An Empirically Developed Method to Aid Decisions on Architectural Technical Debt Refactoring: AnaConDebt

Antonio Martini
Chalmers University of Technology
Software Engineering Division
Gothenburg, Sweden
antonio.martini@chalmers.se

Jan Bosch
Chalmers University of Technology
Software Engineering Division
Gothenburg, Sweden
jan.bosch@chalmers.se

ABSTRACT

Architectural Technical Debt is regarded as sub-optimal architectural solutions that need to be refactored in order to avoid the payment of a costly interest in the future. However, decisions on if and when to refactor architecture are extremely important and difficult to take, since changing software at the architectural level is quite expensive. Therefore it is important, for software organizations, to have methods and tools that aid architects and managers to understand if Architecture Technical Debt will generate a costly and growing interest to be paid or not. Current knowledge, especially empirically developed and evaluated, is quite scarce. In this paper we developed and evaluated a method, AnaConDebt, by analyzing, together with several practitioners, 12 existing cases of Architecture Debt in 6 companies. The method has been refined several times in order to be useful and effective in practice. We also report the evaluation of the method with a final case, for which we present anonymized results and subsequent refactoring decisions. The method consists of several components that need to be analyzed, combining the theoretical Technical Debt framework and the practical experience of the practitioners, in order to identify the key factors involved in the growth of interest. The output of the method shows summarized indicators that visualizes the factors in a useful way for the stakeholders. This analysis aids the practitioners in deciding on if and when to refactor Architectural Technical Debt items. The method has been evaluated and has been proven useful to support the architects into systematically analyze and decide upon a case.

Categories and Subject Descriptors

D.2.11 [**Software Architecture**]

General Terms

Management, Measurement, Design, Economics.

Keywords

Architectural Technical Debt, refactoring, decision making, estimation, method, design research, empirical study.

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. However, responsiveness in the short-term deliveries should not lead to reduced responsiveness in the long run. To illustrate such a phenomenon, a financial metaphor has been coined, which compares implementing sub-optimal solutions in order to meet short-term goals to the taking debt, which has to be repaid with interests in the long term. Such a concept is referred to as Technical Debt (TD), and recently it has been recognized as a useful basis for the development of theoretical and practical frameworks [1]. Part of the overall TD is to be related to architecture sub-optimal solutions, and it is regarded as Architecture Technical Debt (ADT)[1]. ATD is regarded as inconsistencies in the implementation towards the intended architecture for supporting the business goals of the organization. An example of ATD might be the presence of structural violations [2]. A recent study on the subject has highlighted that ATD might be very difficult or even impossible to avoid, for a number of reasons [3] (also external to the organization). It is therefore important to understand what ATD is present in the systems and to timely prioritize the refactoring of the most dangerous one before its interest would lead to development crises.

Given the high cost of architectural changes, prioritizing architectural refactorings is a challenge for software companies in practice [3]. A study showed how the most dangerous ATD is the one for which the interest is growing over time [4]. In practice it's important to repay the debt before the cost of refactoring (principal) becomes more expensive than the cost of not repaying the debt (interest). A method that would aid the architects and the stakeholders to systematically analyze and estimate the impact of ATD, in order to clearly show the urgency of the refactoring, is highly needed by the practitioners [5].

It is also important, for the stakeholders, to know *when* to refactor. Strategically, given a fixed amount of resources, organizations need to prioritize refactorings against feature development: deciding to postpone the refactoring might allow the company to deliver more value for the customers earlier on, which is one of the business imperatives of the companies. On the other hand, a fixed amount of resources forces companies to choose which refactorings to tackle first. This choice is also quite difficult, since it implies the ability to compare and prioritize the refactorings among themselves, which is also quite difficult in practice.

The first research question that we want to answer is therefore:

RQ1: How can ATD be systematically estimated in order for the stakeholders to take informed decisions on *whether* and *when* to prioritize the refactoring?

According to the Technical Debt framework, in order to take such decisions, it's important to understand the growth of the interest over time [4]. We therefore want to answer the following RQ:

RQ2: How can the growth of interest for a given ATD item be estimated?

In order to answer these questions, we have empirically developed a method (AnaConDebt), together with several architects across 6 large software companies. The research process, based on the direct and iterative observation of the stakeholders analyzing and estimating real ATD cases, assures that the method is quite comprehensive of the important factors needed to estimate the growth of interest and therefore to take a refactoring decision. The main contribution of this papers is a method, developed and evaluated in practice, comprehensive of the factors necessary for the estimation and prioritization of refactoring ATD items.

The paper is organized as follows: first we describe the Technical Debt framework in section 2, and we outline the Research Design in section 3. Then we show and discuss our results in section 4. We discuss the limitations, threats to validity and related work in section 5, and we conclude in section 6.

2. BACKGROUND

2.1 Technical Debt theoretical framework

2.1.1 Definition of ATD

ATD is regarded [6] 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 (which is usually a "desired" architecture). However, it's important to notice that (in our studied cases) such optimal trade-off might change over time, as explained in [3], due to business evolution and to new information collected from implementation details. Therefore, it's not realistic to assume that the sub-optimal solutions can all be identified and managed from the beginning. For this reason, it becomes important to continuously monitor ATD and manage continuous refactoring rather than relying only on an upfront design.

2.1.2 Previous research on ATD

The term *Technical Debt* (TD) has been first coined at OOPSLA by W. Cunningham [7] to describe a situation in which developers take decisions that bring short-term benefits but cause long-term detriment of the software. The term has recently been further studied and elaborated in research: in 2013 Tom et al. [6] conducted an exploratory case study technique that involves multi-vocal literature review, supplemented by interviews, in order to draw a first categorization of TD and the principal causes and effects. In that paper we can find the first mentioning of Architectural Technical Debt (ATD, categorized together with Design Debt). A further classification can be found in Kruchten et al. [1], where ATD is regarded as the most challenging TD to be uncovered since there is a lack of research and tool support in practice. Finally, ATD has been further recognized in a recent systematic mapping [8] on TD. Such recent research highlights the gap in the current scientific knowledge, which gives us the motivation for this work. Finally, ATD has been recognized as one of the most difficult to tackle, according to [5].

2.1.3 Previous research on management of TD

Some studies have been conducted on the management of TD, also supported by a dedicated workshop (MTD), usually co-located with premium conferences, such as ICSE and ICSME.

