Technical Debt in Service-Oriented Software Systems

Nikolaos Nikolaidis[1], Apostolos Ampatzoglou[1], Alexander Chatzigeorgiou[1], Sofia Tsekeridou[2], and Avraam Piperidis[2]

[1] University of Macedonia, Thessaloniki, Greece
[2] NetCompany-Intrasoft, Athens, Greece
nnikolaidis@uom.edu.gr, a.ampatzoglou@uom.edu.gr, achat@uom.edu.gr, sofia.tsekeridou@netcompany-intrasoft.com, avraam.piperidis@netcompany-intrasoft.com

Abstract. Service-Oriented Architectures (SOA) have become a standard for developing software applications, including but not limited to cloud-based ones and enterprise systems. When using SOA, the software engineers organize the desired functionality into self-contained and independent services, that are invoked through end-points (API calls). At the maintenance phase, the tickets (bugs, functional updates, new features, etc.) usually correspond to specific services. Therefore, for maintenance-related estimates it makes sense to use as unit of analysis the service-per se, rather than the complete project (too coarse-grained analysis) or a specific class (too fine-grained analysis). Currently, some of the most emergent maintenance estimates are related to Technical Debt (TD), i.e., the additional maintenance cost incurred due to code or design inefficiencies. In the literature, there is no established way on how to quantify TD at the service level. To this end, in this paper, we present a novel methodology to measure the TD of each service considering the underlying code that sup-ports the corresponding endpoint. The proposed methodology relies on the method call graph, initiated by the service end-point, and traverses all methods that provide the service functionality. To evaluate the usefulness of this approach, we have conducted an industrial study, validating the methodology (and the accompanying tool) with respect to usefulness, obtained benefits, and usability.

Keywords: technical debt, service analysis, endpoint analysis, quality

1 Introduction

The notion of Technical Debt (TD) was introduced by Ward Cunningham [1] to describe the shipment of first-time code with inefficiencies, due to early deployment. To quantify the amount of technical debt, various types of TD and ways of identification / quantification have been proposed in the literature. Since one of the most recognized types of TD, both in industry [2] and academia [3] is the code TD, a significant number tools [4] have been developed to quantify code TD: i.e., identify code inefficiencies and estimate the required effort for fixing them. The main ***mechanism beneath code TD identification / quantification is source code static analysis***, pointing to classes that violate certain pre-defined quality rules. For example, calculate specific

metric scores (e.g., cognitive complexity or lines of code) that when surpass a certain threshold an inefficiency is recorded.

One of the most known tools for TD identification / quantification is SonarQube, which is able to quantify the technical debt of projects written in almost any programming language. SonarQube counts the number of inefficiencies and calculates the remediation time that is needed to bring the code to an optimum (or near-optimum) state. According to Tamburri et al. [5] this process is, and should be, a continuous practice, since the concept of an optimum state is constantly changing. However, despite the support for various languages and programming paradigms, the ***approach for the quantification of technical debt remains unchanged, regardless of the system architecture*** (e.g., whether the software is service-based or monolithic).

Lately, the Service-Oriented Architecture (SOA) has become quite popular due to its ability to create quick and easy applications by using existing micro-services [6]. In the SOA model, services form self-contained units of software that communicate across different platforms and languages to form applications. Communication takes place through their end-points, while a loose coupling is promoted to either pass data or coordinate activities. Several studies have assessed different kinds of technical debt quantification in SOA, introducing several approaches [7][8][9]. Most of these ***research approaches treat each service as a black-box*** and quantify TD, based on the effort to compose these services. For instance, by focusing on their interface: e.g., the amount of exchanged data, the different types of data, the number of different services, etc. On the other hand, on the limited cases that TD quantification treats services as a white-box, the amount of TD is calculated again with SonarQube (or similar tools)—***considering as the unit of analysis either the whole project or isolated classes***, without taking into account the fact that the project's architecture differs substantially than a software project that is not based on services.

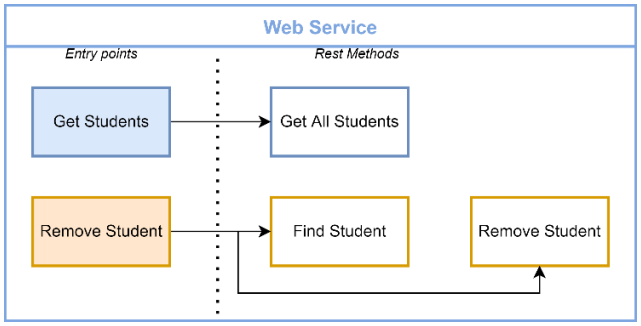

Fig. 1. Generic SOA Structure Example.

To address the specificities of developing service-based systems, when it comes to maintainability assessment, our paper introduces the SmartCLIDE approach that is tailored for such systems. One of the most prominent characteristics of service-based systems is that they encapsulate, to some extent, traceability in their design: i.e., they ***offer end-points that deliver a very specific functionality, and this stands for an entry point to the method call sequence in the code that offers this functionality***–see

Fig. 1. By considering that maintenance tickets, coming either from the customer or the preventive maintenance team, are usually describing the problem in a natural language [10], it becomes evident that a specific requirement can be easily spotted. Next, given the identification of the corresponding end-point, the effort estimation for serving the ticket can be more accurately performed, by focusing only on the parts of the system that are successively invoked through the end-point. In other words, ***using end-points as the units of analysis for technical debt quantification, in service-based software system, seems as a promising and more accurate alternative compared to working at the class- or at the system-level***. As a first step towards this approach, we have developed a methodology and a supporting tool (in the form of Eclipse Theia extension) that quantifies the amount of TD that a developer will face when performing maintenance to all parts of the system that are invoked through a specific end-point of a given web service. To assess the TD at the method level, we relied on SonarQube so as to identify code inefficiencies, reuse the estimated remediation time, enabling the TD quantification of each endpoint. To validate both the methodology and the accompanying tool, we conducted an empirical study, involving 15 developers, working in 5 companies, spread across Europe. The evaluation targeted the exploration of current approaches for TD quantification in SOA systems, the validation of the approach, and the usability assessment of the tool.

