Integrating Traceability within the IDE to Prevent Requirements Documentation Debt

Sofia Charalampidou
University of Groningen
The Netherlands
s.charalampidou@rug.nl

Apostolos Ampatzoglou
University of Groningen
The Netherlands
a.ampatzoglou@rug.nl

Alexander Chatzigeorgiou
University of Macedonia
Thessaloniki, Greece
achat@uom.edu.gr

Nikolaos Tsiridis
Open Technology Services
Thessaloniki, Greece
ntsiridis@gmail.com

Abstract— **Documentation issues in software projects have been recently classified as a type of technical debt (TD), a concept that expresses how shortcuts during software development result in additional maintenance and evolution effort. The specific type of TD is termed documentation debt, and is among the most prevalent ones in practice and research. In this study we propose a tool-based approach for preventing documentation TD during requirements engineering, by: (a) integrating requirements specifications into the IDE, and (b) enabling the real-time creation of traces between requirements and code. To this end, we collaborated with a small/medium software company and conducted a qualitative case study to: (a) analyze the current process and identify existing TD types, (b) collect the requirements and implement a tool that aims at preventing the accumulation of documentation TD, and (c) investigate whether the tool successfully meets its goal. The results of the study suggest that the developers are motivated to use the developed tool, since they feel that they can develop, maintain and utilize requirements specifications and traces as part of their daily routine.**

Keywords— requirements, traceability, documentation debt

I. INTRODUCTION

Software projects' documentation inefficiencies have been recently classified as a type of technical debt (TD) [1], a concept that expresses how shortcuts during software development result in additional maintenance and evolution effort [2]. This type of TD is termed documentation debt and is among the most prevalent ones in research [1] and industry [3]. In particular, Alves et al. [1] suggest that documentation TD is among the top-3 studied types of technical debt in academia, whereas Ampatzoglou et al. [3] suggest that documentation TD is perceived as the top-4 most important types of TD in industry.

Documentation TD is a broad term that can be identified at almost all software development phases, in this study we focus on documentation TD on requirements specifications. We selected to focus on requirement engineering, since it is one of the most crucial phases in the software development lifecycle, in the sense that the 9 out of 11 most notable reasons for software project failures are related to requirements [4]. As a first step towards investigating documentation TD in the requirements phase, we need to define the possible liabilities that can be identified in such specifications. Li et al. [5] suggested that insufficient, incomplete or outdated documentation are the main sub-types of documentation debt. By focusing on requirements, these sub-types can be described as follows:

1. Insufficient or incomplete requirements refer to pieces of specifications (e.g., use cases, user stories, SRS) that are developed either at low quality or do not describe the system under development. Low quality specifications miss at least one (or more) of the following characteristics: readability, traceability, verifiability, consistency, etc. [4].
2. Outdated requirements refer to cases in which specifications have been developed at an appropriate level of quality (in the early releases of the system), but subsequently the specifications are not updated with new requirements, or changes in existing ones [4].

Such documentation inefficiencies can occur intentionally, for example, in cases of selecting not to apply a rigorous documentation process, or change management strategies; or unintentionally, for cases when documents are not sufficiently maintained, due to tight schedule. Both intentional and unintentional documentation TD hinder maintenance productivity (i.e., incurring TD interest), by adding overheads that stem from compromised source code understandability, e.g., difficulty in locating where each feature is implemented, etc.

In practice, ***one of the most prominent examples of documentation TD, related to requirements, is the lack of requirements-to-code traceability***. This can be caused by insufficient and incomplete requirements, which are not linked to the parts of the source-code where they are implemented. Furthermore, lack of traceability increases the effort of keeping requirements updated, since it is harder to identify which requirements can be affected when the source code is changed. Consequently, high effort is required to locate features or bugs within large codebases [6], a fact which impedes maintenance activities (e.g. feature addition or debugging). However, such activities are central aspects for TD management (e.g., modules with low change proneness are not prioritized for TD repayment), in the sense that TD interest occurs only whenever a change is applied to the system (for more information on the relation of traceability and maintainability see Section II).

To deal with the aforementioned issues, in this paper we provide tool-support aiming at the prevention of requirements' documentation TD[1]. To achieve this goal, we propose to connect requirements specification and code artifacts, by integrating the specifications into the programming IDE. For this study, we worked together with a small/medium enterprise that is active for 20 years in mobile and web application development. The company was interested to reduce the amount of documentation TD, by adopting the proposed approach and

[1] *We note that documentation TD should not be confused with requirements debt,* which is a TD type related specifically to requirements and contrasts the distance between optimal requirements and the requirements that are actually implemented in the end software [5].

tool support. To support the company in preventing the accumulation of documentation TD, we need to achieve 3 sub-goals: (***sg1***) understand how the use of the current requirements specification process incurs documentation TD; (***sg2***) build a custom-made tool that integrates requirements specifications into the programming IDE; (***sg3***) investigate how the use of the tool, would benefit the company in preventing the accumulation of documentation TD.

The rest of the paper is organized as follows: first we present related work. Second we present the current status of the requirements specification process of the company; third we present the case study protocol that was designed to achieve the stated goals; fourth we present the outcomes of the study organized by the three aforementioned goals (sg1-sg3), and finally we present some lessons learned in the form of implications to practitioners and researchers, as well as threats to validity.

