation techniques. As it is the case for all TDM activities, TD estimation methods and tools can be also separated into two broad categories, the financial approaches and the software engineering approaches.

**Financial Approaches**

In recent years, various approaches based on economic theory have been applied to quantify TD principal and interest. One of the first TD modeling and visualization efforts include the Highsmith curve [25], which quantifies TD as the difference between actual and optimal Cost of Change (COC) over time.

Following the financial aspects of TD estimation, Guo and Seaman [19] leverage the Portfolio Management theory in the finance domain to determine the optimal collection of TD items that should be incurred or held. Through their approach, the researchers try to quantify TD principal as the effort required to resolve TD items and TD interest as the probability of interest to occur. Based on the Portfolio Management approach, Holvitie and Leppanen introduce DebtFlag [26], a tool designed to support TDM by capturing, tracking and resolving TD of software projects at the implementation level. This tool provides developers with lightweight documentation functionalities to capture TD and link them to corresponding parts in the implementation phase.

In another approach towards TD estimation, Alzaghoul and Bahsoon [27] state that the web service selection decision might incur a TD that is essential to be quantified and managed. Towards this aim, they exploit the Real Options theory by introducing a new method that aims to quantify TD in service level for cloud-based system architectures. In their approach, they construct a two-step binomial tree to quantify and predict the period during which TD is reduced to zero, taking into consideration several dimensions including Service Level Agreement (SLA), none-compliance, quick selection decisions and underutilization of the web service capacity.

Finally, in their study, Curtis et al. [28] are based on Software Economics theories and quantify TD as the cost of violating architectural rules, code rules, and best practices, giving three levels of severity to violations: high, medium and low. In order to achieve that, they introduce a function that quantifies principal and interest taking as input software artefacts, metrics, historical effort, or personnel activity. To further support their findings, they integrate their formula into Application Intelligence Platform (CAST), a tool that quantifies TD by identifying violations in source code and categorizing them by quality attributes. This tool is designed to manage TD by analyzing multi-tiered, multi-technology applications for technical vulnerabilities and adherence to architectural and coding standards.

**Software Engineering Approaches**

The software engineering aspect of TD estimation lays its foundations on the notion that software quality metrics and the time and effort required for a software change can be used to quantify its impact. For instance, if a software is vulnerable or does not satisfy all system requirements, vulnerabilities must be fixed and the requirements met. Therefore, the number of vulnerabilities or unsatisfied requirements is an indicator of TD. In addition, if a software has been produced with excessively complex code, then its future changes are more expensive. In this case, metrics like coupling, cohesion, complexity, etc. can also be applied to assess TD [29].

Over the last years, several software engineering methods have been proposed to quantify a software systems level of TD. In a related study, Gaudin [30] introduces a new TD estimation formula that takes as input custom source code metrics to calculate a global indicator of TD. This indicator reflects how much effort is required to get a flawless score on the Seven Axes of Quality analysis, namely the bad distribution of the complexity, duplications, lack of comments, coding rules violations, potential bugs, no unit tests and bad design. One of the most representative tools for assessing the TD of a software product using the Seven Axes of Quality formula is the Technical Debt Evaluation plugin for SonarQube, an open source platform for continuous inspection of code quality. SonarQube provides a dashboard for visualizing quality attributes of code, tests, design, and architecture. Under the hood, it performs static analysis of the source code to detect bugs, code smells and security vulnerabilities and provides the capability to analyze, assess, visualize and prioritize TD based on the quality axes as mentioned above.

In another study, Bohnet and Dllner [31] calculate TD by using Software Maps to monitor code quality and development activity. In their approach, they argue that software maps enable managers to express and combine information about software development, software quality, and system dynamics. They also claim that software maps can support decision-making processes by investing the scarce developers time to improve code quality and facilitate the future maintenance of the system.

