Towards a Holistic Definition of Requirements Debt

Valentina Lenarduzzi
Tampere University
Tampere, Finland
valentina.lenarduzzi@tuni.fi

Davide Fucci
University of Hamburg
Hamburg, Germany
fucci@informatik.uni-hamburg.de

Abstract—When not appropriately managed, technical debt is considered to have negative effects to the long term success of software projects. However, how the debt metaphor applies to requirements engineering in general, and to requirements engineering activities in particular, is not well understood. Grounded in the existing literature, we present a holistic definition of requirements debt which includes debt incurred during the identification, formalization, and implementation of requirements. We outline future assessment to validate and further refine our proposed definition. This conceptualization is a first step towards a requirements debt monitoring framework to support stakeholders decisions, such as when to incur and eventually pay back requirements debt, and at what costs.

Index Terms—Technical Debt, Requirement Engineering, Requirement Elicitation

I. Introduction

Cunningham defines Technical Debt (TD) as *"The debt incurred through the speeding up of software project development which results in a number of deficiencies ending up in high maintenance overheads"* [1]. TD implies suboptimal design or implementation solutions giving a short-term benefit while making changes more costly or even impossible in the medium and long term. Unpredictable business and environmental forces, internal or external to a company, can result in TD which needs to be managed [2], [3].

The TD metaphor was initially concerned with software implementation (i.e., at code level), but it has been gradually extended to software architecture, design, documentation, testing, and requirements [4]. Li et al. [5] conducted a systematic mapping study on understanding and managing TD drawing an overview on the current state of research. They proposed a classification of 10 types of TD: Requirements, Architectural, Design, Code, Test, Build, Documentation, Infrastructure, and Versioning.

Requirements Debt (ReD) *"refers to the distance between the optimal requirements specification and the actual system implementation, under domain assumptions and constraints"* [6]. Despite the importance of requirements engineering activities during software development process [7], [8] and the definition of the minimum viable product (MVP) [9], there is still no consensus in research whether ReD should be considered as a type of technical debt or not [8]. Different processes could led to different requirement decomposition and accumulate different debt [10]. We believe the reason is the lack of formalization of ReD in the literature [5], [11].

In this paper, we challenge the current definition of ReD [6]—which focuses on downstream activities in the software development lifecycle, such as software development and evolution—and extend it to include upstream activities involving the elicitation of requirements (particularly in user-centered requirements engineering [12]) and their translation into specifications.

Our definition of ReD is the first step towards creating a framework that stakeholders can use to make decisions regarding when to incur debt, at what costs, when to pay it back, and how to monitor it. Our vision of ReD will be empirically evaluated in a series of studies with industry partners and individual stakeholders.

In this paper, we aim at providing a holistic definition of ReD which takes into account the relevant requirements engineering activities currently investigated in the literature. Moreover, we outline the future assessments to conceptualize and define the decision framework.

II. Definitions

In this section, we report the basic definitions of concepts associated to Technical Debt (TD) used to define Requirement Debt (ReD).

Principal. In finance, it refers to the original amount of money borrowed. From a software development perspective, the term is used to describe the cost of remediating planned software system violations. Ampatzoglou et al. [13] defines principal within a TD context as: "The effort that is required to address the difference between the current and the optimal level of design-time quality, in an immature software artifact or the complete software system."

Interest. It is the negative effects of the extra effort that has to be paid due to the accumulated amount of debt in the system, such as executing manual processes that could potentially be automated, excessive effort spent on modifying unnecessarily complex code, performance problems due to lower resource usage by inefficient code, and similar costs [14]. Ampatzoglou et al. [13] defines interest as: "The additional effort that is needed to be spent on maintaining the software, because of its decayed design-time quality."

Quantify and paying back TD. The principal should be paid if it is less than the total interest [15]. Refactoring decision depends on the ratio between Principal and Interest. If the

value is greater than one, it is not convenient to pay the principal now with respect to the interest that will be paid in the future [16].