II. Related Work

As direct related work for this study, one should consider studies that investigate the aspect of documentation technical debt, and thus could be considered as comparable to our work. However, after looking into two literature reviews on the domain of TD [1, 5], we came up with the conclusion that although there are studies referring to the existence of documentation technical debt as a TD type, there are no studies investigating specifically this type of TD. In their vast majority, these studies investigate the TD metaphor from a general perspective, and thus they combine the investigation of documentation TD together with other types of TD. Thus, there is no approach found in the literature that we can directly make a comparison with. As related work section we can provide only indirect related work, as follows: In Section II.A, we present a set of studies that propose alternative traceability management approaches, whereas in Section II.B approaches that investigate the effect of traceability on maintainability (i.e., a proxy of TD). We note that we will not present studies that focus on demonstrating or evaluating traceability approaches, even if their goal is to improve maintainability (e.g. [7], [8]), if they do not provide empirical evidence on the aforementioned relation.

A. *Approaches for Traceability Management*

There are various approaches/tools linking requirements to source code, which can be classified into two major categories: (a) standalone or web-based that handle source code separately from requirements (e.g. [9 – 13]), and (b) IDE-integrated that place requirements and traces inside the IDE used for development (e.g. [14 – 17]). We focus on the latter since our case concerns an IDE-integrated tool that links requirements to code in real time. Several existing Eclipse-based traceability tools, e.g. TraceMe [15] and TraceEclipse [14] perform after-the-fact traceability (although Trace-Eclipse provides also the possibility to manually create traces between documentation and source code in real-time, in different levels of granularity). Additionally, the Capra tool [17] is an eclipse based plug-in tool, which aims at supporting different tracing needs. For this reason the types of links to be supported, the types of artifacts that can be traced and the way the links should be stored are customizable. However, the proposed tool has not been empirically validated [17]. Finally, a tool proposed by Asuncion et al [9] allows the creation of end-to-end traces within the whole process of a company. However, the proposed tool is implemented as a standalone application. A comparison of our approach compared to state-of-the-art is presented in Table I. From the table we can observe that: (a) study [9] although targets real-time traceability management and is well-validated, it does not explore the IDE integration feature, (b) studies [14][15] although offer an IDE plugin, they only cover after-the-fact traceability issues, and (c) study [17] which is poster paper does not provide any validation at all.

TABLE I. Comparison to Related Work

Study	IDE integration	Real Time Traceability Management	Empirical Validation
[14], [15]	X		X
[17]	X	X	
[9]		X	X
Our study	X	X	X

B. *Effect of Traceability on Maintenance*

Mäder and Egyed [5], conducted a controlled experiment that aims at investigating the effect of requirements to code traces on maintenance tasks. Initially they examined if the subjects' performance (in terms of time and correctness) was improved during the software maintenance tasks, and then they investigated the differences due to a range of other criteria (i.e. the kind of tasks subjects solved, the different project domains, and the order in which a task was performed by a subject). The results indicated a beneficial effect of traceability on maintenance quality. Jaber et al. [18], performed also a controlled experiment investigating the benefits of having traces among different types of software artifacts in the maintenance phase. For the needs of the study the authors created a prototype tool (TraceLink), which creates a model of all stored traces among artifacts. The study has three major goals. First, to investigate whether traceability links affect the accuracy of maintenance tasks, second to identify potential benefit in terms of time reduction when performing a maintenance task, and third to examine whether developers' abilities do not affect task accuracy, difficulty, or speed. The results showed that task accuracy improves when traceability links exist. However, no significant results were drawn about the time improvement. Bianchi et al [19], conducted an exploratory case study to analyse the role of different types of traceability models in software maintenance. The study evaluates the degree of granularity that is more effective in modelling traceability. Two maintenance effectiveness aspects were taken into account: efficiency, and accuracy. The results of the study suggest that when a fine-grained trace is used (i.e., methods or attribute level), the effort required to satisfy a maintenance request is greater, but so is the accuracy of the modifications. Therefore, there is a trade-off between the efficiency and the accuracy of the maintenance task. Finally, Neumüller and Grünbacher [20] introduced traceability in a small company and developed a traceability environment for their needs. In their study they discuss the traceability approach they used and report on key lessons learned. The approach is fairly simple establishing trace links based on already existing conventions for developers. The tool focuses on trace utilization by providing attractive visualization and query mechanisms, without complex automations or interfering with the development practices. The lessons learned concern the way of introducing traceability into a company rather than the user experiences.

III. Current Requirements Specification Process

During this study we performed an exploratory case study, with our industrial partner, named Open Technologies Services (OTS). OTS is a small/medium scale company that is considered one of the key-players in Greece's mobile and web development market. For the needs of this study we selected eight participants (six developers, the quality assurance manager and the project manager) that were involved at different phases of the study. The experience of the participants ranged from one year (junior developers) to approximately a decade in the company (project manager). In this section, we present the current status of the requirements specification process of our industrial partner, organized based on: (a) the involved stakeholders, and (b) the current approach and the used tools.

Stakeholders: The stakeholders are both technical and non-technical, from different departments of the company. The main role dealing with specifying and managing requirements is developers, who are interested in tracing the requirements they receive (documented in the form of JIRA issues), into the implementation of the system (i.e. in the source code). Developers are interested in retaining the requirements as close to the source code as possible, so as to minimize the overheads of their management (e.g., not use tools other than their IDE, neither for documenting requirements, nor for maintaining traces). Other stakeholders in the development team include: the quality assurance manager, who is interested in knowing which requirements are implemented and which parts of the code should be tested; and the project manager, who is interested in communicating the progress of the development team to customers or higher management, and thus is concerned with the understandability of documentation by non-technical stakeholders (e.g., the customer support centre, which is responsible for reporting a large portion of issues).