2 Related Work

Technical Debt Quantification: One of the most known studies in the field of TD management has been performed by Alves et al. [11], who performed a systematic mapping study on over 100 primary studies. Alves et al. described the different types of TD and the strategies for identifying TD items. Based on their results, it becomes evident that the majority of the top indicators of TD presence is the existence of "code smells". The same conclusion has also been validated by the first secondary study on TD management, by Li et al. [12]. The identification of code smells is the basis of TD quantification, by the majority of TD tools. Due to existence of various such tools, a recent direction has pursued the identification of the best and more accurate one [4] [12] [13]. Even though there have been a lot of studies and even methodologies that combine different tools [14], there isn't still a consensus on the quantification of TD. Nevertheless, according to a recent analysis of TD quantification tools [4], it seems that the most popular TD quantification tool is SonarQube [15], which is widely known to track the quality and maintainability of source code. The tool quantifies TD by multiplying the number of issues from each category, with the time that is needed to resolve these issues. To provide TD in monetary terms, effort in time can be multiplied with an average man-hour cost. The origins of SonarQube lie on the SQALE method, to provide a pyramid of issues that aids in TD prioritization.

Technical Debt in SOA and Services: Bogner et al. [16] conducted a large-scale survey in order to find the TD management techniques used in service- and micro-service-based systems. The results suggested that SonarQube is the most used tool, followed by FindBugs. Moreover, 67% of the participants have responded that they

do not treat maintainability differently, compared to non-service-based software development. However, an important fraction of the participants, mentioned that they should. Finally, 26% of the participants apply somewhat different controls, and approximately 7% mentioned significantly different treatments. Therefore, the authors concluded that it is very important to distinguish between the service-based and the non-service systems. Additionally, based on other studies, it becomes evident that the only analyzed code part of the web services is the interface of the different web services [17] [18]. For instance, Ouni et al. [17] created a machine learning approach to detect defects in the interfaces of web services. The results suggested that this methodology is promising for specific types of services; however, for others (e.g., REST) the methodology is not applicable. Regarding the TD management of microservice-based application, we can find a variety of studies that are focused on defining what can be considered as technical debt in SOA and how it can be quantified. For the quantification of TD in service-based systems all related studies focus outside the code of each service and explore the composition of the services [19]. For instance, de Toledo et al. [8] organized an industrial case study to identify (among others) what is TD in SOA. The results of the study suggested that, as TD we can characterize: (a) the existence of too many point-to-point connections among services; (b) the insertion of business logic inside the communication layer; (c) the lack of a standard communication model; (d) the weak source code and knowledge management for different services; and (e) the existence of different middleware technologies. Even though these issues are related more to the composition of services, it is clear that there is a need for TD evaluation, within the source code of each service. Similarly, the study by Pigazzini et al. [7] suggests that the existence of: (a) cyclic dependency; (b) hard-coded end-points; and (c) shared persistence can be characterized as indicators of poor quality in SOA. Nevertheless, we need to again make clear that for this study each end-point is treated as a black box, even though it's internals are critical in case of future changes. Furthermore, Taibi et al. [9] conducted a similar study and reported additional quality indicators. Among them, as the most important ones, the authors characterize: (a) hardcoded endpoints; (b) no API-gateway; (c) inappropriate service intimacy; and (d) cyclic dependency. Finally, some studies report as the main TD indicator the easiness of changing a microservice for another [9][19].

3 SmartCLIDE Approach for Calculating TD of Services

In this section, we present the proposed approach for quantifying TD in service-based applications, bringing two important advancements, compared to the state-of-the-art: (a) in our approach services are not treated as black-boxes; and (b) we refine the unit of analysis from the project or class level, to the service level.

The ***SmartCLIDE methodology*** for quantifying the amount of TD that is accumulated within software services is straightforward. Since a service has a number of different entry points (e.g., end-points), we propose that each one of these end-points deserves its own evaluation of TD. In a way, each entry point could be treated as a different application, since it provides a distinct functionality. The benefit lies in the fact that

TD analysis can highlight individual end-points which are in need of improvement, rather than blaming the entire project. The methodology is based on the generation of the call graph of the services end-points. With the term call graph, we refer to the user-defined methods that are being called successively from a given point in the code. This information is critical in order to report only the TD issues appearing in the methods invoked by the given end-point. By knowing the total TD issues reported for all the invoked methods, we are able to quantify the amount of TD that a developer will face, when maintaining the specific end-point from end-to-end, as effort (time) and in monetary terms. To be able to quantify the TD of each endpoint, we had to overcome two major challenges. First, the call graph construction should be initiated from a given starting point (e.g., method). To resolve this issue, the code of the target project should be parsed. To this end, we used the `JavaParser` library [20] which is a very well-known parsing library for Java projects (the downside is that currently only projects written in the Java programming language are supported). Given a source file, or in our case a project, the different syntactic elements are recognized and an `Abstract Syntax Tree` (AST) is generated. This AST is then analyzed by the `JavaSymbolSolver` and locates the declarations associated with each element. We should also note that `JavaParser` makes use of the Visitor design pattern to traverse the created AST and execute the desired operation on the visited nodes. In particular, we developed a new Visitor that finds the annotations of each user-declared method, to identify the methods that all end-points start from. We have been able to find the end-points of projects that use the `JAX-RS` specification, or the `Spring Boot framework`. Once the end-points of the project are known, we then created a Visitor that finds all the methods that are being called successively. For each method, we retain the file path as well as the methods' starting and ending lines.