Similarly, Nugroho et al. [32] propose an approach for quantifying TD principal and interest based on an empirical assessment method of software quality developed by the Software Improvement Group (SIG). Their method comprises of two parts, the estimation of repair effort and the estimation of maintenance effort. Following a different approach, the work of De Groot et al. [33] introduces three models to determine software value based on the notions of TD by using the Rebuild Value, i.e. the cost to rebuild a system from scratch using similar technology. In one of their models, they calculate the TD as the amount of work (in person-months) that is required to improve the level of software quality. However, to the best of knowledge, no tools exist to implement the methods mentioned above.

In addition, Ernst [17] proposes an approach for TD estimation on the requirements level by introducing Solution comparison, a method that calculates the distance between the optimal specification and the actual implementation of the system. To validate his method, in the same study he also introduces RE-KOMBINE, a requirements modeling tool that enables useful measures and models the TD present in requirements tradeoffs. Another tool based on the same notion is proposed by Strasser et al. [34]. The Automated Software Tool for Validating Design Patterns based on the Role Based Metamodeling Language (RBML) is a compliance checker that quantifies TD on the design level by calculating the distance between a realization of a design pattern and the intended design. For that purpose, the tool compares UML class diagrams of instances of design patterns with their RBML representations and reports back if the given UML diagram is compliant or not.

Moreover, Curtis et al. [35] present a formula for TD calculation with adjustable parameters for estimating the principal of TD from structural quality data. On the other hand, Letouzey [36] presents the widely used SQALE method for monitoring and assessing the quality and TD of the source code. One of the most representative tools for assessing the TD of a software product using the SQUALE method is SQUORE, a commercial quality management tool that uses four indicators namely: efficiency, portability, maintainability, and reliability to calculate the TD. For each of these indicators, a set of quality rules is assigned. One of the advantages of this tool is that it takes into account source code, unit tests, documentation quality, available functional requirements, etc. resulting in a more accurate and complete calculation of TD. Also, SonarQube used the SQUALE method to assess the TD of a software product in previous years, but it has switched to another method.

In the same way, Nord et al. [37] follow an architecture-focused and measurement-based approach to introduce a metric for the rapid management of TD associated with architecture level in order to optimize development costs. To support their work, they argue that making the architectural debt visible provides all necessary information for making informed decisions for managing the potential impact of rework over time. Similarly, Marinescu [38] introduces a novel framework for assessing TD using a technique for detecting design flaws and violations of well-known rules and design principles. To make the framework inclusive, the author integrates a set of metric-based detection rules for design flaws that cover the majority of the aspects of design such as complexity, coupling and encapsulation.

Finally, in a recent study, Sanchez et al. [39] introduce TEDMA, an open tool that quantifies TD by computing TD metrics and integrating techniques implemented by third party tools. The novelty of this tool is that it supports analysis of the evolution of the metrics over the software evolution of the project. This kind of approach has not been introduced in any of the previous methods and tools presented in the study.

## C. *Software Evolution and Technical Debt Forecasting*

Software evolution is a term used in software engineering to refer to the process that starts with the development and then provides incremental updates of the software. According to Lehmans laws of software evolution, software systems must evolve over time or they will become irrelevant. With the evolution of the software systems, accumulated TD is evolving as well. Under those circumstances, being able to forecast not only the evolution of software quality but also the evolution of TD principal and interest of a software system in the future is of great significance and value. Such a work would enable project managers and developers to support decision-making in uncertainty and plan precise payback strategies, in order to manage TD promptly and avoid unforeseen situations long-term.

Gaining a higher level of information about the evolution of large software systems is a key challenge in dealing with increasing complexity and decreasing software quality [40]. For this reason, the attempts to analyze, understand and predict the evolution of a software system have increased considerably in the last years and nowadays, the terms software evolution and software maintenance are often used as synonyms [41]. In a relevant study, Lehman [42] highlights the importance of studying the evolutionary trends by defining a set of laws that rule the growth of software systems. Similarly, a study by Godfrey and German [43] compares software evolution to other kinds of evolution in a set of different domains, while Girba and Ducasse [44] propose a set of requirements for building evolution models. In his work, Mens [41] stresses the need to develop better predictive models for measuring and estimating the cost and effort of software maintenance and evolution activities with a higher accuracy. Evolution models are useful in software development, since being able to estimate the evolution of a software product, could provide valuable insight for its quality as well.