Current Approach and Tools: Currently the company does not follow a strict requirements specification process (mostly due to tight development schedule) and there are no means of explicit traceability between software artifacts. According to the descriptions of the stakeholders, the requirements of a project are basically documented by the developers as JIRA issues, in the form of user stories (US). Once a user story is implemented, the corresponding code is committed to Git. The linking between the source code artifacts that implement a specific user story is performed when committing the code to Git, by using the JIRA issue ID in the commit comments. According to the project manager, flexibility and efficiency of this process are its positive characteristics, since developers do not like to spend time in documenting unnecessary information. Since in the current process the developers specify the requirements themselves, they have control over their tasks, and usually they just write enough details to start writing code. However this approach has as downside the unintentional creation of TD, since due to time limitations developers do not document requirements properly (e.g., user stories are not updated for minor changes, textual requirements are not detailed enough to fully specify the requested functionality, etc.), and this results to bad documentation, which affects the understandability of others in the team, or even the understandability of the code owner in the future.

The created traces are utilized along maintenance activities (e.g., bug fixes or new feature requests). However, the current process is based in many cases on oral communication. When a specific code artifact needs to be maintained, the person that wrote the code is the first one to be asked. If this is not possible, developers look for the class that needs to be updated by applying a full text search, and navigate based on the comments existing in the source code (when applicable). To get an insight on a specific part of the code, the commits that changed the respective code are tracked and manually reviewed. Upon successful completion of the maintenance task, developers are expected to signify other requirements that need to be tested, due to potential overlaps in the source code artifacts implementing these requirements. The link to requirements is done by creating a new JIRA issue, which references the issue ID that initiated the need of the check. However, the current process offers no guidance on identifying such overlapping requirements.

IV. Case Study Design

The goal of this case study is: (a) to analyse the currently used requirements specification process regarding how it incurs documentation TD (i.e. insufficient, incomplete or outdated requirements specifications); (b) to propose a tool for preventing the accumulation of such documentation TD; (c) to investigate to what extent the proposed tool achieves its goal. The case study was designed and is reported following the guidelines provided by Runeson et al. [21]. Based on the sub-goals defined in the introduction, we have derived three research questions (RQ):

RQ$_1$: *What aspects of the current requirements specification process are prone to incur documentation TD?*

To answer this research question, different steps of the current requirements specification process are discussed with the involved stakeholders to understand how and why they potentially incur documentation TD. Such steps will be linked to specific types of documentation TD (i.e. insufficient / incomplete / outdated requirements).

RQ$_2$: *What type of tool would be helpful in preventing the accumulation of documentation TD?*

In this research question we explore the main requirements (of our industrial partner) on building a requirements specification tool with the aim of preventing the accumulation of TD. We aim at collecting a set of functional and non-functional requirements, describing the ideal requirements specification system according to the company stakeholders. As a starting point for discussing the requirements for the developed tool, the TD-prone aspects of the process identified in RQ$_1$ will be used. As part of the answer to this research question, we will present the developed tool, and discuss how we evaluated its functional and non-functional behavior. The ability of the tool to aid in preventing documentation TD accumulation is investigated in RQ$_3$.

RQ$_3$: *How does the developed requirements specification tool aid in the prevention of accumulating documentation TD?*

In this research question we explore whether the developed tool can help stakeholders *prevent the accumulation of* documentation TD during the requirements specification, and the extent to which maintenance effort is reduced. Apart from the expert opinion of the stakeholders, in this research question we will focus on the rationale of the opinions, explaining how the tool can actually aid in technical debt prevention.

For the needs of our study we collected qualitative data through different collection methods (see Table II) which are further discussed below. For all research questions, triangulation of data sources or collection methods (two per research question) has been achieved, to mitigate data collection bias.

TABLE II. DATA COLLECTION METHODS PER RESEACH QUESTION

Collection method	RQ1	RQ2	RQ3
Pre-tool interviews	X		
Observation session	X		X
Focus group (Requirements elicitation)		X	
Focus group (Discussion on prototypes)		X	
Post-tool interviews			X

In the first part of our study (RQ_1) we conducted three individual interviews with a developer, the quality assurance manager and the project manager. In order to enhance the reliability aspect, all conducted interviews followed interview guides, which had been tested through a pilot case study (the study guides, and the developed tool are all available online[2]). These interviews aimed at giving us some insight on how the current state-of-practice results in the creation of different types of documentation TD. The identified documentation TD types acted as pointers for the development of the tool. As additional means of collecting data, we performed an observation session, aiming at collecting information about steps of the process or actions that might have been omitted during the interviews. The outcome of RQ_1 indicates the parts of requirements specification that result in documentation TD; and therefore are in need of tool support.