Illustrative Example: In this subsection, we present an illustrative example through an open-source Java e-commerce software. The `Shopizer`[1] project contains a large number of endpoints as it exposes its functionalities through a RESTful API. As a first step, we analyzed the entire project with SonarQube. The project-level analysis yields all code inefficiencies and the time that is needed to resolve them, as follows:

```
TD: 478.8h / 14362.5€
Number of Issues: 3426
```

By applying the proposed methodology, we were able to map the total TD to the project end-point, and identified cases for which the total number is not representative. Below, we report our calculations for two endpoints: namely, `Shipping Modules` and `List Permissions`. The call graphs for the two end-points are presented in Fig. 2 and Fig. 3, respectively. By applying the proposed methodology for the two cases, we have calculated TD, as follows:

```
Shipping Modules
  TD: 22min / 10.5€
  Number of Issues: 3
```

```
List Permissions
  TD: 0min / 0€
  Number of Issues: 0
```

[1] https://github.com/shopizer-ecommerce/shopizer

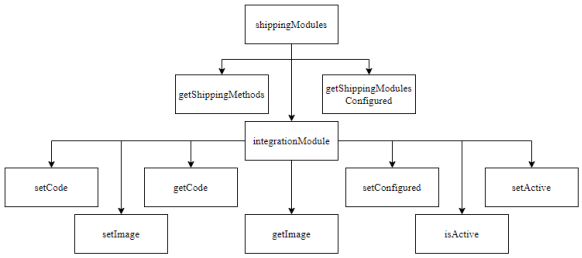

Fig. 2. Shipping Modules Example Call Graph

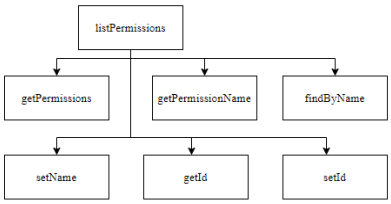

Fig. 3. List Permissions Call Graph.

It goes without saying that by focusing TD quantification at the end-point level, a more accurate information is provided, allowing stakeholders to take more informed decisions: while the entire project appears to suffer from a large number of issues, the `List Permissions` end-point is code technical debt free (no TD-related issues); whereas the `Shipping Modules` end-point is responsible only for the very limited amount of three technical debt-related issues, not raising any alarm for the maintenance team. Therefore, we argue that differentiating between the healthy and problematic parts of a service-based project can help developers prioritize their refactorings, improve their effort estimation, and improve their decision-making processes.

SmartCLIDE Eclipse Theia Extension: The proposed approach for quantifying the TD of ser-vice-based systems is part of the SmartCLIDE project, and has been integrated into the SmartCLIDE IDE, as an Eclipse Theia Extension. Eclipse Theia is a web-based IDE that acts as a code editor, in which a developer is able to create and add extra functionalities as extensions or plugins. We chose to create an extension, due to the extra available functionalities that it offers, and the customizable User Interface (UI). With the only drawback being that the extensions in Theia cannot be dynamically added, as Theia needs to be built from the beginning with the selected extensions. But since the proposed approach is part of the SmartCLIDE IDE, this is not an important drawback, in the sense that the Theia image needs only to be built once, and then installed with the desired functionality. Moreover, with respect to the backend part of the system that undertakes the actual analysis, we have developed a web service that exposes all the necessary functionalities via a RESTful API.

The SmartCLIDE Eclipse Theia extension is an official Eclipse Research Lab project, and is freely available through the Eclipse Research Labs Git repository[2]. For research purposes, we provide online an already deployed instance of the Eclipse Theia, containing the proposed extension[3]. Once the Eclipse Theia is opened, the extension can be reached under the menu "`View`" through the "`SmartCLIDE TD and Reusability`" option. After that, the user is able to provide the URL of the `git` repository that he / she wants to analyse (GitHub, GitLab, etc.). Also, the project key of the SonarQube installation is required so that a new analysis is generated according to the `git` owner and `git` name (e.g., `owner:name`). By providing these values the

[2] https://github.com/eclipse-researchlabs/smartclide-td-reusability-theia
[3] http://195.251.210.147:3131

user is able to start a new analysis for a new project or load an existing analysis, through the "`New Analysis`" and "`Project Analysis`" buttons.

In Figure 4, we present the analysis, i.e., the amount of TD for the entire project along with all the reported issues (SonarQube). By starting an "`Endpoint Analysis`" the user can take advantage of the proposed approach. The GUI action, enables the backend service, which locates all the end-points of the provided project and presents the results for them (Figure 5). The extension is populated with all the end-points, along with the quantified TD amount for each service, as well as the number of issues that have been identified in the method call chain for each end-point. Finally, if the user expands a specific end-point, he / she will get access to the list of issues related to the specific endpoint, along with their criticality and remediation time.