According to ISO/IEC 25010 [45], which is a well-accepted international standard on Software Quality, the notion of software quality is hierarchically decomposed into a set of quality attributes, like Maintainability, Reliability, and Security. A multitude of quality models have been proposed over the years allowing the assessment and/or prediction of these quality attributes individually [46] [47] [48] or of the overall quality itself [49] [50]. For instance, in [46] a model based on Bayesian Belief Networks is implemented for assessing and predicting the Maintainability of a software application based on a set of software metrics. Similarly, Van Koten et al.

[47] try to predict object-oriented software maintainability by applying a Bayesian network, while Zhou et al. [48] approach the same problem by using multivariate adaptive regression splines. Reliability Growth Models (RGMs) [51] also constitute representative examples of predictive models for software quality. These models typically use defect detection data or past observations of failures, which are collected during test and operation phases of the SDLC, to predict the future level of Reliability, expressed in terms of a number of failures.

As far as the quality attribute of Security is concerned, a large number of models for predicting the existence of vulnerabilities in software applications have been proposed over the years [52] [53] [54] [55]. For instance, Alhazmi et al. [52] use the density of the reported vulnerabilities of a software application to predict the number of actual vulnerabilities in future versions of the application. Similarly, in [53] the authors propose SAVI, a vulnerability indicator that predicts the application's post-release vulnerabilities, based on pre-release security-related static analysis results. Factors that are not directly related to software can be also leveraged for vulnerability prediction. For instance, in a relatively recent study, Roumani et al. [54] examine the relationship between the firms financial records (e.g., size, financial performance, sales, research and development expenditures etc.) and security vulnerabilities that may exist in their software products, revealing a strong association between these two factors.

Since quality attributes are relatively abstract and difficult to be measured directly from the artifacts of software products (e.g., source code), ISO/IEC 25010 [45] further decomposes them into a set of more concrete quality properties (e.g., complexity), which can be directly quantified through common metrics (e.g. McGabes Cyclomatic Complexity [56]). Similarly, to the high-level quality attributes, a large number of methods have been proposed to estimate the future evolution of software quality properties. The majority of these methods try to approach the subject by applying forecasting models on individual software properties based on the analysis of available information (historical data, trends, source code metrics, etc.). For instance, Fontana et al. [7] compare various machine learning techniques for code smell detection. In another study, Basili et al. [57] apply Logistic Regression for the validation of object-oriented design metrics, while Arisholm et al. [8] use Principal Component Analysis to predict software errorproneness of software components. Moreover, Yazdi et al. [58] try to model the evolution of the design of software systems by applying ARMA Time Series. Finally, in a recent study Chaicalis and Chatzigeorgiou [9] employ Network Models to forecast software evolution trends of Java systems.

The multitude of models that are available in the literature for predicting the evolution of specific quality attributes and quality properties reveal the importance of quality prediction and forecasting in the software engineering community. Since TD is an indicator of software quality (with an emphasis on maintainability), predicting its future value is considered equally important. However, although many studies have focused on the evolution of software systems, only a few have focused on the evolution of TD [59]. In fact, to the best of our knowledge the only known study on TD forecasting is [60], in which, Scourletopoulos et al. attempt to introduce the concept of predicting TD for Software as a Service (SaaS) systems, by exploiting COCOMO, a software cost model proposed by Boehm [61]. However, their study is limited only to cloud computing systems.

The need for knowing the evolution of TD has been highlighted by a recent study, in which Ampatzoglou et al. [21] link software maintainability with the notion of TD, while stressing the need for project managers to be able to preserve a software product maintainable for as long as possible. For that purpose, the authors propose the term breaking point, which refers to the point in time where the accumulated interest will be equal to the TD principal, i.e., the cost becomes higher than the benefit [62], thus providing managers with an insightful decision-making tool. Hence, forecasting the evolution of TD principal and interest could be valuable for estimating the point in which the software product could become unmaintainable.