III. ReD—Requirements Debt

In this section, we provide a definition of Requirements Debt (ReD), propose its conceptualization, and strategies for detecting, quantifying, and paying it back.
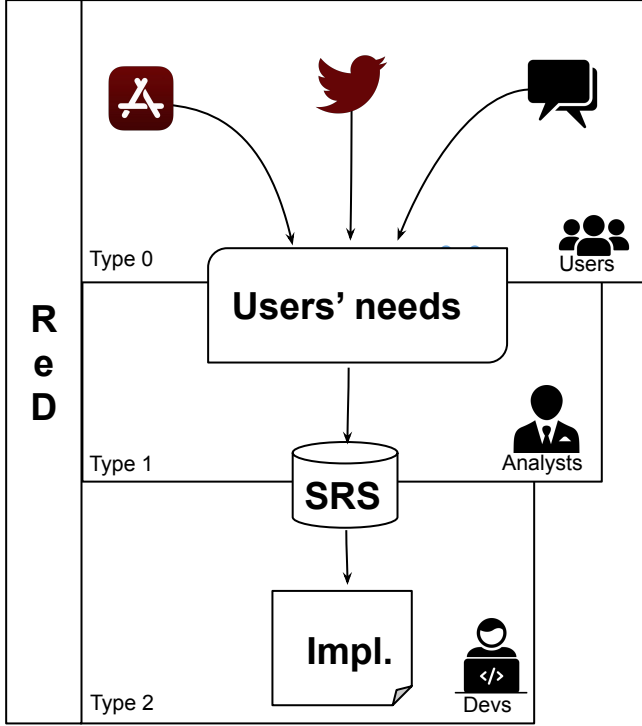
We define three types of ReD (see Figure 1).

Fig. 1: Types of Requirements Debt (ReD) incurred, their relationship, and main stakeholder involved.

A. ReD Type 0: Incomplete Users' needs

Represents the debt incurred when neglecting users' needs expressed using feedback channels. Such channels are, for example, app stores, social media, and interviews with customers. In particular, this type of debt is incurred when i) within a channel, not all users' needs are captured (e.g., due to the complexity of processing large amount of feedback) and ii) one or more relevant channels are not considered. In both cases, the incurred ReD can be implicit (e.g., causing an unplanned cost) or explicit (e.g., necessary due to a deadline).

How to detect. Currently, i) could be addressed by leveraging techniques for automatically classifying users' feedback and stakeholders, summarizing it, and recommend new features based on it. One example is the work of Maalej et al. which applies such techniques for app stores analytics [12]. Regarding ii) Nayebi et al. [17] presents the limitations of app stores in acquiring users' needs. The authors show that Twitter can provide additional information which app developers can

exploit to deliver better products. Moreover, feedback reported in Twitter is more objective than the one reported in app reviews. This work shows the importance of investigating several sources of users' needs and presents an initial approach to detect whether all relevant channels are considered.

How to quantify. ReD Type 0 can be quantified as the cost to formalize and implement the neglecting needs (*Principal*). We should considered two extra extra effort (*Interest*) related to:

- the current development stage, as implementing a neglected need is more expensive in an advanced stage,
- which components need to be modified to fix ReD. For example, implementing a neglected users' need impacting the graphic user interface will cost less than one impacting the architecture.

How to pay back. Once the neglected users' need is identified, it is formalized and included in the software requirements specification document.

B. ReD Type 1: Requirement smells

Represents the debt incurred when a requirements engineer, business analyst, or developer (i.e., *analyst*) formalizes users' needed into SRS. Femmer et al. [18] defined a set of Requirements Smells (Table I)—i.e., linguistic constructs which can indicate a violation of the ISO29148 standard for requirements quality. If such ambiguity is not removed the requirement can be wrongly implemented, hard to reuse, evaluate and extend.

How to detect. Like code smells, requirements smells do not necessarily lead to a defect, can be (semi-)automatically detected within a SRS document, and removed using standard techniques. These techniques leverage natural language processing (e.g., POS tagging) or simple dictionary lookups to identify problematic terms and language constructs (Table I).

How to quantify. ReD Type 1 can be quantified as the cost to fix the requirement smells within a SRS (*Principal*). However, the cost related to the harmfulness of each requirement smell (*Interest*) needs to be considered. With harmfulness, we mean the different negative impact that each requirements smell can have on activities relying on SRS.

How to pay back. As for code smells, refactoring (e.g., removing a problematic language construct leading to ambiguity while maintaining the original goal of the specification) is needs to be applied to pay back this type of ReD.

C. ReD Type 2: Mismatch implementation

Represents the debt incurred when developers implement a solution to a requirement problem. This type captures the mismatch between stakeholders' goal framed in the SRS and the actual system implementation. This type of debt can be also incurred when the requirements problem, framed in the SRS, changes while the implementation does not change accordingly [6]. A sub-par implementation can be the result of the incurred Type 1 debt.