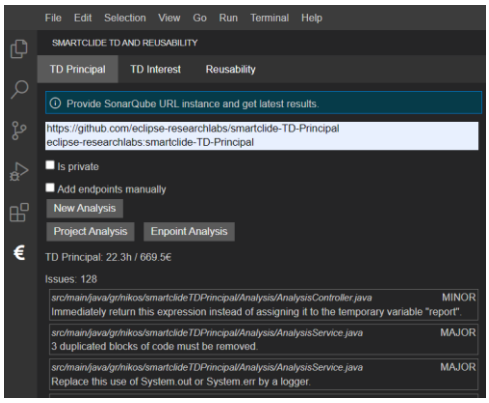

Fig. 4. Tool Instance of Project Analysis

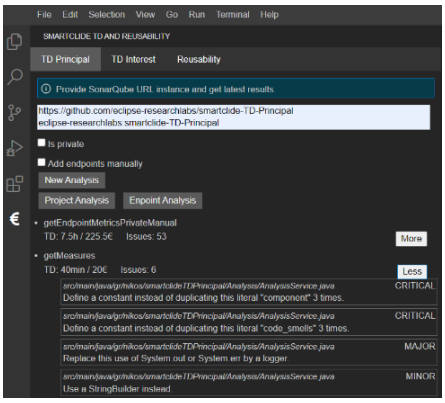

Fig. 5. Tool Instance of Endpoint Analysis.

Finally, through the Eclipse Theia Extension, the user can specify one or more end-points, getting results only on the selected endpoints—see "`Add endpoints manually`" in Figure 6: providing the method and filename. We should note that since a method name is not sufficient (there can be many methods with the same name), we require the full method signature. The method signature has the following format:

```
[accessSpecifier] [static] [abstract] [final] [native] [synchronized]
returnType methodName ([paramType [paramName]]) [throws exceptionsList]

private void getGivenEndpointsFromAllFiles
(List<RequestBodyEachEndpoint> requestBodyEachEndpointList) throws File-
NotFoundException, IOException
```

4 Validation Study Design

In this section we report the protocol that we have followed for validating the proposed approach and the accompanying tool, in an industrial setting. To evaluate the current status of TD quantification in software-based systems, as well as the proposed approach and tool we have performed an empirical study, designed and reported, based on the guidelines of Runeson et al. [21].

Research Objectives: The goal of the validation is three-fold: (a) explore and assess the current state-of-practice for approaches and tools that can be used to quantify the TD of service-based systems; (b) validate the proposed approach; and (c) evaluate the developed Eclipse Theia Extension; leading to three research questions:

[RQ1] What are the adopted approaches and tools for TD quantification in service-based software development, in the context of the studied organizations?

[RQ2] How is the proposed approach evaluated by practitioners in terms of usefulness and ease-of-usage?

[RQ3] Does the developed endpoint-level analysis tool meet the expectations of the practitioners?

Study Design: The validation study was conducted with the participation of 15 developers in the field of software services. The group of developers is spread to 5 software industries across Europe (2 Large and 3 Small-Medium Enterprises). In terms of demographics: 4 participants have a low experience as backend developers (with about 1-3 years of experience), 4 of them are medium experienced (with about 4-8 years of experience) and 7 are highly experienced (with over 8 years of experience). The validation study was conducted in the form of a half-day workshop—see Table 1. The same workshop structure was replicated 5 times, one for every involved industry.

Table 1. Workshop Activities.

Activity	Duration
Introduction / Goals of the workshop	20 minutes
Interviews for RQ_1	20 minutes (for each participant)
Demonstration of end-point level analysis	20 minutes
Break	20 minutes
Interviews for RQ_2	30 minutes (for each participant)
Task RQ_3	20 minutes (for each participant)
Interviews for RQ_3	30 minutes (for each participant)

First, the researchers' team has first given a short introduction of TD in case some of the participants were not familiar with the terminology. Subsequently, the first part of the workshop that was aiming to understand the current practices, along with their benefits and limitations, was carried out so as to get acquainted with the TD practices that our study participants are aware of and use. The data collection was made in the form of interviews. At this point we need to note that a major parameter for opening up the workshop to 5 industries was the nature of RQ1, which required a broad recording of practices in SOA-based projects, obtained by various companies. For better organization and time monitoring, we have split each interview into three blocks. The questions asked during the interviews can be found as supplementary material[4].

Being aware of the current technical debt quantification status, we continued by presenting the proposed approach and the developed Eclipse Theia Extension. We demonstrated the approach and tool on how to quantify the technical debt amount for each end-point of the service, along with the examples presented in Section 3. Assur-

[4] https://www.dropbox.com/s/vagrr2wdc9p6nu9/SupplementaryMaterial.pdf?dl=0

ing that all participants have understood in sufficient detail the SmartCLIDE approach for TD quantification, and after a short break, we proceeded to the second round of interviews, aiming to shed light on RQ2. Finally, in order to assess the usability of the Eclipse Theia Extension, we needed to involve the participants in a simple task, so that they get a hands-on experience with the tool. In particular, each participant was asked to do the following actions through the Eclipse Theia Extension, for a private service of their own: (a) get and check the project analysis report; (b) get and check the end-point analysis report for all the endpoints; and (c) provide one or more endpoints and get and check the analysis report. After completing the given task, the subjects participated in a final interview round, aiming to the evaluation of the functionality and usability of the tool (answer RQ_3). The first block of this interview guide was related to functionality, whereas the second to usability. Usability evaluation was performed, based on the System Usability Scale (SUS) instrument [22].