A first roadmap has been created in 2010 by Brown et al. [9]. In 2011 Guo et al. proposed an initial portfolio approach with the creation of TD *items*. The same authors proposed a further empirical study on tracking TD [10] Seaman et al. identified the theoretical importance of TD as risk assessment tool in decision making [11]. TD has also been used for defining part of a method for assessing software quality, SQALE [12]. Such model has been also implemented in a tool, but the main support is currently given on a source code level (very limited on the ATD aspect).

2.1.4 Models for technical debt

The studies in TD are quite recent: some models, empirical [13] or theoretical [14] have been proposed in order to map the metaphor to concrete entities in software development. We use a conceptual model comprehending the main components of TD:

- The *debt* is regarded as the actual technical issue. Related to the ATD in particular, we consider the ATD *item* [10] as a specific instance of the implementation that is sub-optimal with respect to the intended architecture to fulfill the business goals. We list a number of these cases in Table 1.
- The *principal* is considered the cost for refactoring the specific TD item, or else it is the cost for reworking the source code in order to have the ATD removed. However, as we show in this paper, the principal can include other costs.
- The *interest* of an ATD item consists on the extra-costs associated to the impact of the ATD. Such impact can be on the system, on the development process or even on the customer. A subtle part of the interest is affecting the principal. This means that one of the impacts of the sub-optimal solution is that the cost of removing it increases of a certain percentage over time, and this is considered as interest. In fact, the more steeply the principal grows over time, the more difficult and costly it will be to fix the ATD. We call this *interest on the principal*.

2.1.5 The time perspective

The TD framework is strongly related to time. Contrarily to having absolute quality models, the TD theoretical framework instantiates a relationship between the cost and the impact of a single sub-optimal solution over time. In particular, the metaphor stresses the short-term gain given by a sub-optimal solution against the long-term one considered optimal.

Such risk management practice is also very important in the everyday work of software architects, as mentioned in Kruchten [15] and Martini et al. [16]. Although research has been done on how to take decision on architecture development (such as ATAM, ALMA, etc. [17]), there is no empirical research about how sub-optimal architectural solutions (ATD) can be *continuously* managed.

2.1.6 Theoretical framework for refactoring decision

In order to reach the refactoring decision, we describe the goal of such decision and the information that is needed. As mentioned in the RQs, the goal of the stakeholders is to estimate *if* and *when* the refactoring should be performed.

"If" decision – According to the TD metaphor, the principal should be paid if it is less than the total interest [11]. A formal approach has been described in [14]. However, such formal approach, although useful for defining the initial framework to develop the method, was found already in [14] to be difficult to apply in practice. Also, it relies heavily on probability distribution of the events, which was found extremely difficult to calculate by the practitioners during our study.

However, the two mentioned frameworks agree on the fact that the decision *if* to refactor depend on the ratio:

$$\frac{CPrincipal}{TInterest} \tag{1}$$

CPrincipal is the current principal to be paid and *TInterest* is the total interest on the lifecycle of the system. The total interest usually cannot be calculated unless the lifecycle of the product is known, so *TInterest* is usually considered as the interest calculated at a chosen point in the (far) future. In other words, the stakeholders need to understand if the cost of refactoring now (principal) is less than the total interest paid from now to the end of the lifecycle of the product (or far enough in the future to be able to make predictions). If the result is greater than 1, it means that it is not convenient to pay the principal with respect to the interest that will be paid in the future. Notice that choosing a future point that is earlier than the final lifecycle of the product is a safe choice. In fact, in the worst case we take into account only part of the interest, which means that the principal will cover *at least* the considered interest. In this paper we do not describe *CPrincipal* and *TInterest* in terms of absolute *costs* in dollars, but in a set of factors with properties. In some cases a cost can be associated with a factor, while in other cases we need to have other kinds of values, such as ordinal (relative) values. We assume that to each of these factors can be associated a positive or a negative value that can increase or decrease over time.

"When" decision – The second part of RQ1 that we want to answer is *when* to refactor. For example, a typical practical question is: *"should we refactor now or we can wait 6 months?"*. To answer this question, the stakeholders need to know if refactoring at a chosen point in the future (F_1) is more or less convenient than refactoring now. F_1 can be chosen at any point in time between now and the future point that has been chosen for the *TInterest* (note that if chosen as *TInterest*, there would not be a difference between the *if* and the *when*). The decision relies on the following equation:

$$\frac{F_1Principal}{TInterest-F_1Interest} - \frac{CPrincipal}{TInterest} \tag{2}$$

The formula calculates the ratio between the principal at F_1 and the remaining interest calculated as the total interest *TInterest* minus the interest that will be already paid at F_1, and then subtracts (in practice, compares) the calculated ratio with respect to the same ratio about refactoring in the current situation. If the first term is higher than the second one, the results will be a negative number, which means that it will be less convenient to refactor later, since the gain of refactoring the debt with respect to the remaining interest will be less than now. If the result were sufficiently low (close to 0), it would mean that it is not urgent to refactor, since the refactoring is not going to give many benefits now with respect to perform it at future F_1. Notice that the former situation happens when the interest grows from the current situation to a point in the future F_1, and this is true also for the *interest on the principal* (described in section 2 and in 3.1.3): even if not shown in the formula for simplicity, we know that:

$$F_1Principal = CPrincipal + F_1IPrincipal \tag{3}$$

were $F_1IPrincipal$ is the part of the principal that has grown because of the interest. In other words, when the interest on the principal grows, the total principal for repaying the debt grows as well, which makes in general the refactoring less convenient.

The aim of this framework is to aid the refactoring decision. Even though in practice it might not be possible to give each factor a precise number, as explained earlier, the formulas are useful in order to reason about the relationship among the concrete factors involved in the decision. The concrete factors can be placed in one or the other part of the formula, "contributing" to one or the other component of the formulas. An example on the way of using the formulas is reported in the results section when we explain how to use the indicators produced by the method for decisions.

3. RESEARCH DESIGN

The overall research design is outlined in Figure 1. Since our goal was the development of a method, or else to design an artifact to be used in practice, we have followed the main guidelines related to such kind of research, called *design science research* [18]. Such strategy includes 6 activities: *Problem identification and motivation*, *Define the objective of a solution*, *Design and development*, *Demonstration*, *Evaluation* and *Communication*.