The extraction of the requirements for the developed tool (RQ_2) was performed through two focus group sessions (which were structured according to the guidelines of Kontio et al. [22]). During the **planning** of the focus groups we defined the goals. The first focus group aimed at collecting a set of initial requirements, focusing mostly on functional requirements. The second one aimed at finalizing the requirements elicitation process and collecting non-functional requirements. The two focus groups had a similar **design**. They both lasted for 45' and the participants were company stakeholders, with different fields of expertize and backgrounds. While **conducting** the first focus group a list of candidate requirements from literature and other existing systems were used for driving the discussion. During the second focus group a set of mock-up prototypes were used for the same purpose. The two focus groups were performed with a time difference of one month, so that to process the results of the first focus group, before designing the mock-up prototypes. Subsequently we developed the requested tool. To evaluate the conformance to the needs of the company we examined, through interviews and observation sessions, how five developers experience the use of the tool, in terms of both the functional and non-functional requirements.
Finally, ***after one month that the tool development was stabilized***, we evaluated the effect of using the tool for preventing the accumulation of documentation TD (RQ_3). To do so, we conducted five post-tool usage interviews with developers who had used the tool, during which we discussed how the use of the tool affected the maintenance effort required to resolve future tickets, and the extent to which it affects the accumulation of technical debt, related to the requirement documentation TD types (as identified in RQ_1). For similar reasons to RQ_1, we performed post-tool observation sessions as well. The outcome of this process was the rationale of experts, explaining why the tool is beneficial in terms of TD prevention.

V. RESULTS

In this section we present the results obtained by analyzing the data from the interviews, focus groups and observation sessions. The collected dataset for all research questions was analyzed using the constant comparison technique, which is a systematic way for analyzing qualitative data [23]. Specifically, we transcribed all interview and focus group audio files, and we compiled them into a data set, including also the notes kept during the observation sessions. Then we coded the data set, i.e. categorized all pieces of text that were relevant to a particular theme of interest [23], and we grouped together similar codes, creating higher-level categories (see Table III). The categories were created during the analysis process by both the first and the second author, and were discussed and grouped together through an iterative process in daily meetings. In the upcoming sections, in parenthesis, we use the codes identified in the textual analysis process. We note that in Table III we present the answers provided by the respective participants (i.e. for the first part of the study, the developer (D), the project manager (M) and the quality assurance manager (Q) and for the second part of the study the five developers (participants 1-5). Additionally, we note that cells with grey color indicate that the corresponding participants were not involved in those code categories.

A. Documentation TD in Current Process

In this section we discuss the inefficiencies of the current requirements specification process that lead to the accumulation of documentation TD. An illustrative case describing this process is summarized in Figure 1.
Incomplete and insufficient requirements. The process is considered to create *obstacles in the communication among departments* (e.g., support, development, and testing) (c1). From the perspective of developers, the main reason is that when non-technical stakeholders use JIRA to add new issues, the issue specifications are not formulated as proper user stories (c2). Similarly, *the freedom that developers have while creating documentation often results to poor documentation (due to time limitations),* which hinders understandability (c2). As a result, *oral communication is required* to bridge these gaps and often ends up driving the process (c3).

Additionally, according to all participants, the current process does not support sufficient traceability among software artifacts (c4). Specifically, there is a lack of a structured way to link requirements and test cases, or JIRA issues to user stories. As a result, *maintenance activities (bug fix or addition of features) become more difficult, since it is hard to locate concrete artifacts*. Thus the task of navigating into the source code, is perceived as even more complex for the new developers in the company (c5).

[2] http://www.cs.rug.nl/search/Resources/Resources

TABLE III. MAPPING BETWEEN CATEGORIES AND PARTICIPANTS

Category	Participants							
	D	M	Q	1	2	3	4	5
(**c1**) Communication difficulties among stakeholders	+	+						
(**c2**) Insufficient/incomplete US documentation affects understandability	+	+	+					
(**c3**) Oral based communication with documentation of basic decisions	+	+	+					
(**c4**) There are traces among limited artifacts	+		+					
(**c5**) Traces support understandability of developers	+	+	+					
(**c6**) Need of system overviews (FR, test cases etc.)	+	+	+					
(**c7**) Isolated documentation hinders maintainability	+	+	+					
(**c8**) Use of an IDE based traceability tool	+	+	+					
(**c9**) Non-technical stakeholders can suffice with an output document	+	+	+					
(**c10**) Capability to link US to code and vice versa	+	+			+			
(**c11**) Flexibility and efficiency are major benefits	+	+	+					
(**c12**) Links' granularity should be at least at class level		+						
(**c13**) Requirements management via tree structure	+	+		+	+	+	+	+
(**c14**) Search functionality for requirements or code	+	+	+	+	+			
(**c15**) Users have different interaction preferences	+	+		+	+		+	+
(**c16**) Integration with Git	+	+			+		+	+
(**c17**) Low consumption of resources	+			+	+	+		
(**c18**) Low response time	+	+		+	+	+	+	
(**c19**) Scalability is important	+			+	+			
(**c20**) There is no strict traceability process	+	+	+					

Furthermore, all participants agree that one of the biggest challenges of the current process is that they cannot easily *trace which requirements are already implemented* and extract a feature set (c6). This causes troubles in several ways: (a) there is inefficiency in project progress tracking; (b) communication with customers on bug-fixing progress is hindered; and (c) the testers are not aware of the requirements that need to be tested, since they are not expected to track commit records for identifying changes. The manager reported that *it would be a big benefit to "be able to see the stories well organized, without looking into other external systems*".

Outdated requirements. When the *documentation of a project is isolated* from the project itself (i.e., use of different tools), developers do not perceive updating it as part of their responsibility, but as an extra burden for their main tasks (c7). An additional problem is that in the current process, when developers deal with a JIRA issue, they do not have access to the intended functionality as a whole. As a result, *they often miss the context of the user story* (c6).