Data Collection: To validate the proposed TD quantification approach and the corresponding tool implementation, we have relied on both quantitative and qualitative analysis. To synthesize qualitative and quantitative findings, we have relied on the guidelines provided by Seaman [23]. On the one hand, to obtain ***quantitative results***, we employed descriptive statistics. For usability, we assessed the total SUS score, along with the most common scales for interpretation, in terms of acceptance, adjective, and grade. On the other hand, to obtain the ***qualitative assessments***, we use the interviews data, which we have analyzed based on the Qualitative Content Analysis (QCA) technique [24], which is a research method for the subjective interpretation of the content of text data through the systematic classification process of coding and identifying themes or patterns. In particular, we used open coding to create categories, and abstraction. To identify the codes to report, we used the Open-Card Sorting [25] approach. Initially we transcribed the audio file from the interviews and analyzed it along with the notes we kept during its execution. Then a lexical analysis took place: in particular, we have counted word frequency, and then searched for synonyms and removed irrelevant words. Then, we coded the dataset, i.e., categorized all pieces of text that were relevant to a particular theme of interest, and we grouped together similar codes, creating higher-level categories. The categories were created during the analysis process by both the 3rd and the 4th author, and were discussed and grouped together through an iterative process in several meetings of all authors. The reporting is performed by using codes (frequency table) and participants' quotes. Based on Seaman [23] qualitative studies can support quantitative findings by counting the number of units of analysis that certain keywords and compare the counts, or comparing the set of cases containing the keyword to those that do not.

5 Results and Discussion

In this section, we present the findings of our validation study, organized by research question. Along the discussion, codes are denoted with capital letters, where-as quotes in italics. In Table 2, we present the codes that have been identified along the interviews, accompanied by quotes and the number of participants that used them.

Table 2. Codes of the Qualitative Analysis.

Code	Quote	#
ACCUARCY	"I think it will be really accurate on the end-point understudy" "The challenge would be how it measures these endpoints with accuracy" "Having an approach like this, a developer can estimate the TD for the code path that is going to be executed for an end-point, or for any new feature" "it makes sense, I would try it" "…but is not reliable with the actual transform action that needs to be done: If I have 5 end-points with almost the same issues, I will see the same issues almost 5 times, and 1 fix will delete 5 issues"	14
NO SERVICE-SPECIFIC TD	"No, not really, we are not changing anything in SOA" "We are interested more in scalability and reliability, but no special treatment for TD"	12
PRIORITIZATION	"…lets developers focus only to specific end-points (with the higher TD)" "I think it's a very good approach. It offers the keys that any manager needs in which issues to focus first." "Seems more structured and easier to focus on end-points that matter most" "Measuring each endpoint's TD, as a standalone application would be really helpful for the developers to understand the complexity of each service, to keep an eye on the most complicated ones"	10
VISUALIZATION		8
TIME SAVINGS	"Probably requires less time to find issues" "if you are interested on specific functionalities its great because it saves you time to find the issues for a specific case"	6
USEFULNESS	"In terms of usability I think it's easy and helpful" "Seems simple and easy to use" "Its helpful to know how many issues you can met on each flow" "Seems easy and it make a lot of sense. It would be very helpful for services" "The other practices just perform a static code analysis and detect the code-level issues, but do not make the link to higher-level artifacts"	6
NEED FOR TAILORING	"We follow the same model, regardless of business needs or the architecture" "No specific metrics and indicators (e.g., specific SOA-related metrics)"	4
HIDDEN PARTS OF THE SYSTEM	"A limitation could be that complexity of hidden aspects of the end-point are ignored, such as related classes (i.e., entities or other utility classes)" "A limitation is that it will not count the functions and classes that are not called right now from end-points"	3
MONITORING	"The team could potentially watch the TD graph rate, as the project grows to monitor and deal with bad code early enough"	3
RATIONALLE	"…the TD analysis matches the thought process of developers" "...it is very close to what I do manually to check the quality the code, before maintenance"	2
QUALITY GATES	"…for each new end-point that is added, in terms of not exceeding the average (or any limit) of TD that the project has configured"	1

Current Approach for Service TD Measurement: The vast majority of the participants in the study (~85%) have performed TD quantification at least one time into their projects; more than half of them rely their assessment on SonarQube. Other options for TD assessment in the participants' group of our study are: SIG, CAST; whereas for more generic quality assessment the participants have used: code review without tooling, Jaccoco, and CheckStyle. In terms of unit of measurement, most of the participants' organizations are recording the number of issues, since only 2 (out of 15) are recording the monetary values of TD, as calculated by SonarQube. The TD management practices target both front- and back-end components, but the majority of the participants of the study (since our goal was to propose a SOA-based approach) are focusing on back-end software development technologies. Additionally, 75% of the participants are willing to (and usually do) apply the suggestions that they receive from the technical debt management tool; whereas the frequency of getting such suggestions vary from a daily basis to a one-off evaluation before the first release of the project. In terms of prioritization, several options have been discussed by the participants, such as based on the tool assessment of criticality, priority to custom rules based on organization's standards, SonarQube severity, build blocking issues only, etc. Finally, based on the policy of the involved organizations, technical debt assessment is performed by project managers, software architects, or team leaders. Regarding the assessment of TD in SOA projects, the participants have claimed that they are not following a different strategy (NO SERVICE-SPECIFIC TD), compared to "*traditional*" software development. With respect to more "*generic*" quality assessment one participant mentioned the different quality properties that are of interest, such as scalability, availability, resilience, etc., but such run-time quality properties usually do not fall in the context of TD Management. In general, the participants are satisfied with process, raising two concerns: "*We have to follow the given model regardless the business needs or the architecture, and sometime this produces delays or inaccuracies*" and "*No specific quality metrics and indicators (e.g., specific SOA-related metrics)*" suggesting the need for some tailoring of the process (NEED FOR TAILORING). On the positive side, the well-established generic (i.e., non-SOA related) benefits of technical debt management have been highlighted, such as "*Automated reports, and in some cases good catches*", "*Easy to explain and fix, with clear feedback*", "*The code repo is cleaner and more secure*", etc.