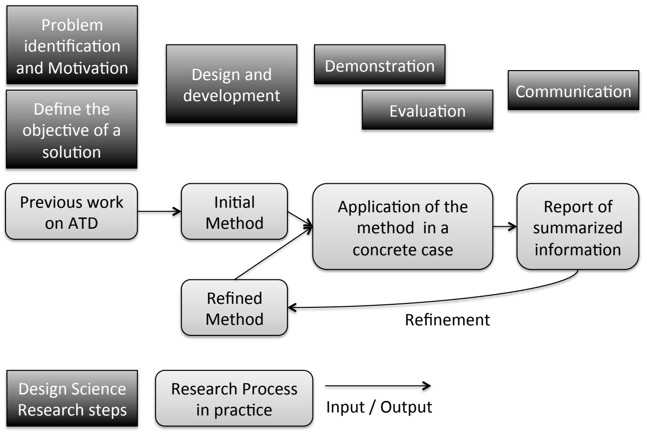
Problem identification and Motivation
Define the objective of a solution
Design and development
Demonstration
Evaluation
Communication
Previous work on ATD
Initial Method
Application of the method in a concrete case
Report of summarized information
Refined Method
Refinement
Design Science Research steps
Research Process in practice
Input / Output

Figure 1. Research process: practical steps according to the overall strategy based on Design Science Research

Problem identification: it is explained in section 1 in accordance to the literature review and the related research questions.

Define the objective of a solution: it is reported in section 2, by explaining the TD framework and what is required by the practitioners to reach a decision on refactoring.

Design and development phase: we created an initial method (see the following section *Initial Method*) based on previous work on Architecture Technical Debt, summarized in section 1 and 2.

Demonstration: in this phase, we used the prototyped method in order to observe the estimation process used by the architects when analyzing and estimating 12 real ATD items in practice. In order to do this, the researchers conducted specific sessions together with the architects of 6 companies (in the following, company A-F), in order to apply the method on 12 concrete and real cases that were evaluated at the time by the practitioners. In other words, the researchers and the architects used the method in practice to support the evaluation of ATD items.

The chosen companies were all large companies. Most of them (A-E) developed embedded software. We always analyzed large projects, consisting of millions of lines of codes and several components. Although we expected some variance among the companies, we found only few contextual factors that would make the method specific for the given company, for example the specific measures used in the data collection. However, the framework that we have used for the method and that we report here, proved to be quite generic. The cases used for developing and refining the method are described in Table 1.

Evaluation: before each session, we evaluated the previous results and refined the method to iteratively improve it. When the iterative process stopped yielding new important elements to be analyzed in the method, we evaluated the method by applying it on a detailed case. We also evaluated the method with the stakeholders (architects) to understand if such method would fulfill important requirements to be applied in practice.

Communication: after the application of the method, we created a report in which we summarized the most important information according to the decision-making goal defined in step 2. The goal was to understand if the interest of the ATD was growing, and therefore we have focused on representing the data collected according to such purpose. This was important to understand if the artifact was useful for its scope: taking a decision about if and when to conduct the refactoring.

This research process can be seen as a longitudinal case study, in which a method is created and evaluated. The difference is that, instead of developing the method offline and evaluating it once, we continuously refined the method with multiple and iterative evaluations of the artifact involving several stakeholders in several organizations.

Table 1 Cases used during the study in which we have applied the method

Case Id	Comp	Description
Case_A$_1$	A	Lack of good "communication mechanism" for different applications sharing a memory resource. Such TD created quality issues (bugs) for the users (customers)
Case_B$_1$	B	Quick fix of data structure to allow performance for a single project.
Case_B$_2$	B	The old version of an external library was used because it was difficult to adopt the new one.
Case_C1$_1$	C1	A common component was not designed optimally. Some applications have started using it. The component has been refactored, but the applications using it cannot be refactored because of other priorities, while new applications are using the new version.
Case_C1$_2$	C1	There is a non-allowed dependency between two components.
Case_C3$_1$	C3	A component is being reused and adapted in order to save time for the short-term delivery of a customer feature that would allow the customization of some functionalities for the user. Instead, a better version is planned to be coded, which would bring better NFRs.
Case_C3$_2$	C3	A mediation layer is written without satisfying scalability requirements. The solution is going to work now, but cold be refactored to be scalable for later.
Case_D$_1$	D	An interface is growing sub-optimal because of the backward compatibility required by previous customers.
Case_D$_2$	D	An internal interface is not well defined. A large number of components are already using it, which will need to be refactored together with the interface.
Case_E$_1$	E	Business logic was embedded in the dialogs of the UI.
Case_F$_1$	F	A database component does not provide a standard API, and an application is using the private API.

3.1 Initial method (design and development)

The initial method, called AnaConDebt (see later the reason for the name) was created according to several sources related to previous empirical and theoretical work on ATD. The method is outlined in Figure 2 and it's an investigation device that includes the analysis of the difference between two scenarios (current and future) with respect to principal and interest of the ATD item: however, such difference is not calculated directly but through a list of factors (explained later) for which data need to be collected. The method gives, as output, some summarized information in form of *indicators*, which would aid the decision of the stakeholders on if and when to repay the principal of the ATD item. Even though the final goal for the method would be to become completely automated, given the current state of the art and practice, such goal is not achievable in the scope of this paper.

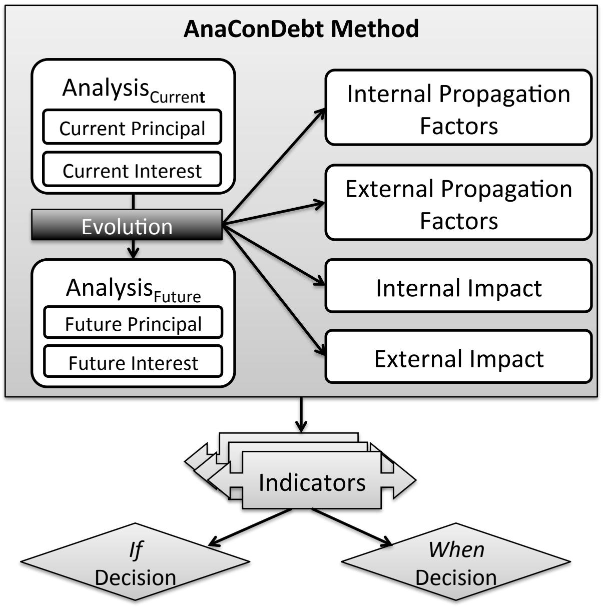

Figure 2 The AnaConDebt method and its output for decisions

3.1.1 Calculating technical debt evolution over time

In order to understand *if* and *when* to refactor the ATD item (RQ1), the method needs to analyze the principal and the interest in the current situation and report if they change with respect to in future scenarios (RQ2). We assume that the ATD item has already been identified at this point in time, and the stakeholders need to decide if to repay the principal. This is a reasonable assumption, since recent evidence suggests that ATD is rarely completely avoidable [3]. The main challenge in the decision is to understand when the principal or the interest would grow. We therefore included two different analyses in the method:

- Analysis$_{Current}$: analysis of the current situation for the ATD item: we investigated the *Source* of the problem (the part of the system in which the debt is located), the *Current Principal* and the *Current Interest.*
- Analysis$_{Future}$: analysis to estimate the growth of the impact of the ATD item and the growth of the refactoring, in order to understand what was the stakeholders' current knowledge and what information was needed in order to take a decision. We calculate the *Future Principal* and the *Future Interest.*

We therefore proceeded to investigate how the interest of the ATD item would grow between these two points in time, as explained below. This is done by comparing Analysis$_{Current}$ with Analysis$_{Future}$ in order to calculate the delta (*Evolution* in Figure 2) between the interest in the future with respect to the current situation, as explained in the next section.

