Information and Software Technology xxx (2015) xxx–xxx

Contents lists available at ScienceDirect

Information and Software Technology

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

Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study

Antonio Martini *, Jan Bosch, Michel Chaudron

Computer Science and Engineering, Software Engineering, Chalmers University of Technology | Gothenburg University, Göteborg, Sweden

ARTICLE INFO

Article history:
Received 30 November 2014
Received in revised form 8 July 2015
Accepted 10 July 2015
Available online xxxx

Keywords:
Architectural Technical Debt
Software management
Software architecture
Agile software development
Software life-cycle
Qualitative model

ABSTRACT

Context: A known problem in large software companies is to balance the prioritization of short-term with long-term feature delivery speed. Specifically, Architecture Technical Debt is regarded as sub-optimal architectural solutions taken to deliver fast that might hinder future feature development, which, in turn, would hinder agility.
Objective: This paper aims at improving software management by shedding light on the current factors responsible for the accumulation of Architectural Technical Debt and to understand how it evolves over time.
Method: We conducted an exploratory multiple-case embedded case study in 7 sites at 5 large companies. We evaluated the results with additional cross-company interviews and an in-depth, company-specific case study in which we initially evaluate factors and models.
Results: We compiled a taxonomy of the factors and their influence in the accumulation of Architectural Technical Debt, and we provide two qualitative models of how the debt is accumulated and refactored over time in the studied companies. We also list a set of exploratory propositions on possible refactoring strategies that can be useful as insights for practitioners and as hypotheses for further research.
Conclusion: Several factors cause constant and unavoidable accumulation of Architecture Technical Debt, which leads to development crises. Refactorings are often overlooked in prioritization and they are often triggered by development crises, in a reactive fashion. Some of the factors are manageable, while others are external to the companies. ATD needs to be made visible, in order to postpone the crises according to the strategic goals of the companies. There is a need for practices and automated tools to proactively manage ATD.

1. Introduction

Large software industries strive to make their development processes fast and more responsive, minimizing the time between the identification of a customer need and the delivery of a solution. The trend in the last decade has been the employment of Agile Software Development (ASD) [1]. At the same time, the responsiveness in the short-term deliveries should not lead to less responsiveness in the long run. To illustrate such a phenomenon, a financial metaphor has been coined, which relates taking sub-optimal decisions in order to meet short-term goals to taking a financial debt, which has to be repaid with interests in the long term. Such a concept is referred as Technical Debt (TD), and recently it has been recognized as a useful basis for the development of theoretical and practical frameworks [2]. Tom et al. [3] have explored the TD metaphor and outlined a first framework in 2013. Part of the overall TD is to be related to architecture sub-optimal decisions, and it is regarded as Architecture Technical Debt (ADT) [4]. More precisely, ATD is regarded as implemented solutions that are sub-optimal with respect to the quality attributes (internal or external) defined in the desired architecture intended to meet the companies' business goals.

ATD has been recognized as part of TD in a recent (2015) systematic mapping study on TD [4]. However, such study highlights several deficiencies in the current body of knowledge: lack of reliable industrial studies, lack of focus on architecture anti-patterns and lack of studies involving the whole TD management process. In this paper we aim at filling such current gaps by investigating, in several companies, the overall phenomenon of accumulation and refactoring of ATD. The study of such subject would also contribute to ASD frameworks, by highlighting activities for enhancing agility in the task of developing and maintaining software architecture in large projects [5].

* Corresponding author.
E-mail addresses: antonio.martini@chalmers.se (A. Martini), jan.bosch@chalmers.se (J. Bosch), michel.chaudron@cse.gu.se (M. Chaudron).

http://dx.doi.org/10.1016/j.infsof.2015.07.005
0950-5849/

In the context of large-scale ASD, our research questions are:

RQ1: What factors cause the accumulation of ATD?
RQ2: How is ATD accumulated and refactored over time?
RQ3: What possible refactoring strategies can be employed for managing ATD?

In this paper we have employed a 18 months long, multiple-case study involving 7 different sites in 5 large Scandinavian companies in order to shed light on the phenomenon of accumulation and refactoring of ATD. We have analyzed the qualitative data obtained from more than 50 h of formal interviews complemented with continuous informal meetings with key roles involved in the architectural work, using a combination of inductive and deductive approach proper of Grounded Theory. We have qualitatively developed and evaluated a taxonomy of the factors to inform RQ1 and a set of models to inform RQ2. We have also derived some preliminary conclusions on which refactoring strategies can be applied and what effects they have to inform RQ3.