TABLE I: Requirement Smells [18]

Requirement Smells	Description	Detection Strategy
Subjective Language	"Subjective Language refers to words of which the semantics is not objectively defined, such as user friendly, easy to use, cost effective"	Dictionary
Ambiguous Adverbs and Adjectives	"Ambiguous Adverbs and Adjectives refer to certain adverbs and adjectives that are unspecific by nature, such as almost always, significant and minimal"	Dictionary
Loopholes	"Loopholes refer to phrases that express that the following requirement must be fulfilled only to a certain, imprecisely defined extent"	Dictionary
Open-ended, non-verifiable terms	"Open-ended, non-verifiable terms are hard to verify as they offer a choice of possibilities, e.g. for the developers"	Dictionary
Superlatives	"Superlatives refer to requirements that express a relation of the system to all other systems"	Morphological Analysis, POS tagging
Comparatives	"Comparatives are used in requirements that express a relation of the system to specific other systems or previous situations"	Morphological Analysis, POS tagging
Negative Statements	"Negative Statements are statements of system capability not to be provided. Some argue that negative statements can lead to under specification, such as lack of explaining the system's reaction on such a case"	Dictionary, POS tagging
Vague Pronouns	"Vague Pronouns are unclear relations of a pronoun"	POS tagging

How to detect. Detection of this type of ReD can be based on approaches for traceability between SRS and source code. Knowledge-based approaches (e.g., RE-KOMBINE [6]) can be used to monitor requirements for changes and understanding their impact on the current implementation of the system.

How to quantify. The interest on ReD Type 2 is the amount of change between the current implementation and the SRS. Accordingly, it is quantified as the cost of comparing the current implementation with the set of possible changes [19] (*Principal*) plus the implementation of the selected change (*Interest*).

How to pay back. The actions for paying back this type of ReD consists in the implementation of the best new solution matching the updated SRS.

IV. FUTURE ASSESSMENT

A. ReD concept preliminary validation

Our next step is to preliminary validate the conceptualization of debt in requirements engineering. To that end, we designed a study to understand what practitioners consider as debt during requirements engineering process, compare it to the ReD conceptualization, and understand their motivations to incur the different types of ReD.

We will carry out an exploratory study (Figure 2), structured as a mixed research method, composed by a set of interviews, a focus group, and a final set of group interviews.

Based on these results, we will design and conduct detailed case studies, involving companies in order to monitor the requirement elicitation process.

Fig. 2: Study design for ReD concept validation.

Interviews. The first round of interviews will be carried out by means of a questionnaire based on open-ended questions to avoid driving the interviewee to a predefined set of answers. The interviews will be organize in three sections.

1) Personal and company information. We aim to collected the profile of the practitioners, considering age, country, gender, predominant roles, and working experience in requirement engineering. Moreover, we will collect the organization size via the number of employees and the common application domain.

2) Requirement Debt (ReD). We will include questions regarding the three Requirements Debt types (Type 0, 1 and 2) as defined in Section III. We will ask the practitioners to evaluate and discuss these definitions from their point of view.

3) Perceived Critically of Requirement Dept Types. We aim to capture the perception of requirements issues from our respondents. We will ask practitioners to rate their concerns about requirement debt and what they consider harmful.

Focus group. Relevant issues and problems are freely discussed and the answers provided during the interviews will cluster into topics. Open discussion can reveal the type of information that can be helpful in outlining key issues in each of the three types of ReD.

Group interviews. The last step will be executed with the support of a closed-ended questionnaire, based on the clustered answers identified in the focus group. The interviewer will explain each question to the participants who answered to the questions on a paper-based questionnaire. The interviews will be organized considering only the questions related to *Requirement Debt (ReD)* and *Perceived Critically of Requirement issues*.

Recruitment and Data Collection. The study will be conducted by invitation only to have a better control over the individual respondents. The strategy to define an invitation list is two-fold, i) requirements engineers, business analysts, and software developers within companies in the NaPiRE network [20] and among our contacts, and ii) software developers sampled from the app stores (e.g., Google Play Store, Apple Store). The latter are especially important to assess Type 0

ReD as they usually have direct access to users through the stores feedback and review functionalities.