3.1.2 Calculating the growth of Interest (RQ2)

In order to estimate the growth of interest, we need to understand the impact of the sub-optimal solution with respect to Analysis$_{Current}$ and Analysis$_{Future}$. In the initial method, we therefore included the investigation of two kinds of impacts: the *internal impact*, or else the extra-cost related to the new development or maintenance of the system, and the *external impact*, or else the impact of the sub-optimal solution on the customers or other stakeholders of the system. As examples we take two cases in consideration. In the first case, the software does not have an optimal architectural structure: for example, the concerns are not well separated and every time a developer needs to change something, s/he needs to modify many parts of the system, which causes extra effort in implementation and testing. This kind of impact is of internal kind, since the customer is not affected. In the second case, a complex design is causing the developers to incur in many errors, which makes the code defect-prone. In this case, the impact is on the external quality of the software, since the customer can potentially experience the bugs.

In order to calculate the interest, in the initial method we used a model described in a previously published paper [4], in which it was recognized how the technical debt propagates in the system in a *contagious* manner. In order to assess the *contagiousness* of the case, and therefore to estimate the growth of interest, we started by investigating the *Propagation Factors* responsible for the growth of the interest. In other words, we needed to understand what factors were causing the interest to grow from $\text{Analysis}_{\text{Current}}$ to $\text{Analysis}_{\text{Future}}$.

In the initial method, we included the propagation factors related to the growth of the system in terms of developed software, as explained in [4]: the growth of the software related to the *Source* of the debt (or else the part of the system affected by the TD at current time) and the growth of the software in other components that were *dependent* to the source of the debt.

In order to explain how this was done in the method with a concrete example, we take in consideration a real case (described as Case_F_1 in Table 1). In Case_F_1 a general component (GC), containing business logic, depends on a database component (DC). The GC should have been communicating with the DC through a standard API. Instead, the GC called the private APIs of the DC: this represents the *architectural debt* (since there is a suboptimal solution in place). Each time a development team adds a method to GC that calls the internal APIs of the DC, the method will be using calls to internal APIs. In case the DC would need to be replaced (the reason for having the standard API in order to implement evolvability of the system), the cost of refactoring would have grown (and this represents one kind of interest, as explained in section 2) at each new method implemented, since such method would need to be refactored. In this case, we say that the growth of the code in GC is an *external propagation factor,* which contributes to the growth of the principal. Similarly, this might involve the growth of the software within DC. In such case, we call it the *internal propagation factor*.

These components are outlined in Figure 2: they would help understanding the evolution of the Interest (and the Interest on the Principal) over time (RQ2) and therefore they could be used in the formulas described previously, which would aid the decision of *if* and *when* to refactor the ATD item (RQ1). Note that in the initial method we do not specify concrete components to be analyzed: this was done according to each case. By refining multiple times the initial method during the study, we obtained a final version of the method with the concrete propagation factors and internal and external impacts that should be investigated according to our observation of the concrete cases (see section 4).

3.2 Iterative application and refinement of the method (demonstration and evaluation)

Instead of creating the method offline, evaluating it once and reporting on its success/failure, we decided to empirically and iteratively develop, refine and evaluate the method multiple times with the industrial stakeholders. In our opinion, this would yield results that would be more in line with the practical needs of the stakeholders.

For each part of the method described earlier, we asked the practitioners to discuss the case mentioning the important factors for the decision, and providing as precise information as it was possible to be retrieved in the organization. In some cases it was possible to use quantitative data provided by project management or by analyzing the source code. In some other occasions, especially for the estimations concerning future scenarios, the experience of the experts was used. Since these measures depended greatly on the different organization analyzed, instead of reporting the various measures, we identified the key factors that would drive the data collection. Using a checklist with these factors provides a useful structure for reasoning around ATD and helps grounding the decisions to existing and objective data rather than only on the experts' subjective experience. Also, the architects can better visualize the most important factors for analysis and communication purposes.

It was extremely important, in this phase, for the researchers to understand how to apply the method in order to match the theoretical concept to the jargon and the terminology used in the company. For example, we almost never used the terms *debt*, *interest* and *principal*, but we asked questions such as *"what is the part of the architecture that is considered sub-optimal"* to identify the debt or *"what impact or extra-cost are you experiencing?"* in order to estimate the interest. This was important for the internal validity of the results (see section 5).

At each iteration, we evaluated the method together with the practitioners according to the following checklist:

- The improvements made by the method to their practice
- The missing parts of the method
- What parts were not satisfying the following requirements:
    - Applicability of the method
    - Cost/Benefit of the method
    - Maintainability of the artifacts created with the method
    - The involved knowledge that needed to be retrieved in the organization (when the architects' knowledge was not enough). This was important to understand the overhead necessary to retrieve the data for the stakeholders.

We also recorded the session and analyzed them afterwards. In the analysis, we identified the practitioners' statements that were leading to possible improvements for the method. We used a coding scheme for qualitative data analysis that would allow us to map quotations to new factors to be included in the method or to more generic improvements statements (e.g. data that was found difficult to retrieve, such as probabilities of events to occur).

Once we found the "delta" between the employed method and the needs of the practitioners according to the evaluation, we improved the method and we applied it again in new cases. When analyzing the last cases, we found less and less deltas between our method and the evaluation data, which assured that we were correctly refining the method. As an example of the delta, we mention the indicator concerning the non-functional qualities that were worsening over time: in the first representation we asked the architects to list them, but they found this process difficult to follow. In the second iteration we used a list of NFR extracted from the ISO standard [19] together with their explanation, and the stakeholders found this a much better approach. Also, instead of having an indicator with a trend for the NFR, they preferred to have just a list of the NFR that were getting worse.