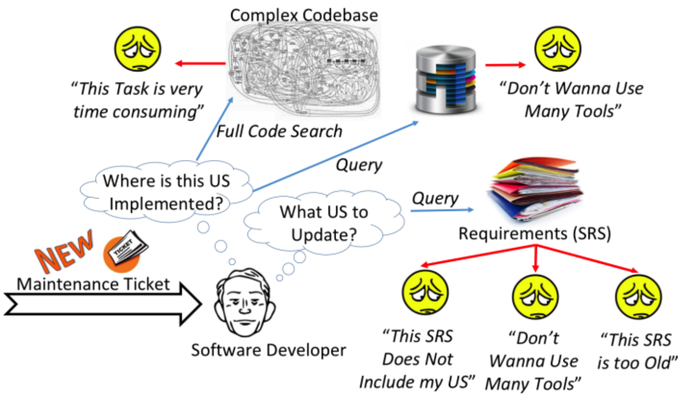
Complex Codebase
"This Task is very time consuming"
"Don't Wanna Use Many Tools"
Full Code Search
Query
Where is this US Implemented?
What US to Update?
Query
Requirements (SRS)
NEW
Maintenance Ticket
Software Developer
"This SRS Does Not Include my US"
"Don't Wanna Use Many Tools"
"This SRS is too Old"

Figure 1: Illustrative presentation of the process

B. *Proposed Tool for TD Prevention*

How the tool could prevent specification TD: All participants agree that external tools would not be easily integrated in their process. Instead, the company needs a tool to *document and trace requirements, integrated into the used IDE* (c8). Such an integration aids in the prevention of accumulating instances of the documentation TD types discussed in RQ$_1$ (i.e., incomplete, insufficient or outdated requirements).

First, it is important that the user can choose how much documentation is really necessary to understand a user story; therefore, the tool should provide flexibility in the level of details for requirements specifications. This functionality helps in avoiding over-engineering requirements, but at the same time assures the existence of a minimal level of details needed to avoid having ***insufficiently specified requirements***. Furthermore, all participants underlined that the tool should support communication among stakeholders alleviating existing problems (c1). For this reason the tool should enable the sharing of requirements specifications among developers and non-technical stakeholders (e.g., the product owner and the sales department), by producing reports that are friendly to non-technical stakeholders. This is expected to contribute towards limiting the number of ***incomplete requirement specifications***, since they will be verified by several stakeholders. Such a verification, assures the completeness of requirements, from potentially different perspectives. Since non-technical stakeholders may not be willing to use the IDE-based environment of the tool, the tool should provide as output overview files (c9). Concerning the creation of links, the tool should provide *functionality for connecting requirements to source code* (i.e. following a link should open the right method/class where a requirement has been implemented) and vice versa (c10). This requirement is expected to alleviate the accumulation of TD due to lack-of-requirements-to-code traceability.

Second, all participants consider that *a tool that should be easy to use, so as to be easily adopted into the workflow of the developers* (c11)—promoting the ***update of requirements specifications*** (i.e., preventing the accumulation of additional outdated requirements TD). In terms of source code, the linking should be done at least at class level, while according to a participant creating traces at method level could also potentially be useful (c12). Additionally, in the tool it should be possible to link many requirements to one class, and vice versa (c10).

In addition to TD prevention the tool should obey to generic guidelines of requirements specification processes. In particular, the imposed requirements specification should capture the goal and the reasoning for each user story, and keep information about the system/subsystem the story belongs to, the responsible actor and the respective test scenarios. The user stories should be visualized in a tree structure, and should be organized in smaller categories, based on the subsystems or actors they refer to (c13). What is more, *most participants name the search functionality as a basic one* (c14). Specifically the user should be able to search for a concrete requirement by providing a term, or by choosing an actor or a subsystem and the system should identify the classes in which this requirement is implemented, and vice versa. This would reduce the time required for locating the part of the system where changes will be made.

To guarantee that stakeholders will use the tool in a regular basis, a set of four non-functional requirements (namely: ***usability***, ***interoperability***, ***performance***, and ***scalability)*** have been identified by the participants, acting as key drivers for the development of an ideal requirements specification management tool. In particular ***usability*** should be the major point of attention: most participants stated that one of the biggest challenges is to have a usable tool for all developers, due to the different preferences and habits that each one has (e.g., use the tool either by mouse or code annotations) (c15). ***Interoperability*** is another aspect discussed at several points by most participants, namely how the tool integrates with other tools and environments used by the company—e.g. Git and Eclipse (c16). In terms of ***performance*** (i.e. the response time and the resources used by the system when initializing the traces between user stories and source code) it was stated that "*loading a project should not consume much resources*" (c17) and that "*the tool would prevent us from using it only if it had serious issues, like delays*" (c18). Finally, the participants mentioned the importance of ***scalability*** (c19), since the project sizes vary, and grow over time, and thus the tool should be able to handle an increased number of user stories or classes.