Evaluation of the Proposed Approach: This section reports on the evaluation of the proposed approach by the practitioners. More specifically, with respect to the accuracy of the results obtained by applying the approach, the mode response value was "*Accurate*" (4.0 out of 5.0), followed by "*Very Accurate*" (5.0 out of 5.0) given by 33% of the respondents. In total 86.6% of the practitioners characterized the results as "*Accurate*" or "*Highly Accurate*". In terms of usefulness 93% of the participants graded the approach as either "Useful" or "Very Useful" (mode value: "Very Useful"). Finally, in terms of industrial-readiness of the approach, and how frequently would the developers use it in their daily routine, the mode value was "Sometimes" and "Often". Notably, there were no responses for "Never" or "Very Often". One of the main points that have been raised by the participants was the ACCURACY of the results that are obtained through the approach. The majority of the participants were

positive in their evaluation on the accuracy of the obtained results (mentioning that "*they make sense*" or that "*they seem as very accurate*"), but some were hesitant: For instance, in terms of double-counting the same issue in the same class for the same end-point (e.g., if two end-points invoke the same method), or that accuracy in any kind (and with any tool) of TD quantification is challenging and probably not accurate. Also, the participants highlighted the USEFULNESS of the approach, e.g., in terms of TD items PRIORITIZATION and TD MONITORING. While discussing the main idea of the proposed approach, i.e., promoting the end-point as the main unit of analysis, some interesting findings have been identified. One of the most positive judgements on this part of the discussion was provided by a practitioner, describing the main benefit of the approach rationale, in terms of ACCURACY, as follows: "*The idea of measuring the TD from the entry point of a web service request is very nice, because any developer is aware of that, but Sonar cannot see that difference (compared to other types of projects). So having a tool like this, a developer can estimate the TD for the code path that is going to be executed for a single end-point, and even for any new feature*". In this round of discussion, it is important to note that no practitioner had a negative comment on the methodology, validating that the approach RATIONALE is conceptually very close to what a developer would do mentally. The importance of PRIORITIZATION management benefits, as well as in terms of bringing a highly systematic and structured process for TD items prioritization is service-based software development, was highlighted.

Finally, we discuss the benefits that the approach brings, compared to the state-of-practice approaches, as well as its limitations. In general, the participants were very positive on the proposed approach, and they acknowledge that it advances state-of-the-art, since "*it keeps all the benefits of existing tools and appends them*". A specific advancement compared to the state-of-the-art, as described by one practitioner suggests that this approach is not a simple source code analysis, but a more tailored and informative one: "*The benefit of the approach is that it focuses on the end-points and the source code that is required to execute them. The other practices just perform a static code analysis and detect the code-level issues, but do not make the very useful linking to higher-level artifacts*". Nevertheless, some participants mentioned that the proposed approach might lack in terms of ACCURACY since it does not evaluate some HIDDEN PARTS OF THE SYSTEM, such as "*related classes (i.e., entities or other utility classes)*", or "*functions and classes that are not called right now from end-points*", or "*configuration classes*". In addition, one practitioner brought up an interesting future feature of the approach that increases its potential. The suggestion would be to incorporate QUALITY GATES in the approach: i.e., check the quality of the code of every new end-point (compared to some pre-configured threshold) before the pull request before merging, and accept or reject it. Finally, one participant mentioned that the USEFULNESS of this approach might depend on the goals of the quality assessment: e.g., if the quality assurance team is interested in a panoramic view of the project quality, then the proposed approach is not useful. However, if the team wants to focus on some specific functionalities, the proposed approach "*is great because it saves you time to find the issues for a specific one case*" (TIME SAVINGS).

Evaluation of the Eclipse Theia Extension: In terms of ***Functional Suitability***, a high-level representation of the results is presented in Figure 6. The stacked bar chart corresponds to the evaluation in a 0 to 10 scale of each main functionalities offered by the tool. As we can see there are no features of the tool that have been evaluated with a grade lower than 5, by any participant. The features that were best received by the practitioners were the two ways of VISUALIZATION of the results (light blue and orange bars). The two ways of visualization (at service and at project level) have received a similar evaluation. The slightly more positive evaluation of the visualization at the system level can probably be attributed to the habitual tendency of developers to check quality assurance evaluations at the project level. Nevertheless, the fact that the novel way of representation was as well adopted, as the one dictated by a habit of 5-10 working years, is a very positive indication for the success of the proposed approach. Regarding the ACCURACY of TD quantification, we can observe that it corresponds to the feature with most participants votes higher or equal to 8 out of 10 (sum of green, orange, and light blue bars)—meaning that only two low evaluations (dark blue, red, or yellow bars) have been assigned to this feature. Regarding the addition of end-points and new projects the results were similar and rather balanced. This is a bit unexpected finding, in the sense that these features were considered as trivial from the researchers' team.