3.3 Final evaluation

After the method was applied 11 times, we applied the method one last time and we extended the evaluation with additional questions and data collection, in order to have a more quantifiable result. We asked both about the method and the report with the summarized indicators. The participants in the final evaluation were all architects. We asked the following questions:

Q1. Were the method and the indicators useful for understanding the growth of principal and interest and therefore the risk of the Architectural Technical Debt issue?

Q2. Did this method and report increase your overall knowledge of the case?

Q3. How did the method improve the architects' work?

Q4. Do you think that this document led to a better estimation of the case?

Q5. Has this case contributed to change or reinforced a decision? For example, has the refactoring been scheduled?

Q6. Did this case contribute to make the architectural debt management process more proactive?

4. RESULTS AND ANALYSIS

In this section we report the findings with respect the RQs. Since the structure of the method has been already reported in Figure 2, we present the findings as the factors that the stakeholders found useful for estimating the growth of interest (RQ2). Then we show the results of the method applied to a concrete case, presenting the indicators that were used for the refactoring decision by the stakeholders (RQ1). Finally, we report the evaluation.

4.1 Components of AnaConDebt (RQ2)

4.1.1 Time spans for ATD evolution analysis

As described in the initial method section, the interest should be compared in the current situation and in a chosen future scenario ($\text{Analysis}_{\text{Current}}$ and $\text{Analysis}_{\text{Future}}$) in order to understand if and when the refactoring should be performed. During the application of the method, we found that the stakeholders preferred to applied the $\text{Analysis}_{\text{Future}}$ in 3 different future scenarios: *short-term*, *medium-term* and *long-term*. The actual time related to each timespan greatly depended on the company that we were applying the method with. The difference is based on the internal milestones and iterations that the companies used for the releases. For example, in one company the short-term was 4 months, while in another company it was 6 months, or in some cases even 1 year. Long term was chosen to be between 18 months and 3 years.

The participants found especially useful to compare the growth of the interest in the short term and in the long term (considering the latter one as the total interest, as explained in section 3). By selecting the short-term time-span as the length of internal iteration, this strategy would help them understanding if the ATD item would need to be refactored within the next iteration or if it could wait. In case they would decide to postpone the refactoring, they could repeat the method in the following iteration updating the data with more precise estimations for the medium-term timespan (which would become the short-term one in the next iteration). This allowed the stakeholders to continuously monitor the ATD items and the growth of interest.

4.1.2 Estimating the growth of the interest (RQ2)

As stated in section 3 and visible in Figure 2, the growth of interest is determined by *propagation factors* (internal and external) and/or other *impacts* (internal and external).

We recall here that the *propagation factors* are factors that cause the ATD to propagate in other parts of the system, thus creating more interest on the principal (causing the principal itself to grow) and an amplified interest. The following propagation factors were found useful by the practitioners for estimating the growth:

- *Internal propagation factors*: *growth of the source.* If the ATD was included in a single component, we estimated how many new functionalities would be included in the component in the chosen time span. If the ATD did already encompass more components, we estimated the sum of the new functionalities that needed to be added to all such components. This datum was used for calculating the interest on the principal, since it was quite likely that adding functionalities to the same part of the system were the ATD was located, would need to be refactored if the refactoring would been postponed.
- *External propagation factors: number of ATD-related increments planned.* One important datum to use was the number of increments (for example, new features, but this could be expressed differently in different organizations) that were planned in the roadmaps and that would involve the ATD. In order to do this, the architects analyzed the feature roadmap created by the project and product management. We examined if the features that were going to be developed would interact with the ATD. If so, we estimated a refactoring cost for those features as well if the ATD would not be refactored before such addition, which constituted an additional interest on the principal. This factor was also used for estimating the internal impact (interest) of the ATD, as explained below for other impacts (impact of development speed and impact on maintainability).
- *External propagation factors: number of external users in the ecosystem*. In some cases, the developed software was not only used internally in the organization, but could be used by many other users (for example, clients). The cases from company B were especially enlightening about this factor, since the company was developing part of the software that was released as an open-source component in an ecosystem. Therefore, many other developers were going to use the system. When an ecosystem is involved, it might be difficult to estimate a precise number of possible users of the developed system. However, the architects estimated this as a generic high number, which was enough for understanding that, once the ATD was released in the ecosystem, it would have been very costly to refactor all the clients. In that case, the refactoring was conducted right away, before the release (in both cases Case_B_1 and Case_B_2).
- *Internal propagation factor: growth of complexity as a multiplier.* In some cases, the stakeholders knew that the complexity of the components that would need to be refactored would grow over time, and it would therefore become more expensive to refactor them. They decide to apply a linear multiplier to the cost of refactoring as time passed, simulating the increased cost to change more complex code.

Besides the propagation factors, we found that there were other impacts that needed to be taken in consideration to calculate the interest. These impacts could either grow according to the propagation factors, or depending on other factors, totally unrelated to the propagation of the ATD through the system:

- *Impact on development speed*: for this step, we used the propagation factor *number of increments*, i.e. the number of increments (for example features) planned for the analyzed time-span. We estimated how much the development teams would spend in extra-effort if the ATD would not be removed before the features were implemented. For example, in one case we estimated that the presence of ATD would decrease feature development speed of 30%.

- *Impact on maintainability (internal quality)*: we estimated a risk of the bug proneness and the consequent extra-cost spent in fixing bugs by the teams. For example, in one case the overhead was growing of 10% for each new increment added to the system. We set this quality aside from the following ones, since the architects put a lot of emphasis on this one, and in some case they precisely quantified this impact.
- *Impact on qualities:* there are several qualities of the system that might be affected by the ATD item. The refinement of the method led us to add a list of the qualities extracted from the ISO standard [19] (together with their explanation). The stakeholders would pick the qualities that were considered more important for the developed system, rated them by importance (3-5) and then estimating if such qualities would be decreasing over time because of the ATD. For example, in one case it was found that the ATD item would have hindered the usability of the GUI for the customer. Using a checklist in this step was quite appreciated by the stakeholders, who would be sure to not leaving out important qualities from the analysis. On the other hand, different domains would have different importance for different qualities (as an example, in one case reusability was considered extremely important, while in another domain it was not that important).
- *Impact on learning:* in case the refactoring would be postponed and new employees would be hired, the new employees would need to learn the system twice, before and after it was refactored. This was considered as an extra-cost of delaying the refactoring, which can be considered interest on refactoring.
- *Impact on revenues:* in some cases, waiting to refactor would block the possibility of selling features separately from the current offer, which would prevent the organization from obtaining more revenues from such features.
- *Other costs:* in some cases, waiting to refactor would cause the organization in incurring in other kinds of costs, but these cases were quite specific and we have not reported them here. However, it's quite likely that each organization would have such costs, and it's useful to add such costs in the equation when found.