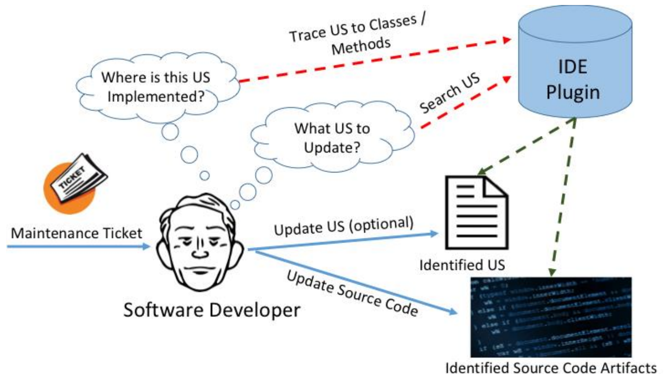

Figure 2: Illustrative example of the core tool functionality

<u>*Developed Tool*</u>: Based on the aforementioned collected requirements we created an Eclipse plug-in that provides developers the desired functionalities for managing requirements specifications. This functionality concerns the specification of new user stories inside the IDE, and the easy identification of existing user stories, which can be traced to the code, where they are implemented. Thus, when a new maintenance ticket arrives, the developer can effortless reach the respective user story and update the source code that implements it.

The interface of the core functionalities of the developed Eclipse plug-in is presented in Figure 3, in order to provide the reader with an idea about the look and feel of the tool. The tool was validated through interviews and observations and the results were positive, since all users were able to use it for the assigned tasks (functional requirements) accurately and timely. This finding can be considered as expected since the tool has been developed based on the needs and processes of our industrial partner. However, the evaluation process suggested that the quality characteristic that still needs improvements is usability. This is not surprising, as usability is the most subjective of these qualities, and thus hard to satisfy for all stakeholders.

C. Effectiveness of the Tool

In this section we discuss the obtained benefits by the use of the tool, in terms of preventing documentation TD. The benefits with respect to prevention of accumulating TD are organized per documentation TD type.

The developed tool provides a simple and efficient way to integrate requirements specifications inside the IDE environment. According to the project manager connecting requirements documentation with the source code is "the biggest benefit" of the new process (c8). In this set up developers tend to perceive specifications maintenance as part of their job and not as something external. This is a great achievement in terms of preventing ***outdated documentation***. The IDE-integration imposes the use of the tool in the daily routine of the developers in a very natural way, and thus developers are willing to use it. In terms of how the tool can affect the future maintenance of the specifications, the project manager stated that "*it is much easier for developers to maintain the stories, because it's in the IDE, so developers do not have to maintain data in different tools. It's not a large effort. It is very interesting to see how using an independent tool and using a tool that is close to the code affect the use from the developers"* (c7). At the same time, the tool maintains the advantage of flexibility (c11), i.e. developers do not need to spend much time on documentation (avoiding over-engineering), but they can document just-enough and just-in-time to clearly specify the given requirement just before starting its implementation. The threat of accumulating TD related to ***requirements insufficiently or incompletely specified*** is mitigated by sharing the specifications with technical- or non-technical stakeholders, who can potentially ask for clarifications or additions. Finally, the introduction of the tool in the daily routine of developers provided them *a systematic way to capture traces between requirements and source code, an aspect that was completely absent* before the development of the tool (c20). Thus, intermediate solutions like commenting on commit records, or commenting in the source code have been successfully substituted with a more efficient and comprehensive solution, preventing the accumulation of TD due to lack of requirements-to-code traceability.

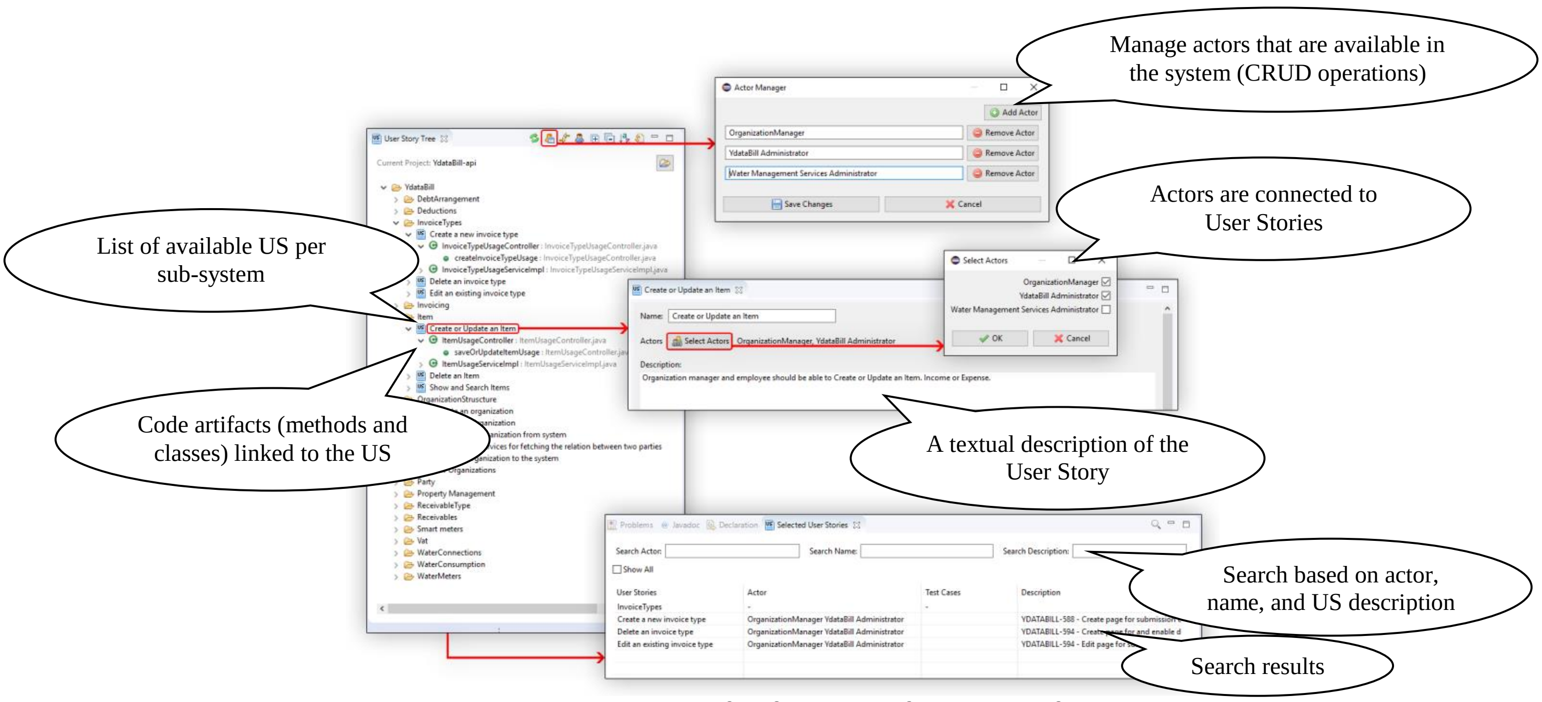
Manage actors that are available in the system (CRUD operations)
Actors are connected to User Stories
List of available US per sub-system
Code artifacts (methods and classes) linked to the US
A textual description of the User Story
Search based on actor, name, and US description
Search results

