The Journal of Systems and Software 142 (2018) 92–114

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

# Early evaluation of technical debt impact on maintainability

José M. Conejero[a,*], Roberto Rodríguez-Echeverría[a], Juan Hernández[a], Pedro J. Clemente[a], Carmen Ortiz-Caraballo[b], Elena Jurado[a], Fernando Sánchez-Figueroa[a]

[a] *Quercus Software Engineering Group, University of Extremadura, Avda. de la Universidad, s/n, 10071, Spain*
[b] *Escola d'Enginyeria d'Igualada, Universitat Politècnica de Catalunya, Av. Pla de la Massa, n° 8, 08700 Igualada, Spain*

A R T I C L E I N F O

*Article history:*
Received 15 March 2017
Revised 24 March 2018
Accepted 18 April 2018
Available online 21 April 2018

*Keywords:*
Technical Debt indicator
Requirements
Modularity anomalies
Maintainability
Empirical evaluation

A B S T R A C T

It is widely claimed that Technical Debt is related to quality problems being often produced by poor processes, lack of verification or basic incompetence. Several techniques have been proposed to detect Technical Debt in source code, as identification of modularity violations, code smells or grime buildups. These approaches have been used to empirically demonstrate the relation among Technical Debt indicators and quality harms. However, these works are mainly focused on programming level, when the system has already been implemented. There may also be sources of Technical Debt in non-code artifacts, e.g. requirements, and its identification may provide important information to move refactoring efforts to previous stages and reduce future Technical Debt interest. This paper presents an empirical study to evaluate whether modularity anomalies at requirements level are directly related to maintainability attributes affecting systems quality and increasing, thus, system's interest. The study relies on a framework that allows the identification of modularity anomalies and its quantification by using modularity metrics. Maintainability metrics are also used to assess dynamic maintainability properties. The results obtained by both sets of metrics are pairwise compared to check whether the more modularity anomalies the system presents, the less stable and more difficult to maintain it is.

## 1. Introduction

Since Technical Debt was firstly introduced in Cunningham (1992), many approaches have emerged to identify (Vetro' et al., 2010; Wong et al., 2011; Schumacher et al., 2010), estimate (Chin et al., 2010; Curtis et al., 2012a; Letouzey and Ilkiewicz, 2012; Marinescu, 2012) or, in general, deal with Technical Debt by different techniques (Ramasubbu and Kemerer, 2014). As the authors state in Kruchten et al. (2012), "*most authors agree that the major cause of Technical Debt is schedule pressure, e.g. ignoring refactorings to reduce time to market*" (Abad and Ruhe, 2015). However, as they also claim, Technical Debt is also related to quality problems being often produced by *carelessness, lack of education, poor processes, lack of verification or, even, basic incompetence*. These origins of Technical Debt are called *unintentional debt* (Brown et al., 2010) and examples of these quality problems occasioned by Technical Debt are bad reusability and low understandability (Griffith et al., 2014), error-prone and higher number of defects (Zazworka et al., 2014), negative impact on robustness, performance, security and transferability (Curtis et al., 2012a, 2012b) or, especially, on maintainability issues like stability (Zazworka et al., 2014). A study conducted by Chen and Huang (2009) highlights that stability is one of the top 10 higher-severity software development problem factors which affect software maintainability. Moreover, maintainability is currently draining 60–90% of the total cost of software development (Chen and Huang, 2009; Erlikh, 2000; Hung, 2007).

To solve these issues, several techniques have been proposed in the literature to detect Technical Debt in source code, such as the identification of modularity violations (Wong et al., 2011), code smells (Schumacher et al., 2010; Marinescu, 2004), grime buildups (Gueheneuc and Albin-Amiot, 2001; Izurieta and Bieman, 2007) or the identification of violations of good programmer practices by using Automatic Static Analysis (ASA) approaches (Vetro' et al., 2010). Indeed, the combination of these four different techniques has been empirically evaluated in Zazworka et al. (2014) to test which practices perform better under different conditions and how they could complement each other to estimate Technical Debt interests (quality harms). Technical debt interest may be defined as *the payment in the form of extra time, effort, and cost to address future changes in a project* (Abad and Ruhe, 2015). Similarly, in Ramasubbu and Kemerer (2014), Griffith et al. (2014),

* Corresponding author.
*E-mail addresses:* chemacm@unex.es (J.M. Conejero), rre@unex.es (R. Rodríguez-Echeverría), juanher@unex.es (J. Hernández), pjclemente@unex.es (P.J. Clemente), carmen.ortiz@eei.upc.edu (C. Ortiz-Caraballo), elenajur@unex.es (E. Jurado), fernando@unex.es (F. Sánchez-Figueroa).

https://doi.org/10.1016/j.jss.2018.04.035
0164-1212/© 2018 Elsevier Inc. All rights reserved.

Curtis et al. (2012b), and Zazworka et al. (2011), the authors conducted studies where they empirically evaluated the relation among different Technical Debt indicators and software quality characteristics in order to test whether the former are really related to the latter.