4.2 Indicators for ATD refactoring decision (RQ1)

The results of the method were summarized in a combination of different visualizations complemented with textual explanations. The stakeholders found difficult, in practice, to summarize the whole information from the process in just one indicator that would drive the whole refactoring decision. The analysis yielded the following results with respect to the key components. We show only the relative measures (for confidentiality reasons) taken from the last case analyzed, the one on which we have conducted the evaluation.

4.2.1 Growth of Interest on the Principal

The interest on the principal is the cost of refactoring growing over time because of the propagation of the ATD in the system. For this calculation, we have used all the propagation factors reported earlier. We cannot report absolute values since such data is confidential, but the graphs show relative values with respect to the principal calculated at the current time of analysis.

In Figure 3 we can see how the cost of refactoring the source of ATD (considering the internal propagation factors) in the long term would grow up to 60% (1.6 times the original refactoring cost) more than the cost of refactoring now (current).

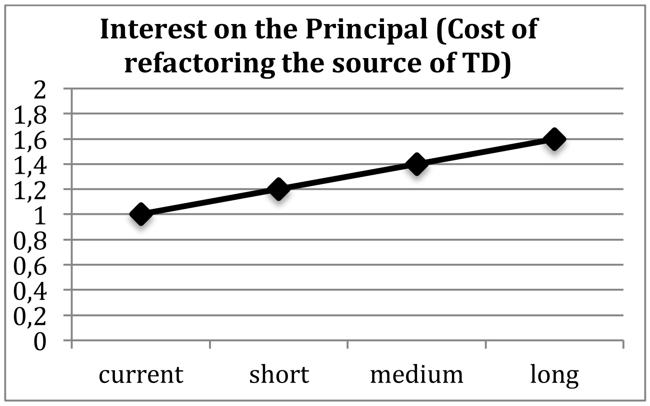

Figure 3 Relative growth of the cost of refactoring the source of ATD

The graph in Figure 4 shows (using a relative scale) how the interest on the principal would grow according the external propagation factors: in the short term, developing the increments (features or new components) would cost 1.5 more than in the current situation (50% more), while in the long term it would cost 3 times as much (200% more). This is due to the planned development of new increments, according to the roadmap, that would be dependent on the ATD item.

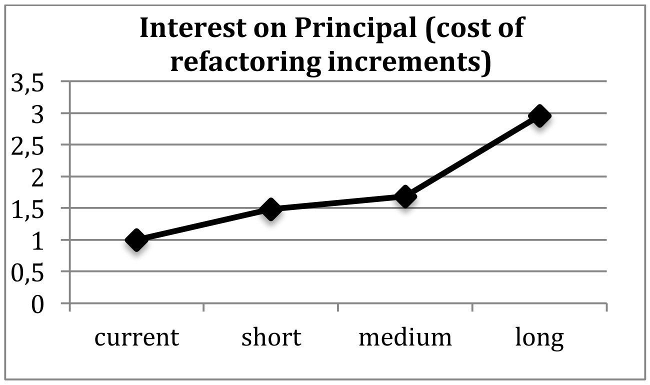

Figure 4 Relative growth of cost of refactoring with respect to the number of increments planned for the development

4.2.2 Growth of Impact

The information yielded by the method allowed us to build the following graph about the interest paid in development speed (Figure 5). By estimating the wasted time in interest by non refactoring, the graph shows (as a relative measure on the interest paid in the short term) that in the medium term the ATD would cause the speed to be affected as much as 1.5 times (50% more) the short-term interest, and in the long run the interest would be 4 times (300% more) as much as the interest in the short term. This does not mean that the whole speed becomes 4 times worse, being this a relative measure, but it shows that the accumulation of interest would be 4 times as much as it will be in the short term. For example, if in the short term the waste in speed would slow the development of 1 increment, in the timespan between the medium term and the long term the waste would be of 4 increments that would not be developed because of the interest. This visualization was considered particularly useful by the architects, who would be able to understand how many increments they would save by doing the refactoring.

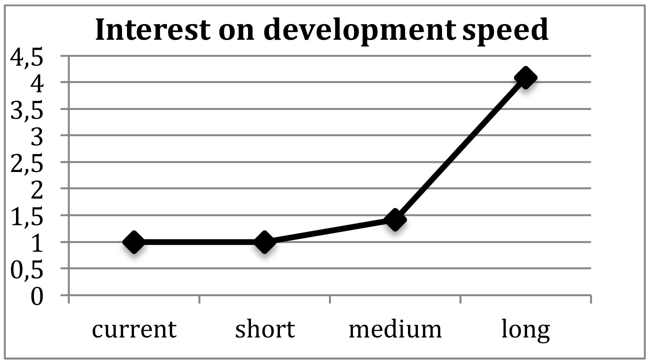

Figure 5 Relative growth of time wasted in paying the interest

The rest of the information about the interest was decided to be visualized without graphs (for example other impacts and qualities). In this specific case the growth of maintainability (time spent on fixing defects) was accounted for another 10% of time wasted, but the stakeholders did not find important to show a graph, which would just present an already known linear interest.

As for the other qualities, the method helped to point out that, if not refactored, the ATD would cause the following four qualities, considered critical for the specific company (according to the ISO standard [19]), to decrease: *Compatibility*, *Reliability*, *Analyzability* and *Modifiability*.

4.2.3 Refactoring decisions

We first analyze the indicators reported in the previous section, and we show what conclusions it is possible to draw by decomposing the equations presented in section 2 in the components for which we obtained the indicators. The real conclusions drawn by the architects cannot be reported here for confidentiality reasons, but we will show with two simple examples (assigning unitary values to the various contributing factors) what kind of reasoning was used to reach a decision.

From the indicators reported in the previous section, we can see how both the principal and the interest grow over time (in this specific case). One clear reason is that the ATD item is *contagious* [4], or else it's likely to propagate into the system as it grows. For example, in this specific case the ATD interest would grow for any new-implemented feature, and it becomes more and more costly both to refactor it and to deal with. But there were other unrelated factors that would determine the growth of the interest, such as the number or new employees that needed to be hired.

To the question *if* the ATD item should be refactored, we related the obtained data to equation (1) for refactoring decision introduced in section 2. In this case the Total Interest *TInterest* is calculated as the long-term interest. We show here a simple example of the line of reasoning by decomposing the formulas with respect to the factors and indicators and, for the sake of understanding, we will associate a unitary value to the various factors in the formula.