Figure 3: Screenshots showing the interface for the basic functionality of the developed tool

VI. Lessons Learned

The results of our study confirmed the existence of technical debt at the requirements specification level, in the sense that all types of documentation TD have been identified by the stakeholders. This finding can be considered as expected, since in literature, industrial studies suggest that documentation TD is among the most common forms of TD [2]. In addition to that, we explored which aspects of requirements specification process are more prone to produce TD. The results suggested that TD is accumulated on artifacts for which communication among many stakeholders is required, or on those specified in different documents. Both aforementioned findings can be considered intuitive since: (a) communication between stakeholders can cause understandability issues and inconsistencies; (b) documentation in multiple documents is very time-consuming and may lead to omissions in specification.

As a possible solution to these issues the stakeholders have promoted the integration of requirements specification in the IDE, rendering developers as responsible for their update and maintenance. The provided solution has been well-accepted by developers, who considered it as a viable way to prevent the accumulation of further TD. The eager adoption of the tool by developers is intuitive; the documentation maintenance can be done inside the IDE and thus it feels as part of their daily routines, considering the extra burden as negligible. Additionally, the reduction of TD is expected to reduce the required maintenance effort, as well. In particular, the participants of the study expect that this change will make the maintenance of the software easier (and less costly) since the time required for identifying the affected parts of the code will be reduced. This benefit stems from the exploitation of requirements to code traces. The introduction of traceability management into the process of the company, through the creation of links between the requirements specifications and the place in the source code where they are implemented has a great effect in terms of understandability of the code. Thus, any maintenance action (i.e., bug-fixing or feature addition) would be less effort intensive, since developers would not need to invest time on identifying the parts of the code where the user story resides in. The benefits of using the tool are expected to be stronger for novices.

Based on the aforementioned findings, we identified some ***implications for practitioners and researchers***. First, a requirements specification tool, which is integrated in the IDE and also promotes requirements-to-code traceability, can support the creation and maintenance of well-defined user stories, and can increase the understandability of the project especially to new developers. Thus, we suggest practitioners to use a tool for managing requirements specifications and traces according to their specific needs. In particular, given the positive evaluation of the proposed plug-in, we suggest practitioners that already use Eclipse and document their requirements as user stories to adopt the proposed plug-in[1]. From a research point of view the current study can be followed up with a longitudinal, quantitative case study to collect empirical evidence on the benefits of the long-term use of the developed tool in the company. Additionally, a replication in the context of a different company would be useful, so as to check the generalizability of our findings in different processes.

VII. Threats To Validity

Potential threats to validity of the conducted study concern construct, external, and internal validity and reliability threats. Since the goal of the study is not to establish a causal relationship between the use of the tool and the prevention of TD, but only to provide an initial exploration, we believe that internal validity is not a main concern for this study's validity. 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 and method triangulation to avoid a wrong interpretation of a single data source. One could argue that the decision to involve two different sets of participants, one for extracting requirements and one for evaluating the developed tool could be a threat in terms of construct validity. However, we believe that participation of experienced stakeholders in different roles during the elicitation process provides a more global view of the needs of the company, while the evaluation by various developers using the tool in a daily basis helps to mitigate bias. Another threat is the fact that the use of the tool was performed in the company for a small time period (one month) before the post-use case study. However, we believe that this timeframe was adequate for the software engineers to perform an initial validation.

In terms of external validity (i.e., threats concerning 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. Specifically, the participants of the study were professional software engineers with varying years of experience in software development. Additionally, OTS is an established SME in the domain of mobile and web development; therefore we can argue that the studied units are representative for similar companies in the same domain.

The reliability of the case 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 a 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, two researchers collaborated during the whole analysis phase. Finally, we have archived internally (due to a non-disclosure agreement with our industrial partner) all collected data both raw and coded, so that the analysis can be verified and traced: interview and focus group guidelines are openly available.

