Information and Software Technology 000 (2017) 1–16

Contents lists available at ScienceDirect

Information and Software Technology

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

A semi-automated framework for the identification and estimation of Architectural Technical Debt: A comparative case-study on the modularization of a software component

Antonio Martini[a,*], Erik Sikander[b], Niel Madlani[b]

[a] University of Oslo, Department of Informatics Programming and Software Engineering Group, Oslo, Norway
[b] Chalmers University of Technology, Software Engineering Division Gothenburg, Sweden

ARTICLE INFO

Article history:
Received 14 December 2016
Revised 30 July 2017
Accepted 11 August 2017
Available online xxx

Keywords:
Measurement System
Software management
Refactoring
Estimation
Technical Debt
Modularization
Software Architecture

ABSTRACT

Context: Research and industry's attention has been focusing on developing systems that enable fast time to market in the short term, but would assure a sustainable delivery of business value and maintenance operations in the long run. A related phenomenon has been identified in Architectural Technical Debt: if the system architecture is sub-optimal for long-term business goals, it might need to be refactored. A key property of the system assuring long-term goals is its modularity, or else the degree to which components are decoupled: such property allows the product to be evolved without costly changes pervading the whole system. However, understanding the business benefits of refactoring to achieve modularity is not trivial, especially for large refactorings involving substantial architectural changes.
Objective: The aim of this study was to develop a technique to identify Architectural Technical Debt in the form of a non-modularized component and to quantify the convenience of its repayment.
Method: We have conducted a single, embedded case study in a large company, comparing a component before and after it was refactored to achieve modularity. We have developed a holistic framework for the semi-automated identification and estimation of Architectural Technical Debt in the form of non-modularized components. We then evaluate the technique reporting a comparative study of the difference in maintenance and development costs in two coexisting systems, one including the refactored component and one including the non-refactored one.
Results: The main contributions are a measurement system for the identification of the Architectural Technical Debt according to the stakeholders' goals, a mathematical relationship for calculating and quantifying its interest in terms of extra-effort spent in additional development and maintenance, and an overall decision framework to assess the benefit of refactoring. We also report context-specific results that show the estimated benefits of refactoring the specific case of Architectural Technical Debt.
Conclusion: We found that it is possible to identify this kind of Architectural Technical Debt and to quantify its repayment convenience. Thanks to the developed framework, it was possible to estimate that the Architectural Technical Debt present in the component was causing substantial continuous extra-effort, and that the modularization would be repaid in several months of development and maintenance.

1. Introduction

Large software organizations need to achieve a sustainable strategy to balance short-term implementations, to quickly deliver value to customers, with solutions that would sustain their business in the long term. To illustrate such a phenomenon, a financial metaphor has been coined, which compares the act of implementing sub-optimal solutions, in order to meet short-term goals, to taking debt, which has to be repaid with interests in the long term. The term *Technical Debt* (TD) has been first coined at OOPSLA by Cunningham [1] to describe a situation in which developers take decisions that bring short-term benefits but cause long-term detriment of the software.

The up-to-date definition, developed by the Technical Debt community, is available in the Dagstuhl seminar report 16162 [2]:

> "*In software-intensive systems,* ***technical debt*** *is a design or implementation construct that is expedient in the short term, but sets up a technical context that can make a future change more costly or impossible. Technical debt is a contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability*"

[*] Corresponding author.
E-mail address: antonio.martini@ifi.uio.no (A. Martini).

http://dx.doi.org/10.1016/j.infsof.2017.08.005
0950-5849/© 2017 Published by Elsevier B.V.

Kruchten et al. [3] define several kinds of Technical Debt. In this paper, we focus on the Architectural Technical Debt (ATD) kind, which is Technical Debt that concerns the way in which the system has been architected. Architectural Technical Debt has been, in recent years, elaborated and refined in Software Engineering literature: Tom et al. [4] first mentioned Architectural Technical Debt (ATD, categorized together with Design Debt). A further classification can be found in [3], where ATD is regarded as the most challenging TD to be uncovered by Kruchten et al. In particular, such study highlights that there is a lack of research and tool support to manage ATD in practice. ATD has been further recognized in a recent systematic mapping [5] on TD. Finally, a systematic literature review on ATD has been published recently [6]. Such recent body of knowledge highlights the lack of empirical research and the critical need of management tools to identify and estimate Architectural Technical Debt. The main goal of this paper is to develop a viable technique to support such need.