First of all we calculate *CPrincipal* as the cost of refactoring the source (CRS) plus the cost of refactoring the increments (CRI). As for *TInterest*, we use the total interest (calculated in the long term, LTI as 4 times the current interest CI) plus the interest in maintainability of 10% each term (3*IM) and the interest in a number (4 in this case) of quality requirements decreased (IQ).

$$\frac{CRS + CRI}{4 * CI + 3 * IM + 4 * IQ}$$

Assigning a unitary value (for demonstration purposes) to each of the components, the result of this example is 2/11. This number, in an oversimplified way, would tell the stakeholders that the refactoring is beneficial because paying the CPrincipal, in this case 2, is less than paying the interest, in this case 11. Although this number in practice is not meaningful without precise data, it serves here as a means to explain that the stakeholders have taken into account the cost of refactoring together with the cost of the interest and evaluated them according to the data collected through the method.

As for *when* to refactor, we need to decompose equation (2), where F_1 is the choice of either the short-term or the medium-term time span in the graphs and *TInterest* is the total interest calculated in the long-term time span.

For demonstration purposes, we choose F_1 as the medium term. Using the factors listed in the previous sections, we want to calculate *F_1Principal*, which is composed by *CPrincipal* plus the whole interest on *CPrincipal*, or else *F_1IPrincipal* (according to equation (3)). Decomposing in the known factors, this is represented by the cost of refactoring the source (CRS) plus the interest on it, I(CRS), which is 40% of CRS. In total, we have that the cost of refactoring the source at F_1 is CRS plus I(CRS), or else 1.4*(CRS). The same reasoning can be done with the other factor, the cost of refactoring the increments, I(CRI), which is 70% of CRI and therefore the total cost of refactoring the increments at F_1 is 1.7(CRI). As for *F_1Interest*, we know that at medium term (F_1) the interest on development speed is 50% more than the one in the current situation (CI), which is equal to 1.5(CI). Then, we have to calculate the increment in maintenance IM, which grew by 10% (between short-term and medium-term) and the quality IQ. Since we do not have exact numbers for the quality requirements, we assume that at medium term they worsened approximately half of the long term, so we use a multiplier of 2 instead of 4. In the end, the left part of the equation (2) is equal to the formula:

$$\frac{F_1Principal}{TInterest - F_1Interest} = \frac{1.4 * CRS + 1.7 * CRI}{(4 * CI + 3 * IM + 4 * IQ) - (1.5 * CI + 1 * IM + 2 * IQ)}$$

Once again, for demonstration purposes we can assign a unitary value to the various components. The result of this equation is 3.1/4.5. If we take in consideration the difference between this number (which represents the gain of refactoring in the medium term) and the previous number (2/11, or else the gain from refactoring in the current situation), the result is (3.1/4.5 – 2/11), or else approximately – 0.51. This negative number would tell us that waiting to refactor in the medium term yields approximately 50% less benefits than refactoring now. Therefore, this would clearly aid the refactoring decision, since the result shows that it would be better to refactor now.

It is important to understand that the formulas might not be used with exact numbers in practice. What the stakeholders would do, in the current state of practice, is to collect the data (or at least estimate the trends of the important components), analyze the indicators and the textual explanations and apply the formulas in an intuitive way of reasoning (for example assigning customized weights to the factors) rather than as a strict measurement:

- First the practitioners would decide *if* it would be worth employing the estimated costs calculated for the principal in order to avoid the interest, which consists of reduced development speed, increased maintenance cost and decreased quality.

- Then, the practitioners would decide if to repay the debt or wait to refactor with respect to a given term (short or medium): by estimating and visualizing the growth of the principal and the interest, it became clear if the refactoring will or will not be as beneficial in the future as it is in the current situation. If the result from the report showed that it would not be convenient to refactor now, the analysis would be repeated in the following iteration to understand if the situation on the interest had changed over time.

Future work aims at increasing the precision of these estimations with more precise measures. We have to keep in mind that ATD management is prevalently a risk assessment practice, and until better prediction models have been created to predict the cost of architecture anti-patterns (as confirmed in a recent systematic review [8]), the architects' decisions are based on gut feeling and estimations based on their subjective experiences [20]. However, structuring such process of decision making by applying the method AnaConDebt, has been proven useful in order to make the decision around ATD refactoring based to concrete factors that are causing the interest to be paid in the future, and therefore it would explicit and increase the architects' knowledge for making informed decision on the ATD refactoring cases.

4.2.4 Evaluation of the method

We report the evaluation with respect to the last application of the method on a real case (the one described in the previous section), according to the questions reported in the research design section.

Q1: the participants were quite satisfied with how the method was useful to summarize the data in indicators that clearly showed the trends of the growth of interest.

Q2: The architects confirmed that the indicators yielded by the method contributed to increase awareness of the principal and interest related to the case in all part of the organization, not only for architects, but also for developers and managers.

Q3 – Q4: the architects recognized that using the AnaConDebt, based on the Technical Debt framework, to analyze the case, caused a clear change in the common mindset, and contributed to take a more informed decision on refactoring. Also, the architects said that currently the employees do not think in terms of Technical Debt when they decide to refactor, but they just consider it as something that has to be done. This creates a frustration for the stakeholders since they do not know how to prioritize the refactorings among themselves and with respect to feature development. However, using AnaConDebt, it becomes possible to compare two ATD refactorings and it also becomes possible to communicate the business value related to the ATD refactoring to the entire development organization and especially the management, who can compare the business value of the refactoring with the business value of the features.

Q5 – Q6: The refactoring of the analyzed case was indeed prioritized, and the method AnaConDebt was especially considered useful to speed up the start of such refactoring, since it highlighted the information that the principal and the interest were going to grow. Therefore, in practice the usage of the method contributed to proactively taking decisions on refactoring.

In summary, the method was found useful both for increasing the awareness of the development organization about the importance and the urgency of proactively refactoring ATD. The results were found especially useful for communicating and discussing the need of refactoring with the management part of the organization (which is the main goal of the Technical Debt metaphor).

5. LIMITATIONS AND RELATED WORK

5.1.1 Limitations and threats to validity

There are some limitations in this study. Most of the effort measures used for calculating the different components are based on the experiences of the experts involved. This might have injected some bias in the study. However, the overall framework and the components that can be used to estimate ATD refactoring are quite generalizable, the indicators have been considered as useful and the decision making process proper. The different metrics that can be used as input needs to be further studied, however they usually depend on the different company. Another limitation is the learning curve: it took a while before the architects would employ the method, but after the sessions, they recognized that the method was very logical and that the process would probably be much shorter as the method was repeated and the tasks of collecting information split among the architects.