B. Requirement smells harmfulness

In the immediate future, we plan to elaborate on Type 1 ReD as there is no empirical evidence of harmfulness of requirement smells, according to the definition and the detection approach proposed by Femmer et al. [18]. We will follow the approaches widely adopted to assess the harmfulness of code smells on different software qualities [21], [22], [23], [24]. We will triangulate data from requirements platforms, such as issue trackers and requirements repositories platforms [25], with studies involving requirements engineers, business analysts, and software developers [26].

V. RELATED WORK

In this section, we report key related work on Technical Debt (TD) and Requirement Technical Debt (ReD) evaluation and management.

A. Technical Debt

Different approaches and strategies have been proposed to evaluate TD. Nugroho et al. [27] proposed an approach to quantify debts in terms of cost to fix technical issues and its interest. They monitored data from 44 software systems and empirically validated the approach in a real system.

Seaman et al. [15] proposed a TD management framework that formalizes the relationship between cost and benefit in order to improve software quality and help decision making process during maintenance activities.

Zazworka et al. [28] investigated source code analysis techniques and tools to identify code debt in software systems, focusing on TD interest and TD impact on increasing defect- and change-proneness. They applied four TD identification techniques (code smells, automatic static analysis issues, grime buildup, and modularity violations) on 13 versions of the Apache Hadoop open source software project. They collected different metrics, such as code smells and code violations. The results showed good correlation between some metrics and defect and change proneness, such as Dispersed Coupling and modularity violations.

Different approaches or strategies have been proposed to manage TD. Guo et al. [29] proposed a portfolio approach in order to help the software manager in decision making. This approach provides a new perspective for TD management.

Nord et al. [30] defined a measurement-based approach to develop metrics in order to strategically managing TD. This approach could optimize the development cost over time without stopping the development process. They successfully applied the approach to an ongoing system development effort.

B. Related Work on Requirement Technical Debt

At the best of our knowledge, there are few works that investigated Requirement Technical Debt proposing some approach to evaluate and take under control this type of debt.

Ernst et al. [6] defined technical debt in requirements as "*the distance between the implementation and the actual state of*
the world" They conceptualized a requirements modeling tool, `RE-KOMBINE`, that 1) identifies technical debt by means of the notion of optimal solutions to a requirements and 2) allows to understand how implementations match stakeholder goals.

Abad and Ruhe [31] defined a systematic method to manage requirements-related decisions. The methods includes several factors that affect Technical Debt; the authors extend the concept to requirements and use historical project data to provide a predictive model for requirements decisions with the goal of reducing uncertainty.

Moreover, Wattanakriengkrai et al. [32] investigated self-admitted requirement debt—defined as "*source code comments deliberately created by developers in order to demonstrate that some parts of the code are missing, incomplete, or cannot satisfy the requirement of clients*"—that developers clearly identify in the code due to requirements incompleteness. Based on this definition, they show an approach to identify requirement self-admitted technical debt on 10 open source projects analyzed using text processing techniques.

VI. CONCLUSION

Despite the importance of requirements elicitation and management during software development process, there is still no consensus in research whether Requirement Debt should be considered as a type of technical debt and a lack of formalization in the literature.

In this paper, we challenge the current definition of ReD, extending it with upstream requirements engineering activities involving the elicitation of requirements and their translation into specifications.

Our definition of ReD is the first step towards creating a framework that stakeholders can use to make decisions regarding when to incur debt, at what costs, when to pay it back, and how to monitor it.

Our vision of ReD will be empirically evaluated in a series of studies with industry partners and individual stakeholders.

REFERENCES

[1] W. Cunningham, "The wycash portfolio management system," in *OOP-SLA '92*, 1992.

[2] A. Martini, J. Bosch, and M. Chaudron, "Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study," *Information and Software Technology*, vol. 67, pp. 237 – 253, 2015.

[3] T. Besker, A. Martini, R. E. Lokuge, K. Blincoe, and J. Bosch, "Embracing technical debt, from a startup company perspective," in *International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2018, pp. 415–425.

[4] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka, "Managing technical debt in software-reliant systems," in *Workshop on Future of Software Engineering Research*, 2010, pp. 47–52.

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

[6] N. A. Ernst, "On the role of requirements in understanding and managing technical debt," in *Proceedings of the Third International Workshop on Managing Technical Debt*, ser. MTD '12, 2012, pp. 61–64.