According to the glossary proposed in [7], we recall here that the ATD present in a system is a set of "sub-optimal architectural solutions". An optimal architecture refers to the architecture identified by the software and system architects as the optimal trade-off when considering the concerns collected from the different stakeholders (which is the "desired" architecture). Such architecture needs to support the business goals of the organization. An example of ATD might be the presence of structural violations [8]: a component might not be well modularized, which causes extra costs when the business requirements change. However, it is important to notice that an optimal architectural trade-off might change over time, due to business evolution and to new information collected during implementation [9]. It is important to notice that it might be difficult to anticipate an optimal solution that remains sustainable for several years. For this reason, it becomes important to continuously monitor the architecture and to identify a *costly* emerging sub-optimality (ATD). The *identification* [5] of ATD is therefore one of the main activities of software architects. In this paper, we seek to develop an automated solution that would support the identification of a specific ATD kind.

ATD needs to be identified; however, it is at least as important to estimate the convenience of refactor it. The costs related to ATD are based on two main components, reported in TD literature: principal and interest. The *principal* is the cost of refactoring, necessary to remove the debt. In a concrete example, if a system contains ATD in the form of a non-modularized component, the principal is the cost of reworking the component and split it into multiple, loosely-coupled components. The *interest* is considered the extra-cost that is associated with the ATD, or else the extra-cost that would not be paid by the organization if the ATD would not be present in the system. For example, if the system is not well modularized, each change (for example a bug fix) might require the developers to change several parts of the system, which would require more time and resources from the developers [10]. An ATD item is usually refactored if the principal is considered less than the interest left to be paid [11,12]. Estimating principal and interest, to understand the convenience of refactoring ATD, is therefore one of the main goals of current research and one of the main need of software organizations. However, such task has proven to be extremely difficult. The goal of this paper is therefore to develop a technique to estimate the convenience of refactoring the ATD.

In summary, for software organizations, it is important to identify ATD and to understand if repaying it would be convenient with respect to the cost of not repaying the debt (calculated as interest). Architects and software managers need methods and tools to systematically analyze and estimate the impact of ATD, in order to clearly show the urgency of the refactoring [13]. There are only few quantitative approaches revealing the business value of repaying the Technical Debt [10]. Such approaches are not related to quantify the benefits of modularity. Although there exists several studies on identifying the lack of modular components [10], such studies do not connect the identification of such ATD with a systematic estimation of its principal and interest. Technical and non-technical experts need a holistic framework to identify ATD, in the form of lack of modularization, and to quantitatively estimate the benefits of removing it. This problem leads to our overall research question:

RQ – How can we identify and estimate, in a quantitative way, if a non-modularized component is worth refactoring?

This is the overall goal of the study, i.e. understanding if the component contains ATD in the form of lack of modularization and how convenient it would be to repay the debt. Hence, we need to know how much is the principal and the interest of such ATD. Using the ratio between principal and interest, we can calculate the cost/benefit of repaying the debt. In order to understand if the ATD needs refactoring, we have to first identify the ATD and then estimate the costs, which leads to the following RQs.

RQ1 – How can we identify if a component contains ATD in the form of lack of modularization?
RQ2 – How can we estimate the current extra-cost paid as interest of the ATD?
RQ3 – How can we estimate the long-term cost saved by modularizing the component?

We have conducted a case study in a large software company: we have studied a case in which the functionality of a component was modularized and used by new applications. We created a measurement system that identified the need of refactoring according to the stakeholders' goals. Then, by comparing the development and maintenance effort between the two solutions, we assessed the goals met by the stakeholders and we developed a formula quantifying the benefits of refactoring in terms of development months.

In Section 2 we explain our research methodology together with the relevant work used to define the measurement system to identify ATD and calculate the benefits of refactoring, in Section 3 we report the results, which are discussed in Sections 4 and 5 we draw our conclusions.

2. Research design

2.1. Case-study design

We conducted a single, embedded case-study, following the guidelines in [14]. The main case was a large company developing software. We can consider the case as embedded as we studied two instances of the component, refactored and non-refactored (the unit of analysis). The two units of analysis were studied in parallel.

The execution of the study followed three steps:

1. Elicitation of the goals related to the refactoring of ATD
2. Development of the framework for identification and estimation of ATD
   a. Development of the Measurement System according to the goals in point 1
   b. Development of the estimation
3. Evaluation of the framework, based on a comparative case-study of pre- and post- refactoring
   a. Quantitatively analyzing the source code and the versioning system
   b. Quantitatively and qualitatively inquiring the developers and architects on the results of the refactoring