As for the threats to validity, we refer to the ones listed for case study research [21]: construct-, internal-, external validity and reliability. As for construct validity, we made, as explained in section 3, to not refer to "technical debt" or "interest" but rather refer to costs and impacts of the ATD item. We also explicitly spent part of the sessions agreeing on what kind of architectural issues were referred to as Technical Debt. Since internal validity concerns with the studies on cause-effect phenomena, it is not applicable in this study. The possible external validity threats are that we have used the method on large companies, so we cannot generalize to small and medium companies. However, we contribute with a method that has been evaluated in 6 large organizations, which provides a higher degree of generalizability. Lastly, a possible threat to the reliability of the study is the introduction of the researcher's bias in the interpretation of the data and their conversion to the indicators. However, the continuous evaluation of the method with the practitioners assures that the data were validated multiple times.

5.1.2 Related Work

Only a few other empirical works have been done on architectural refactoring. For example, in [20] the authors use a cost-benefit model based on work tickets in order to estimate the effort that could be saved by a refactoring to decouple the architecture. The study takes in consideration only one aspect of the architecture (coupling) and one effect (effort) in one case (Vistaprint). Also, the focus of such work is not related to *when* to refactor. The same discussion can be done with respect to the work done in [2], where the authors analyzed the dependencies of one case in order to understand if to apply the refactoring in the current situation or not. The approach has some similarities with our method, for example in calculating the upcoming features and the rework that needs to be done on them, but it's limited to such common factor. In our work we provide a more comprehensive and holistic method in which we present several components to be analyzed in order to decide on ATD refactorings, but we do not specify detailed measures. Our results are based on 12 studied ATD items related to 6 organizations, which add a degree of generalizability to our results (differing from the other work). However, we have mainly used the method in large embedded software companies. For such domain the method was proven useful, and we have some preliminary evidence that it can be generalized for pure software development in large companies (company F). However, we cannot assure that the same approach would be useful for small and medium companies.

6. CONCLUSION

Refactoring Architectural Technical Debt is a task that, for the architects and for large software organization, is risky, difficult to estimate and difficult to be prioritized since the lack of a clear business value for the management [20]. In practice, usually the refactoring is postponed until it is too late and repaying the principal becomes too expensive with respect to paying the interest. As a consequence, the refactoring might not be conducted at all, which leads to slow down the development and brings development crises in which there is no delivered value [3].

In order to tackle this problem, we have empirically developed and evaluated a method, AnaConDebt, with the continuous contribution of several architects in 6 organizations and continuously evaluated it in practice by analyzing 12 concrete cases of ATD. As far as we know, this is the only study in which a method for ATD management has been developed in collaboration with this much industrial input.

The aim of study was to provide a tool for taking decisions on *if* and *when* ATD should be refactor (RQ1). We have shown how the method can be used for such purpose, and how it was found useful by the architects. The method provides indicators that would estimate the important factors responsible for the growth of interest (RQ2) and therefore would warn the organization (including non-technical stakeholders) that the refactoring is urgent. As an example of the impact on non-technical stakeholders, a product manager participating in one of the session commented: "*I entered in this room with an idea, but I'm leaving with a completely new perspective"*.

7. ACKNOWLEDGMENTS

We thank the companies that are partners of the Software Center and especially the architects group. We would also thank Mattias Tichy and Michel Chaudron for their suggestions.

8. REFERENCES

[1] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice," *IEEE Softw.*, vol. 29, no. 6, pp. 18–21, 2012.

[2] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In Search of a Metric for Managing Architectural Technical Debt," in *2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)*, 2012, pp. 91–100.

[3] A. Martini, J. Bosch, and M. Chaudron, "Investigating Architectural Technical Debt Accumulation and Refactoring over Time: a Multiple-Case Study," *Inf. Softw. Technol.*

[4] A. Martini and J. Bosch, "The Danger of Architectural Technical Debt: Contagious Debt and Vicious Circles," in *accepted for publication at WICSA 2015*, Montreal, Canada.

[5] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, New York, NY, USA, 2015, pp. 50–60.

[6] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *J. Syst. Softw.*, vol. 86, no. 6, pp. 1498–1516, Jun. 2013.

[7] W. Cunningham, "The WyCash portfolio management system," in *ACM SIGPLAN OOPS Messenger*, 1992, vol. 4, pp. 29–30.

[8] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *J. Syst. Softw.*, vol. 101, pp. 193–220, Mar. 2015.

[9] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, and others, "Managing technical debt in software-reliant systems," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 2010, pp. 47–52.

[10] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Q. Da Silva, A. L. M. Santos, and C. Siebra, "Tracking technical debt—An exploratory case study," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, 2011, pp. 528–531.

[11] C. Seaman, Y. Guo, N. Zazworka, F. Shull, C. Izurieta, Y. Cai, and A. Vetro, "Using technical debt data in decision making: Potential decision approaches," in *2012 Third International Workshop on Managing Technical Debt (MTD)*, 2012, pp. 45–48.

[12] J.-L. Letouzey, "The SQALE Method for Evaluating Technical Debt," in *Proceedings of the Third International Workshop on Managing Technical Debt*, Piscataway, NJ, USA, 2012, pp. 31–36.

[13] A. Nugroho, J. Visser, and T. Kuipers, "An empirical model of technical debt and interest," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, New York, NY, USA, 2011, pp. 1–8.

[14] K. Schmid, "A formal approach to technical debt decision making," in *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*, 2013, pp. 153–162.

[15] P. Kruchten, "What do software architects really do?," *J. Syst. Softw.*, vol. 81, no. 12, pp. 2413–2416, Dec. 2008.

[16] A. Martini, L. Pareto, and J. Bosch, "Role of Architects in Agile Organizations," in *Continuous Software Engineering*, J. Bosch, Ed. Springer International Publishing, 2014, pp. 39–50.

[17] M. A. Babar and I. Gorton, "Comparison of scenario-based software architecture evaluation methods," in *Software Engineering Conference, 2004. 11th Asia-Pacific*, 2004, pp. 600–607.

[18] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A Design Science Research Methodology for Information Systems Research," *J. Manag. Inf. Syst.*, vol. 24, no. 3, pp. 45–77, Dec. 2007.

[19] ISO - International Organization for Standardization, "System and software quality models." [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=52075. [Accessed: 08-Mar-2015].

[20] J. Carriere, R. Kazman, and I. Ozkaya, "A cost-benefit framework for making architectural decisions in a business context," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, 2010, vol. 2, pp. 149–157.

[21] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empir. Softw. Eng.*, vol. 14, no. 2, pp. 131–164, Dec. 2008.