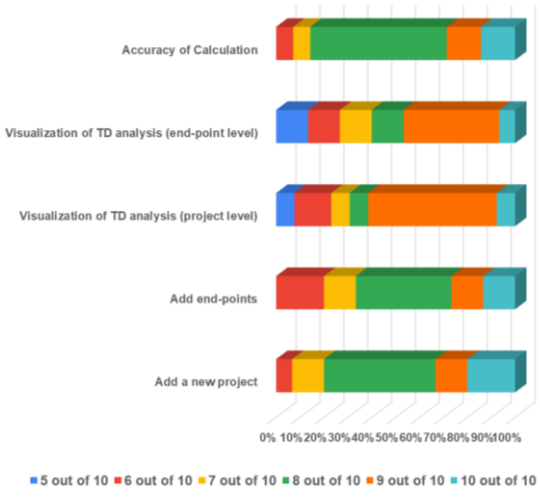

Fig. 6. Functionality Analysis.

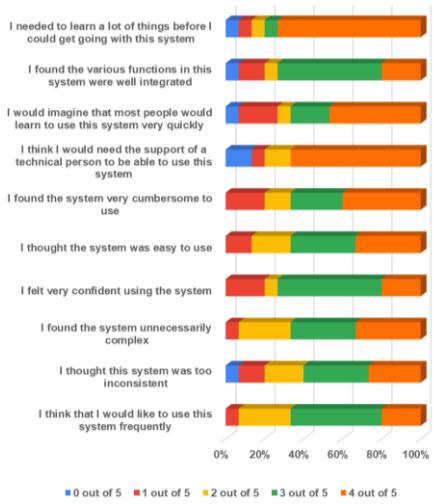

Fig. 7. SUS Analysis per Question

To further interpret the aforementioned results, next, we present the main outcomes of the qualitative analysis. First, regarding the addition of a new project, the participants believed that: (a) it was not really obvious how to fill in the needed information in the GUI; and (b) how to add a private repository. On the other hand, the addition of a new end-point and the execution of the analysis was characterized as extremely user-friendly. Nevertheless, a suggestion to select the end-points from a drop-down menu seemed to be favorable among the participants. However, as a tenta-

tive improvement, the participants mentioned that an Abstract Syntax Tree-like structure for choosing the method instead of copying and pasting text (i.e., the signature of the method) would be preferable. As an overall comment, the participants mentioned that: (a) a better space organization would make the visualization of the results easier to read. For instance, a pagination library could have been used in reporting the results per service or the issues that service suffers from; and (b) the use of different colors in the GUI would lead to more distinguishable items. For example, a different color per issue or service would be very helpful "*to catch the eye*". In terms of missing functionalities, the participants explained that enabling the customization of the rule-set of SonarQube through the Theia Extension would be very helpful, since many times organizations create custom ruleset and replace the default configuration of SonarQube. With respect to desired functionalities, the participants noted that future versions of the extension, could: (a) provide various sorting / filtering options (e.g., by criticality); (b) enable the navigation to the location of the code smell through GUI interaction (e.g., when clicking on the issue or the class, the corresponding part of the code to open in the Theia Editor); (c) provide help on how TD Principal is calculated and configure the constants (e.g., remediation times, default costs); (d) integrate Continuous Integration features to enable the quality gates and the continuous and automated monitoring of TD (e.g., link with Jenkins or build tools); and (e) export reports and import past analysis from previous versions or commits provide more detailed explanations on the issues.

6 Threats to Validity and Conclusions

Construct validity reflects to what extent the phenomenon under study really represents what is investigated according to the research questions [21]. To mitigate construct validity threats, we established a research protocol to guide the case study, which was thoroughly reviewed by two experienced researchers in the domain of empirical studies. Additionally, during the data collection process we aimed at data triangulation to avoid a wrong interpretation of a single data source. Another threat is the fact that the tool and the approach have been evaluated separately, and without a long-term usage of the tool before the study; this has introduced both negative or positive bias. On the one hand (negative bias), the evaluation of the stakeholders was probably stricter, since the users were completely inexperienced with the tool, they have probably faced more usability issues, compared to an evaluation that would have been performed after some training or self-training period. Therefore, we believe that the presented results, correspond to the worst-case scenario of usage and evaluation. On the contrary (positive bias), the evaluation might have been positively biased by using only a demo session, in which unclear parts were explained by the researchers, making them easier to understand by the practitioners. In terms of ***external validity*** (i.e., the generalizability of the findings derived from the sample [21]), it is difficult to claim that the same results would be derived in other companies. However, emphasizing on analytical generalization we can report on mitigation actions, which allow us to argue that the findings are representative for other cases with common characteristics

(especially for RQ_2 and RQ_3). Specifically, the participants of the study were professional software engineers with various years of experience in software development. Regarding RQ_1, however, the results might be difficult to generalize outside the five involved companies. Finally, the ***reliability*** of an empirical study concerns the trustworthiness of the collected data and the analysis performed, to ensure that same results can be reproduced [21]. We support the reliability of our study by creating a rigor case study protocol and interview guides, which were tested through pilots. To minimize potential reliability threats during the data collection process, we preferred to ask open-ended questions and we requested motivation for the provided answers. To assure the correct and unbiased data analysis, three researchers collaborated during the whole analysis phase. Finally, we have internally archived all collected data (both raw and coded), due to a non-disclosure agreement with our industrial partners. On the other hand, interview guides are presented in Section IV.