## III. Open Issues and Contributions

Despite the multitude of methods proposed in the bibliography for the estimation of TD, there are still many open issues that require further investigation. First of all, none of the already proposed methods and tools have reached a desired level of maturity, and according to recent studies [5], there is no commonly accepted standard for estimating and managing TD. As a result, developers and managers perceive the concept of TD in different ways, while current methods and tools are not able to map software quality attributes to TDM activities. Moreover, the majority of well-established TD estimation methods, including the widely used SQALE method [6], mainly analyze the source code of the software. There is a large volume of potential metrics and techniques that have not been used yet for estimating TD, and which could potentially increase the completeness of the TD estimation concept. In addition, most of the already existing tools provide different TD indexes [63], creating confusion in the community about which of the current metrics should be selected, or how they should be combined [64].

Therefore, an interesting topic would be to investigate whether the combination of software-related metrics extracted from repositories and already existing TD estimation techniques and tools may lead to better and more accurate TD estimation methods. In addition, having in mind that new metrics and techniques for TD are emerging rapidly [65], there is a need for a single tool that combines software metrics and TD estimation techniques implemented by different approaches.

Another critical issue is that no particular approaches have been proposed for the forecasting of TD, which is opposite to extensive research that has been performed for predicting the evolution of individual software features or quality attributes that are directly or indirectly related to the TD of a software project, such as code smells [7], fault-proneness [8] and evolution trends [9]. A contribution to this challenge has high value since TD forecasting could lead to the development of practical decision-making mechanisms aiming to improve the TD repayment strategy. Furthermore, the decision making mechanisms should be integrated into an application or a tool to facilitate efficient identification of the aspects that might cause potential TD accumulation.

Hence, another interesting topic is whether the combination of software-related metrics and already existing software evolution approaches, along with existing forecasting methods could lead to the development of novel models that provide predictions about the evolution of a softwares future TD. Towards this goal, statistical methods such as causal models (including the widely used regression analysis) or time series models (including the widely used ARIMA model) [66] could be investigated. In addition, machine-learning models like Artificial Neural Networks (ANNs) [67] [68], regression trees, support vector regression and nearest neighbor regression [69] [70] could also be examined.

Last but not least, software repositories such as versioning, project management and issue-tracking systems, as well as archived communication between project personnel could be a potential source of TD related data. We believe that there is great potential in mining this information to extract software related metrics and thus, unveil ways that can help to support the development of better TD estimation and prediction methods. In fact, we believe that by analyzing multiple sources of information and predicting the evolution of TD on specific software artifacts, triangulation can be achieved and yield more accurate estimates. To further ease and automate this process, a tool that would utilize multiple sources of information accompanying a software project by pairing existing TD estimation methods with specialized techniques for forecasting, code analysis, software evolution analysis and natural language processing could pave the way for the advance in the state of the art in this domain. By doing so, another important contribution to the research community would be to provide a highly balanced, publicly available dataset of TD related metrics that could be reused by future researchers for relevant studies and comparison or validation of TDM methods and tools.

## IV. Conclusions and Future Work

In the present study, we investigated the state-of-the-art and examined the major contributions that have been made until today in the field of TD estimation and forecasting. Through our study, we identified some interesting open issues that should be addressed through further research. In particular, already existing methods and tools for TD estimation have not reached a satisfactory level of maturity yet, while there is still a large volume of potential metrics and techniques that have not been used and that could potentially increase the completeness of the TD estimation concept. In addition, although there has been extensive research with respect to predicting the evolution of individual software features, quality attributes, and quality properties that are directly or indirectly related to the TD of a software project, no concrete contributions exist in the related literature regarding TD forecasting.

Therefore, the improvement of already existing TD estimation methods, by incorporating previously uninvestigated software-related factors with potential relevance to TD is an interesting direction for future research. Another interesting topic would be to investigate different efficient ways to produce TD forecasting models for accurate prediction of TD principal and interest evolution. In addition, it would be useful to examine if TD forecasting could foster the development of high-quality software products. To the best of our knowledge, this is the first study that raises the awareness of the gap in the field of TD, regarding methods, tools, and techniques for forecasting the evolution of TD principal and interest.