[7] K. Schmid, "On the limits of the technical debt metaphor some guidance on going beyond," in *4th International Workshop on Managing Technical Debt (MTD)*, 2013, pp. 63–66.

[8] N. S. R. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spínola, "Towards an ontology of terms on technical debt," in *International Workshop on Managing Technical Debt*, 2014, pp. 1–7.

[9] V. Lenarduzzi and D. Taibi, "Mvp explained: A systematic mapping study on the definitions of minimal viable product," in *42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug 2016, pp. 112–119.

[10] D. Taibi, V. Lenarduzzi, A. Janes, K. Liukkunen, and M. O. Ahmad, "Comparing requirements decomposition within the scrum, scrum with kanban, xp, and banana development processes," in *Agile Processes in Software Engineering and Extreme Programming*. Springer International Publishing, 2017, pp. 68–83.

[11] V. Lenarduzzi, T. Besker, D. Taibi, A. Martini, and F. A. Fontana, "Technical debt prioritization: State of the art. a systematic literature review," 2019.

[12] W. Maalej, M. Nayebi, T. Johann, and G. Ruhe, "Toward data-driven requirements engineering," *IEEE Software*, vol. 33, no. 1, pp. 48–54, 2015.

[13] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "The financial aspect of managing technical debt: A systematic literature review," *Information and Software Technology*, vol. 64, pp. 52 – 73, 2015.

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

[15] C. B. Seaman and Y. Guo, "Measuring and monitoring technical debt," *Advances in Computers*, vol. 82, pp. 25–46, 12 2011.

[16] A. Martini and J. Bosch, "An empirically developed method to aid decisions on architectural technical debt refactoring: Anacondebt," in *38th International Conference on Software Engineering Companion*, ser. ICSE '16, 2016, pp. 31–40.

[17] M. Nayebi, H. Cho, and G. Ruhe, "App store mining is not enough for app improvement," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2764–2794, Oct 2018.

[18] H. Femmer, D. M. Fernández, S. Wagner, and S. Eder, "Rapid quality assurance with requirements smells," *Journal of Systems and Software*, vol. 123, pp. 190–213, 2017.

[19] N. A. E., S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? manage it? ignore it? software practitioners and technical debt," in *Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, 2015, pp. 50–60.

[20] D. M. Fernández, S. Wagner, M. Kalinowski, M. Felderer, P. Mafra, A. Vetrò, T. Conte, M.-T. Christiansson, D. Greer, C. Lassenius *et al.*, "Naming the pain in requirements engineering," *Empirical software engineering*, vol. 22, no. 5, pp. 2298–2338, 2017.

[21] S. M. Olbrich, D. Cruzes, and D. Sjøberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *IEEE International Conference on Software Maintenance, ICSM*, 09 2010, pp. 1–10.

[22] D. Sjøberg, A. Yamashita, B. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, pp. 1144–1156, 08 2013.

[23] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some code smells have a significant but small effect on faults," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, pp. 33:1–33:39, Sep. 2014. [Online]. Available: http://doi.acm.org/10.1145/2629648

[24] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation," *Empirical Softw. Engg.*, vol. 23, no. 3, pp. 1188–1221, Jun. 2018.

[25] V. Lenarduzzi, , and N. S. D. Taibi, "The technical debt dataset," in *The Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE'19)*, 2019.

[26] D. Taibi, A. Janes, and V. Lenarduzzi, "How developers perceive smells in source code: A replicated study," *Information and Software Technology*, vol. 92, pp. 223 – 235, 2017.

[27] A. Nugroho, J. Visser, and T. Kuipers, "An empirical model of technical debt and interest," in *Workshop on Managing Technical Debt*, ser. MTD '11, 2011, pp. 1–8.

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

[29] Y. Guo and C. Seaman, "A portfolio approach to technical debt management," in *Workshop on Managing Technical Debt*, ser. MTD '11, 2011, pp. 31–34.

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

[31] Z. S. H. Abad and G. Ruhe, "Using real options to manage technical debt in requirements engineering," in *23rd International Requirements Engineering Conference (RE)*, 2015, pp. 230–235.

[32] S. Wattanakriengkrai, R. Maipradit, H. Hata, M. Choetkiertikul, T. Sunetnanta, and K. Matsumoto, "Identifying design and requirement self-admitted technical debt using n-gram idf," in *Workshop on Empirical Software Engineering in Practice (IWESEP)*, 2018, pp. 7–12.