VIII. CONCLUSIONS

One of the reasons resulting to requirements specification TD is their inconsistent management from different stakeholders, which usually involves different documents. To alleviate this, we developed a plugin that integrates requirements specification in the Eclipse IDE enabling their tracing to source code. The main benefit of using the developed plugin is the motivation of developers to create, maintain and utilize requirements specifications and traces as part of their daily routine.

ACKNOWLEDGMENT

We would like to thank all study participants, and Theodoros Kyriazidis and Christos Kotsis for developing the plug-in.

REFERENCES

[1] Alves, N.S.R., Mendes, T.S., de Mendonça, M.G., Spínola, R.O., Shull, F., and Seaman, C., "Identification and management of technical debt", *Information on Software Technology*, 70, pp. 100-121, 2016.

[2] W. Cunningham. 1992. The WyCash Portfolio Management System. In Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications. NY, USA, 29–30.

[3] Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P., Abrahamsson, P., Martini, A., Zdun, U., and Systa, K., "The Perception of Technical Debt in the Embedded Systems Domain: An Industrial Case Study", *8th International Workshop on Managing Technical Debt (MTD' 2016)*, IEEE, 2016.

[4] Van Vliet, H., Software Engineering: Principles and Practice, 3rd edition, John Wiley & Sons, 2008.

[5] Li, Z., Avgeriou, P., and Liang, P., "A systematic mapping study on technical debt and its management", *Journal of Systems and Software*, 101, pp. 193-220, 2015.

[6] Mäder, P., and Egyed, A., "Do developers benefit from requirements traceability when evolving and maintaining a software system?", *Empirical Software Engineering.* 20, (2), pp. 413-441, 2015.

[7] G. Bavota, L. Colangelo, A. De Lucia, S. Fusco, R. Oliveto and A. Panichella. Enhancing Traceability Management in Eclipse via Information Retrieval and User Feedback Analysis. 7th Italian Workshop on Eclipse Technologies, Naples, Italy, 2012. LCNS Press.

[8] M. Shahid and S. Ibrahim, "Change impact analysis with a software traceability approach to support software maintenance," 2016 13th International Bhurban Conference on Applied Sciences and Technology (IBCAST), Islamabad, 2016, pp. 391-396.

[9] H. U. Asuncion, F. Francois, and R. N. Taylor, "An end-to-end industrial software traceability tool," 6th ESEC-FSE, 2007, pp. 115–124.

[10] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Advancing candidate link generation for requirements tracing: The study of methods." IEEE Transactions on Software Engineering, vol. 32, no. 1, pp. 4–19, 2006.

[11] M. Lormans and A. van Deursen, "Can LSI help reconstructing requirements traceability in design and test?" 10th CSMR, 2006.

[12] A. De Lucia, R. Oliveto, and G. Tortora, "ADAMS Re-Trace: Traceability link recovery via latent semantic indexing," in Proc. of 30th ICSE. , 2008, pp. 839–842.

[13] J. Lin, C. C. Lin, J. Cleland-Huang, R. Settimi, J. Amaya, G. Bedford, B. Berenbach, O. B. Khadra, C. Duan, and X. Zou, "Poirot: A distributed tool supporting enterprise-wide automated traceability," in Proc. of 14th IEEE RE. , 2006, pp. 356–357.

[14] S. Klock, M. Gethers, B. Dit, and D. Poshyvanyk, "Traceclipse:an eclipse plug-in for traceability link recovery and management," in Proc. of the 6th TEFSE, 2011, pp. 24–30.

[15] G. Bavota, L. Colangelo, A. De Lucia, S. Fusco, R. Oliveto and A. Panichella. TraceME: Traceability Management in Eclipse. 28th International Conference on Software Maintenance (ICSM 2012) Tool Demo, Riva del Garda, Italy, 2012. IEEE. pp. 642-645.

[16] A. Marcus, X. Xie, and D. Poshyvanyk, "When and how to visualize traceability links?". 3rd international workshop on Traceability in emerging forms of software engineering (TEFSE '05). ACM, 2005.

[17] S Maro, JP Steghöfer, "Capra: A Configurable and Extendable Traceability Management Tool". In IEEE 24th International Requirements Engineering Conference (RE), 2016, pp. 407-408.

[18] K. Jaber, B. Sharif and C. Liu, "A Study on the Effect of Traceability Links in Software Maintenance," IEEE Access, pp. 726-741, 2013.

[19] A. Bianchi, A. R. Fasolino and G. Visaggio, "An exploratory case study of the maintenance effectiveness of traceability models," Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on, Limerick, 2000, pp. 149-158.

[20] C. Neumuller and P. Grunbacher, "Automating Software Traceability in Very Small Companies: A Case Study and Lessons Learne," 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), Tokyo, 2006, pp. 145-156.

[21] Runeson P., Höst M., Rainer A., Regnell B., "Case Study Research in Software Engineering: Guidelines and Examples", Wiley & Sons, 2012.

[22] Kontio, J., Bragge, J., Lehtola, L., "The focus group method as an empirical tool in software engineering", *In: Guide to Advanced Empirical Software Engineering*, Springer-Verlag, pp. 93-116, 2007.

[23] Seaman, C.B., "Qualitative Methods in Empirical Studies of Software Engineering", *IEEE Transactions on Software Engineering*, 25(4), pp. 557–572, IEEE, 1999.