The aforementioned identified open issues are expected to be addressed by the work conducted within the scope of the ongoing European project SDK4ED. Under this prism, we aim to cover the existing gap in the field by deploying a toolbox that combines various software metrics with TD estimation, forecasting and decision-making mechanisms for assisting developers and project managers in taking proactive actions regarding TD repayment. This toolbox will be developed by scientific partners and then evaluated by industrial partners within the SDK4ED context.

## Acknowledgment

This work is partially funded by the European Union's Horizon 2020 Research and Innovation Programme through SDK4ED project under Grant Agreement No. 780572.

## References

[1] W. Cunningham, "The wycash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.
[2] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya *et al.*, "Managing technical debt in software-reliant systems," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 2010, pp. 47–52.
[3] C. Seaman and Y. Guo, "Measuring and monitoring technical debt," in *Advances in Computers*. Elsevier, 2011, vol. 82, pp. 25–46.
[4] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "The financial aspect of managing technical debt: A systematic literature review," *Information and Software Technology*, vol. 64, pp. 52–73, 2015.
[5] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, vol. 101, pp. 193–220, 2015.
[6] J.-L. Letouzey and M. Ilkiewicz, "Managing technical debt with the sqale method," *IEEE software*, vol. 29, no. 6, pp. 44–51, 2012.
[7] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.
[8] E. Arisholm and L. C. Briand, "Predicting fault-prone components in a java legacy system," in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*. ACM, 2006, pp. 8–17.
[9] T. Chaikalis and A. Chatzigeorgiou, "Forecasting java software evolution trends employing network models," *IEEE Transactions on Software Engineering*, vol. 41, no. 6, pp. 582–602, 2015.
[10] M. Fowler. (2003) Technical debt. [Online]. Available: http://www.martinfowler.com/bliki/TechnicalDebt.html
[11] M. Fowler. (2009) Technical debt quadrant. [Online]. Available: http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html
[12] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *Ieee software*, vol. 29, no. 6, pp. 18–21, 2012.
[13] S. McConnell. (2012) How to categorize and communicate technical debt. [Online]. Available: https://www.castsoftware.com/blog/steve-mcconnell-on-categorizing-managing-technical-debt

[14] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.

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

[16] C. Sterling, *Managing Software Debt: Building for Inevitable Change (Adobe Reader)*. Addison-Wesley Professional, 2010.

[17] N. A. Ernst, "On the role of requirements in understanding and managing technical debt," in *Proceedings of the Third International Workshop on Managing Technical Debt*. IEEE Press, 2012, pp. 61–64.

[18] N. Zazworka, C. Seaman, and F. Shull, "Prioritizing design debt investment opportunities," in *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 39–42.

[19] Y. Guo and C. Seaman, "A portfolio approach to technical debt management," in *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 31–34.

[20] A. Chatzigeorgiou, A. Ampatzoglou, A. Ampatzoglou, and T. Amanatidis, "Estimating the breaking point for technical debt," in *Managing technical debt (mtd), 2015 ieee 7th international workshop on*. IEEE, 2015, pp. 53–56.

[21] A. Ampatzoglou, A. Michailidis, C. Sarikyriakidis, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "A framework for managing interest in technical debt: An industrial validation," 2018.

[22] A. Ampatzoglou, A. Ampatzoglou, P. Avgeriou, and A. Chatzigeorgiou, "Establishing a framework for managing interest in technical debt," in *5th International Symposium on Business Modeling and Software Design, BMSD*. Citeseer, 2015.

[23] C. Fernández-Sánchez, J. Garbajosa, C. Vidal, and A. Yagüe, "An analysis of techniques and methods for technical debt management: a reflection from the architecture perspective," in *Software Architecture and Metrics (SAM), 2015 IEEE/ACM 2nd International Workshop on*. IEEE, 2015, pp. 22–28.

[24] C. Seaman, Y. Guo, C. Izurieta, Y. Cai, N. Zazworka, F. Shull, and A. Vetrò, "Using technical debt data in decision making: Potential decision approaches," in *Proceedings of the Third International Workshop on Managing Technical Debt*. IEEE Press, 2012, pp. 45–48.