Conclusions: Service-based software systems have been widely adopted because of their inherent benefits including, but not limited to, reliability, scalability, platform independence, agility and easy maintenance. As in any other software system, the code behind services needs to be maintained to adapt to new requirements and fix bugs. To assess the maintainability of services the Technical Debt metaphor can be used; however, it should be adapted to the particular features of Service-Oriented Architecture. In this paper, we have introduced an approach and an accompanying tool (in the form of an Eclipse Theia extension) to quantify TD in service-based applications, refining the analysis from the project or class level, to the individual service level. An industrial validation study with 15 engineers from 5 companies revealed the importance of assessing TD for service-based systems at the level of services. In particular, the results suggested that the proposed approach can be considered as more accurate compared to the 'traditional' project-level approaches. The main benefits are related to TD prioritization and monitoring, time savings, and is perceived as useful by the practitioners. The tool has been evaluated as highly usable.

Acknowledgment

Work reported in this paper has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871177 (project: SmartCLIDE).

References

1. W. Cunningham, "The WyCash Portfolio Management System," Proceedings on Object-oriented programming systems, languages, and applications, pp. 29- 0, 992.
2. N. Zazworka, R. O. Spínola, A. Vetro', F. Shull, and C. Seaman, "A case study on effectively identifying technical debt," in Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering, New York, NY, USA, Apr. 2013
3. T. Amanatidis, N. Mittas, A. Moschou, A. Chatzigeorgiou, A. Ampatzoglou, and L. Angelis, "Evaluating the agreement among technical debt measurement tools: building an empirical benchmark of technical debt liabilities," Empir. Softw. Eng., 25 (5), Sep. 2020.

4. P. Avgeriou, D. Taibi, A. Ampatzoglou, F. Arcelli Fontana, T. Besker, A. Chatzigeorgiou, V. Lenarduzzi, A. Martini, N. Moschou, I. Pigazzini, N. Saarimäki, D. Sas, S. Soares de Toledo, and A. Tsintzira, “An overview and comparison of technical debt measurement tools,” IEEE Software, 2021.
5. D. A. Tamburri, P. Kruchten, P. Lago and H. van Vliet, "What is social debt in software engineering?" 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), 2013, pp. 93-96.
6. O. Zimmermann, “Microservices tenets,” Comput. Sci. - Res. Dev., Jul. 2017.
7. I. Pigazzini, F. A. Fontana, V. Lenarduzzi, and D. Taibi, “Towards microservice smells detection,” in Proceedings of the 3rd International Conference on Technical Debt, Jun. 2020.
8. S. Soares de Toledo, A. Martini, A. Przybyszewska and D. I. K. Sjøberg, "Architectural Technical Debt in Microservices: A Case Study in a Large Company," 2019 IEEE/ACM International Conference on Technical Debt (TechDebt), 2019, pp. 78-87.
9. D. Taibi, V. Lenarduzzi, and C. Pahl, “Microservices Anti-patterns: A Taxonomy”, Springer International Publishing, 2020, pp. 111–128.
10. M. Hasan, E. Stroulia, D. Barbosa and M. Alalfi, "Analyzing natural-language artifacts of the software process," International Conference on Software Maintenance, 2010, pp. 1-5.
11. Alves, N.S., Mendes, T.S., de Mendonça, M.G., Spínola, R.O., Shull, F. and Seaman, C., 2016. Identification and management of technical debt: A systematic mapping study. Information and Software Technology, 70, pp.100-121.
12. Z. Li, P. Avgeriou, and P. Liang, “A systematic mapping study on technical debt and its management,” J. Syst. Softw., vol. 101, pp. 193–220, Mar. 2015.
13. J. Lefever, Y. Cai, H. Cervantes, R. Kazman, H. Fang, “On the lack of consensus among technical debt detection tools.” Proceedings of the International Conference on Software Engineering (SEIP); 2021:121-130.
14. D. Tsoukalas, N. Mittas, A. Chatzigeorgiou, D. Kehagias, A. Ampatzoglou, T. Amanatidis, L. Angelis “Machine Learning for Technical Debt Identification,” IEEE Trans. Softw. Eng., 2021.
15. Campbell, G.A. and Papapetrou, P.P., 2013. SonarQube in action. Manning Publications.
16. J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann, “Limiting technical debt with maintainability assurance: an industry survey on used techniques and differences with service- and microservice-based systems”, International Conference on Technical Debt, 2018
17. A. Ouni, M. Daagi, M. Kessentini, S. Bouktif and M. M. Gammoudi, "A Machine Learning-Based Approach to Detect Web Service Design Defects," Int. Conf. on Web Services (ICWS), 2017, pp. 532-539.
18. J. Král and M. Zemlicka, "Popular SOA Antipatterns," 2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009.
19. E. Alzaghoul and R. Bahsoon, "Evaluating Technical Debt in Cloud-Based Architectures Using Real Options," 2014 23rd Australian Software Engineering Conference, 2014.
20. N. Smith, D. Van Bruggen, and F. Tomassetti, “Javaparser: visited”. Leanpub, Oct. 2017
21. P. Runeson, M. Höst, R. Austen, and B. Regnell, Case Study Research in Software Engineering – Guidelines and Examples. John Wiley & Sons Inc., 2012.
22. J. Brooke, J. “System Usability Scale (SUS): A quick-and-dirty method of system evaluation user information”, Taylor & Francis, 1996.
23. C. B. Seaman, "Qualitative methods in empirical studies of software engineering," in IEEE Transactions on Software Engineering, vol. 25, no. 4, pp. 557-572, July-Aug. 1999.
24. S. Elo and H. Kyngäs, “The qualitative content analysis process”, Journal of Advanced Nursing, vol. 62, issue 1, pp.107-115, 2008
25. D. Spencer, “Card Sorting: Designing Usable Categories”, Rosenfeld Media, April 2009.