The main contributions of the papers are:

- A taxonomy of the causes for ATD: we present the factors for the explanation of the phenomena such as accumulation and refactoring of ATD. These factors might be studied and treated separately, and offer a better understanding of the overall phenomenon.
- Two qualitative models of the trends in accumulation and refactoring of ATD over time.
  - Crisis model – Shows the strictly increasing trend of ATD accumulation and how it eventually reaches a crisis point. We describe the evidences related to the occurrence of the crisis point and we connect such phenomenon to the different factors and their influence on the accumulation.
  - Phases model – Shows when ATD is currently accumulated and refactored during different software development phases. It helps identifying problem areas and points in time for the development of practices that would 1) avoid accumulation of ATD and/or 2) ease the refactoring of ATD. Such practices would be aimed at delaying the crisis point.
- Possible refactoring strategies: we analyze how different refactoring strategies might lead to best- and worst-case scenarios with respect to crisis points.
- A detailed description of an additional and in-depth industrial case, which contributes to empirically evaluate the factors and to analyze the relationships among them in a specific context.

The rest of the paper is structured as follows: Section 2 gives the reader more references and background on ATD and on the conceptual framework used in this study. In Section 3 we explain our research design: overall design, description of the cases, methods for data collection and analysis and evaluation of results. In Section 4 we list the results: factors causing ATD, models of accumulation of ATD over time, possible refactoring strategies, description of the in-depth case-study and the evaluation of results. In Section 5 we examine how the results inform the RQs, we discuss practical and theoretical implications of this study and we discuss the degree of validity of each result. We also point at limitations and open issues for future research, and we discuss the related work. We summarize the conclusions in Section 6.

2. Architecture and Technical Debt

2.1. Definition of ATD

ATD is regarded [3] as "sub-optimal solutions" with respect to an optimal architecture for supporting the business goals of the organization. Specifically, we refer to the architecture identified by the software and system architects as the optimal trade-off when considering the concerns collected from the different stakeholders. In the rest of the paper, we call the sub-optimal solutions *inconsistencies* between the implementation and the architecture, or *violations*, when the optimal architecture is precisely expressed by rules (for example for dependencies among specific components). However, it is important to notice that (in our studied cases) such optimal trade-off might change over time, as explained in this paper, due to business evolution and to information collected from implementation details. Therefore, it is not correct to assume that the sub-optimal solutions can be completely identified and managed from the beginning.

In the next section, we mention some classes of examples of what can be considered ATD. Such examples are extracted from another paper by the same authors on the same subject [7].

2.2. Examples of ATD

2.2.1. Dependency violations

The presence of architectural dependencies (for example at different component levels) which are considered forbidden in the (context-specific) architecture can be considered ATD. An example of this class of items is represented by a component that, when executed, should not trigger the execution of another component, as specified by the architects/architecture. A study on this problem has been conducted also by Nord et al. [8].

2.2.2. Non-uniformity of patterns and policies

Patterns and policies defined by the architecture (and the architects) might not be kept consistent through the system. For example, there might be a name convention applied in part of the system that is not followed in another part of the system. Another example is the presence of different design or architectural patterns used to implement the same functionality, such as different interaction patterns used among different components (we intend those differences that are not motivated by precise design constraints).

2.2.3. Code duplication (non-reuse)

Quite recognized in literature and in practice is the presence of very similar code (if not identical) in different parts of the system, especially in different products, not grouped into a reused component.

2.2.4. Temporal properties of inter-dependent resources

Some resources might need to be accessed by different part of the system (for simplicity we will take the example of having different components accessing the same resource). In these cases, the way in which a component interacts with the resource might change the interaction of other components with the same resource. This aspect is especially related to the temporal dimension: for example, the order with which two components change the status of a database or access it, would change the behavior of the system. For this reason, specific scheduling patterns can be designed, which are used for assuring the correct interaction of components. As a concrete example, we can mention the convention of having only synchronous calls to a certain component. An ATD item of this category is represented by the non-application of such patterns by some developers or teams.

2.2.5. Sub-optimal mechanism for non-functional requirements

Some non-functional requirements, such as scalability, performance or signal reliability, need to be recognized before or early during the development and need to be tested. The ATD items represent the lack of an implementation that would assure the

Please cite this article in press as: A. Martini et al., Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study, Inform. Softw. Technol. (2015), http://dx.doi.org/10.1016/j.infsof.2015.07.005