[25] J. Highsmith. (2010) The financial implications of technical debt. [Online]. Available: http://jimhighsmith.com/the-financial-implications-of-technical-debt/

[26] J. Holvitie and V. Leppänen, "Debtflag: Technical debt management with a development environment integrated tool," in *Proceedings of the 4th International Workshop on Managing Technical Debt*. IEEE Press, 2013, pp. 20–27.

[27] E. Alzaghoul and R. Bahsoon, "Cloudmtd: Using real options to manage technical debt in cloud-based service selection," in *Managing Technical Debt (MTD), 2013 4th International Workshop on*. IEEE, 2013, pp. 55–62.

[28] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the size, cost, and types of technical debt," in *Proceedings of the Third International Workshop on Managing Technical Debt*. IEEE Press, 2012, pp. 49–53.

[29] J. Shore. (2006) Quality with a name. [Online]. Available: http://jamesshore.com/Articles/Quality-With-a-Name.html

[30] O. Gaudin, "Evaluate your technical debt with sonar," *Sonar, Jun*, 2009.

[31] J. Bohnet and J. Döllner, "Monitoring code quality and development activity by software maps," in *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 9–16.

[32] A. Nugroho, J. Visser, and T. Kuipers, "An empirical model of technical debt and interest," in *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 1–8.

[33] J. de Groot, A. Nugroho, T. Bäck, and J. Visser, "What is the value of your software?" in *Proceedings of the Third International Workshop on Managing Technical Debt*. IEEE Press, 2012, pp. 37–44.

[34] S. Strasser, C. Frederickson, K. Fenger, and C. Izurieta, "An automated software tool for validating design patterns," in *ISCA 24th International Conference on Computer Applications in Industry and Engineering. CAINE*, vol. 11, 2011.

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

[36] J.-L. Letouzey, "The sqale method for evaluating technical debt," in *Managing Technical Debt (MTD), 2012 Third International Workshop on*. IEEE, 2012, pp. 31–36.

[37] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In search of a metric for managing architectural technical debt," in *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*. IEEE, 2012, pp. 91–100.

[38] 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, 2012.

[39] C. Fernández-Sánchez, H. Humanes, J. Garbajosa, and J. Díaz, "An open tool for assisting in technical debt management," in *Software Engineering and Advanced Applications (SEAA), 2017 43rd Euromicro Conference on*. IEEE, 2017, pp. 400–403.

[40] H. C. Gall and M. Lanza, "Software evolution: analysis and visualization," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 1055–1056.

[41] T. Mens, "Introduction and roadmap: History and challenges of software evolution, chapter 1. software evolution," 2008.

[42] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[43] M. W. Godfrey and D. M. German, "The past, present, and future of software evolution," in *Frontiers of Software Maintenance, 2008. FoSM 2008.* IEEE, 2008, pp. 129–138.

[44] T. Gîrba and S. Ducasse, "Modeling history to analyze software evolution," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 3, pp. 207–236, 2006.

[45] I. O. for Standardization/International Electrotechnical Commission *et al.*, "Iso/iec 25010; systems and software engineering-systems and software quality requirements and evaluation (square)-system and software quality models"," *Authors, Switzerland Google Scholar*, 2011.

[46] S. Wagner, "A bayesian network approach to assess and predict software quality using activity-based quality models," *Information and Software Technology*, vol. 52, no. 11, pp. 1230–1241, 2010.

[47] C. Van Koten and A. Gray, "An application of bayesian network for predicting object-oriented software maintainability," *Information and Software Technology*, vol. 48, no. 1, pp. 59–67, 2006.

[48] Y. Zhou and H. Leung, "Predicting object-oriented software maintainability using multivariate adaptive regression splines," *Journal of Systems and Software*, vol. 80, no. 8, pp. 1349–1361, 2007.