What all these works have in common is that they are focused on the programming level, when the system has already been implemented (if not completely, at least, partially). However, as claimed in Li et al. (2014), *Technical Debt can span all the phases of the software lifecycle* and there may also be sources of Technical Debt in non-code artifacts (Brown et al., 2010), e.g. requirements documents. Therefore, its identification at early stages of development may provide developers with important information to apply refactoring approaches (e.g. based on aspect-oriented techniques Moreira et al., 2013; Jacobson and Ng, 2004; Jacobson, 2003) improving, thus, modularity also at source code and therefore reducing Technical Debt at latest development stages (or, at least, reducing the future global *interest*). The reality is that requirements always change and Technical Debt is inevitable (Allman, 2012), however, the issue is not eliminating debt, but rather reducing it or even moving its identification to previous stages. Indeed, this is more important if we consider that *those who incurred the debt may usually not be the same as those who will have to re-pay later* (Brondum and Zhu, 2012).

Nevertheless, to the best of our knowledge, little effort has been dedicated to study the implications of Technical Debt at earlier stages of development. There are some works that have dealt with the definition of Technical Debt at the requirements level (Abad and Ruhe, 2015; Ernst, 2012) or its relation with architectural dependencies (Li et al., 2014; Brondum and Zhu, 2012). Even, these types of debts have been described in the mapping study introduced in Alves et al. (2016) as Requirements and Architecture Debts. However, the empirical evaluation of the quality problems produced by Technical Debt at early stages has been neglected in the literature so far. Based on this assumption, we have formulated the main question that we try to answer in this work: *is there a relationship between Technical Debt indicators at the requirements level and software quality?* Concretely, we focus on modularity violations (a well-known Technical Debt indicator Wong et al., 2011; Alves et al., 2016) and software stability (a quality attribute related to maintainability International Organization of Standardization, 2014). Thus, our main question is reformulated as follows: *is there a relationship between modularity anomalies at the requirements level and system stability?* The existence of this relationship would provide empirical evidence of the harmful relationship between Technical Debt and software quality at early stages of development.

To tackle the problem of answering this question, this paper presents an empirical study where we evaluate whether modularity anomalies at the requirements level occasioned by crosscutting concerns (Baniassad et al., 2006) are directly related to instability of the system, which would increase its interest. The empirical study is supported by the application of a conceptual framework defined in previous work (Conejero, 2010). The framework allows the identification of modularity violations based on scattering, tangling and crosscutting at any abstraction level but concretely at the requirements level. Moreover, based on this conceptual framework a set of software metrics were defined to quantify the Degree of Crosscutting properties that a system may have. In this work, these metrics are validated by comparing them with similar metrics introduced by other authors, whilst their utility is illustrated by comparing them with a set of metrics that measure stability. All the metrics are applied to measure both modularity and stability properties in three different software product lines (with different releases) and the measurements obtained are pairwise compared to test whether those metrics are correlated and to find an answer for our main question.

The rest of the paper is organized as follows. Section 2 briefly introduces the conceptual framework that supports the study by providing a method to identify crosscutting properties at requirements level. Section 3 presents the settings for our empirical study by introducing the hypothesis established, the measures used and the systems considered. Section 4 shows the results obtained and it discusses their interpretation according to our main hypothesis. Section 5 presents an evaluation of the metrics in order to select the most representative for future studies. Section 6 presents the threats to validity for this study. Finally, Section 7 discusses the related work and Section 8 concludes the paper.

## 2. Background

A concern is an interest, which pertains to the system's development, its operation or any other matters that are critical or otherwise important to one or more stakeholders (van den Berg et al., 2005). The term concern is closely related to the term feature (used in the Software Product Line context) in the sense of being a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems (Kang et al., 1990). Software modularity is mainly determined by the concept of Separation of concerns (Dijkstra, 1976), the design principle that proposes the proper encapsulation of systems' concerns into separate entities. One of the main advantages of separation of concerns is the significant reduction of dependencies between these features or concerns. However, concern independence is not always fully achieved and modularity anomalies arise usually occasioned by the well-known concern properties of scattering, tangling and crosscutting. Crosscutting (usually described in terms of scattering and tangling) denotes the situation where a concern may not be completely encapsulated into a single software component but spread over several artifacts and mixed with other concerns due to a poor support for its modularization (van den Berg et al., 2005).

In order to detect these modularity anomalies, crosscutting identification approaches come to the scene. Next section introduces our previous work where a conceptual framework for identifying and characterizing crosscutting properties was proposed. This framework was independent of any particular software development stage. Therefore, it may be applied at stages previous to implementation, e.g. at requirements stage.

### *2.1. A conceptual framework for analysing modularity anomalies*

In Conejero (2010) a conceptual framework was presented where formal definitions of concern properties, such as scattering, tangling, and crosscutting, were provided. This framework is based on the study of trace dependencies that exist between two different domains. These domains, which are generically called Source and Target, could be, for example, concerns and requirements descriptions, respectively or features and use cases in a different situation. We use the term Crosscutting Pattern (Fig. 1) to denote the situation where Source and Target are related to each other by means of trace dependencies.

From a mathematical point of view, the Crosscutting Pattern indicates that the Source and Target domains are related to each other by a mapping. This mapping is the trace relationship that exists between the Source and Target domains, and it can be formalized as follows:

According to Fig. 1, there exists a multivalued function $f'$ from Source to Target domain such that if $f'(s)=t$, then there exists a trace relation between $s \in Source$ and $t \in Target$. Analogously, we can define another multivalued function $g'$ from Target to Source