[49] S. Wagner, A. Goeb, L. Heinemann, M. Kläs, C. Lampasona, K. Lochmann, A. Mayr, R. Plösch, A. Seidl, J. Streit *et al.*, "Operationalised product quality models and assessment: The quamoco approach," *Information and Software Technology*, vol. 62, pp. 101–123, 2015.

[50] M. G. Siavvas, K. C. Chatzidimitriou, and A. L. Symeonidis, "Qatch-an adaptive framework for software product quality assessment," *Expert Systems with Applications*, vol. 86, pp. 350–366, 2017.

[51] J. D. Musa, *Software reliability engineering: more reliable software, faster and cheaper*. Tata McGraw-Hill Education, 2004.

[52] O. H. Alhazmi, Y. K. Malaiya, and I. Ray, "Measuring, analyzing and predicting security vulnerabilities in software systems," *Computers & Security*, vol. 26, no. 3, pp. 219–228, 2007.

[53] J. Walden and M. Doyle, "Savi: Static-analysis vulnerability indicator," *IEEE Security & Privacy*, no. 1, 2012.

[54] Y. Roumani, J. K. Nwankpa, and Y. F. Roumani, "Examining the relationship between firms financial records and security vulnerabilities," *International Journal of Information Management*, vol. 36, no. 6, pp. 987–994, 2016.

[55] M. Siavvas, D. Kehagias, and D. Tzovaras, "A preliminary study on the relationship among software metrics and specific vulnerability types."

[56] Y. Shin and L. Williams, "Is complexity really the enemy of software security?" in *Proceedings of the 4th ACM workshop on Quality of protection*. ACM, 2008, pp. 47–50.

[57] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on software engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[58] H. S. Yazdi, M. Mirbolouki, P. Pietsch, T. Kehrer, and U. Kelter, "Analysis and prediction of design model evolution using time series," in *International Conference on Advanced Information Systems Engineering*. Springer, 2014, pp. 1–15.

[59] G. Digkas, M. Lungu, A. Chatzigeorgiou, and P. Avgeriou, "The evolution of technical debt in the apache ecosystem," in *European Conference on Software Architecture*. Springer, 2017, pp. 51–66.

[60] G. Skourletopoulos, C. X. Mavromoustakis, R. Bahsoon, G. Mastorakis, and E. Pallis, “Predicting and quantifying the technical debt in cloud software engineering.” in *CAMAD*, 2014, pp. 36–40.

[61] B. W. Boehm *et al.*, *Software engineering economics*. Prentice-hall Englewood Cliffs (NJ), 1981, vol. 197.

[62] A. Ampatzoglou, A. Ampatzoglou, P. Avgeriou, and A. Chatzigeorgiou, “A financial approach for managing interest in technical debt,” in *International Symposium on Business Modeling and Software Design*. Springer, 2015, pp. 117–133.

[63] F. A. Fontana, R. Roveda, and M. Zanoni, “Technical debt indexes provided by tools: a preliminary discussion,” in *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*. IEEE, 2016, pp. 28–31.

[64] N. Zazworka, C. Izurieta, S. Wong, Y. Cai, C. Seaman, F. Shull *et al.*, “Comparing four approaches for technical debt identification,” *Software Quality Journal*, vol. 22, no. 3, pp. 403–426, 2014.

[65] C. Fernández-Sánchez, J. Garbajosa, A. Yagüe, and J. Perez, “Identification and analysis of the elements required to manage technical debt by means of a systematic mapping study,” *Journal of Systems and Software*, vol. 124, pp. 22–38, 2017.

[66] J. Das, *Statistics for Business Decisions*. Academic Publishers, 2012.

[67] P. Werbos, “Beyond regression:” new tools for prediction and analysis in the behavioral sciences,” *Ph. D. dissertation, Harvard University*, 1974.

[68] P. J. Werbos, “Generalization of backpropagation with application to a recurrent gas market model,” *Neural networks*, vol. 1, no. 4, pp. 339–356, 1988.

[69] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*. Springer series in statistics New York, NY, USA:, 2001, vol. 1, no. 10.

[70] E. Alpaydin, “Introduction to machine learning, 2nd edn. adaptive computation and machine learning,” 2010.